"В своей книге Герберт Шилдт освещает все, что требуется знать для применения C# 4.0 на практике".
Майкл Ховард, корпорация Microsoft

ПОЛНОЕ РУКОВОДСТВО

C# 4.0

Книга содержит:

- Полное описание средств языка C#
- Подробное рассмотрение новых средств в версии C# 4.0, в том числе PLINQ, библиотека TPL, именованные и необязательные аргументы, динамический тип данных и многое другое
- Сотни простых и понятных примеров программ с комментариями

БЕСПЛАТНЫЙ КОД В ИНТЕРНЕТЕ

БОНУС!
WWW.OSBORNE.COM

Полностью исправленное и обновленное издание классического руководства по C# 4.0

Герберт Шилдт
Автор лучших книг по программированию, проданных миллионными тиражами по всему миру!

ПОЛНОЕ
РУКОВОДСТВО

C# 4.0

The Complete Reference

C# 4.0

HERBERT SCHILDT

Learn more. Mc Graw Hill Do more.
MHPROFESSIONAL.COM

ПОЛНОЕ РУКОВОДСТВО

C# 4.0

ГЕРБЕРТ ШИЛДТ



Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Шилдт, Герберт.

Ш57 C# 4.0: полное руководство. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2011. — 1056 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1684-6 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства McGrow-Hill Higher Ed.

Authorized translation from the English language edition published by McGraw-Hill Companies, Copyright © 2010
All rights reserved. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2011

Научно-популярное издание
Герберт Шилдт
C# 4.0: полное руководство

Литературный редактор *Е.П. Перестюк*
Верстка *Л.В. Чернокозинская*
Художественный редактор *С.А. Чернокозинский*
Корректор *Л.А. Гордиенко*

Подписано в печать 17.09.2010. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 85,14. Уч.-изд. л. 51,55.
Тираж 1500 экз. Заказ № 24007.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1684-6 (рус.)
ISBN 0-07-174116-X (англ.)

© Издательский дом "Вильямс", 2011
© by The McGraw-Hill Companies, 2010

Оглавление

Благодарности 23
Предисловие 25
ЧАСТЬ I. ЯЗЫК C# 29
Глава 1. Создание C# 31
Глава 2. Краткий обзор элементов C# 41
Глава 3. Типы данных, литералы и переменные 67
Глава 4. Операторы 97
Глава 5. Управляющие операторы 121
Глава 6. Введение в классы,объекты и методы 147
Глава 7. Массивы и строки 177
Глава 8. Подробнее о методах и классах 209
Глава 9. Перегрузка операторов 269
Глава 10. Индексаторы и свойства 303
Глава 11. Наследование 329
Глава 12. Интерфейсы, структуры и перечисления 375
Глава 13. Обработка исключительных ситуаций 403
Глава 14. Применение средств ввода-вывода 431
Глава 15. Делегаты, события и лямбда-выражения 473
Глава 16. Пространства имен, препроцессор и сборки 513
Глава 17. Динамическая идентификация типов, рефлексия и атрибуты 537
Глава 18. Обобщения 575
Глава 19. LINQ 637
Глава 20. Небезопасный код, указатели, обнуляемые типы и разные ключевые слова 681
ЧАСТЬ II. БИБЛИОТЕКА C# 717
Глава 21. Пространство имен System 719
Глава 22. Строки и форматирование 783
Глава 23. Многопоточное программирование. Часть первая: основы 833
Глава 24. Многопоточное программирование. Часть вторая: библиотека TPL 885
Глава 25. Коллекции, перечислители и итераторы 923
Глава 26. Сетевые средства подключения к Интернету 1011
Приложение. Краткий справочник по составлению документирующих комментариев 1039
Предметный указатель 1044

Содержание

Об авторе 22
О научном редакторе 22

Благодарности **23**

Предисловие **25**
Структура книги 27
Книга для всех программирующих 27
Необходимое программное обеспечение 27
Код, доступный в Интернете 27
Что еще почитать 28
От издательства 28

ЧАСТЬ I. ЯЗЫК C# **29**

Глава 1. Создание C# **31**
Генеалогическое дерево C# 32
Язык C — начало современной эпохи программирования 32
Появление ООП и C++ 33
Появление Интернета и Java 33
Создание C# 34
Развитие C# 36
Связь C# со средой .NET Framework 37
О среде NET Framework 37
Принцип действия CLR 38
Управляемый и неуправляемый код 38
Общеязыковая спецификация 39

Глава 2. Краткий обзор элементов C# **41**
Объектно-ориентированное программирование 41
Инкапсуляция 42
Полиморфизм 43
Наследование 44
Первая простая программа 44
Применение компилятора командной строки csc.exe 45
Применение интегрированной среды разработки Visual Studio 46
Построчный анализ первого примера программы 50
Обработка синтаксических ошибок 53
Незначительное изменение программы 54
Вторая простая программа 54

Другие типы данных 57
Два управляющих оператора 58
Условный оператор 58
Оператор цикла 60
Использование кодовых блоков 61
Точка с запятой и оформление исходного текста программы 63
Ключевые слова C# 64
Идентификаторы 65
Библиотека классов среды .NET Framework 66

Глава 3. Типы данных, литералы и переменные 67

О значении типов данных 67
Типы значений в C# 68
Целочисленные типы 69
Типы для представления чисел с плавающей точкой 71
Десятичный тип данных 73
Символы 74
Логический тип данных 75
Некоторые возможности вывода 76
Литералы 79
Шестнадцатеричные литералы 80
Управляющие последовательности символов 80
Строковые литералы 81
Более подробное рассмотрение переменных 83
Инициализация переменной 83
Динамическая инициализация 84
Неявно типизированные переменные 85
Область действия и время существования переменных 86
Преобразование и приведение типов 89
Автоматическое преобразование типов 90
Приведение несовместимых типов 91
Преобразование типов в выражениях 93
Приведение типов в выражениях 95

Глава 4. Операторы 97

Арифметические операторы 97
Операторы инкремента и декремента 98
Операторы отношения и логические операторы 101
Укороченные логические операторы 104
Оператор присваивания 106
Составные операторы присваивания 107
Поразрядные операторы 107
Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ 108
Операторы сдвига 114

Поразрядные составные операторы присваивания 117
Оператор ? 117
Использование пробелов и круглых скобок 119
Предшествование операторов 119
Глава 5. Управляющие операторы 121
Оператор if 121
Вложенные операторы if 122
Конструкция if-else-if 124
Оператор switch 125
Вложенные операторы switch 129
Оператор цикла for 129
Некоторые разновидности оператора цикла for 131
Оператор цикла while 137
Оператор цикла do-while 138
Оператор цикла foreach 139
Применение оператора break для выхода из цикла 139
Применение оператора continue 142
Оператор return 143
Оператор goto 143
Глава 6. Введение в классы,объекты и методы 147
Основные положения о классах 147
Общая форма определения класса 148
Определение класса 149
Создание объектов 153
Переменные ссылочного типа и присваивание 154
Методы 155
Добавление метода в класс Building 156
Возврат из метода 158
Возврат значения 159
Использование параметров 162
Добавление параметризированного метода в класс Building 164
Исключение недоступного кода 166
Конструкторы 166
Параметризированные конструкторы 168
Добавление конструктора в класс Building 169
Еще раз об операторе new 170
Применение оператора new вместе с типами значений 170
"Сборка мусора" и применение деструкторов 171
Деструкторы 172
Ключевое слово this 174
Глава 7. Массивы и строки 177
Массивы 177

Одномерные массивы 178
Многомерные массивы 182
Двумерные массивы 182
Массивы трех и более измерений 183
Инициализация многомерных массивов 184
Ступенчатые массивы 185
Присваивание ссылок на массивы 187
Применение свойства Length 189
Применение свойства Length при обращении со ступенчатыми массивами 191
Неявно типизированные массивы 192
Оператор цикла foreach 194
Строки 198
Построение строк 198
Обращение со строками 199
Массивы строк 203
Постоянство строк 205
Применение строк в операторах switch 206

Глава 8. Подробнее о методах и классах 209

Управление доступом к членам класса 209
Модификаторы доступа 210
Организация закрытого и открытого доступа 212
Практический пример организации управления доступом 212
Передача объектов методам по ссылке 218
Способы передачи аргументов методу 220
Использование модификаторов параметров ref и out 222
Использование модификатора параметра ref 223
Использование модификатора параметра out 224
Использование модификаторов ref и out для ссылок на объекты 227
Использование переменного числа аргументов 229
Возврат объектов из методов 231
Возврат массива из метода 234
Перегрузка методов 235
Перегрузка конструкторов 241
Вызов перегружаемого конструктора с помощью ключевого слова this 245
Инициализаторы объектов 246
Необязательные аргументы 247
Необязательные аргументы и перегрузка методов 249
Необязательные аргументы и неоднозначность 250
Практический пример использования необязательных аргументов 251
Именованные аргументы 252
Метод Main() 254
Возврат значений из метода Main() 254

Передача аргументов методу Main() 255
Рекурсия 257
Применение ключевого слова static 260
Статические конструкторы 265
Статические классы 266

Глава 9. Перегрузка операторов 269

Основы перегрузки операторов 270
Перегрузка бинарных операторов 270
Перегрузка унарных операторов 273
Выполнение операций со встроенными в C# типами данных 277
Перегрузка операторов отношения 281
Перегрузка операторов true и false 283
Перегрузка логических операторов 286
Простой способ перегрузки логических операторов 286
Как сделать укороченные логические операторы доступными для применения 288
Операторы преобразования 292
Рекомендации и ограничения по перегрузке операторов 297
Еще один пример перегрузки операторов 298

Глава 10. Индексаторы и свойства 303

Индексаторы 303
Создание одномерных индексаторов 304
Перегрузка индексаторов 307
Индексаторы без базового массива 310
Многомерные индексаторы 311
Свойства 313
Автоматически реализуемые свойства 318
Применение инициализаторов объектов в свойствах 319
Ограничения, присущие свойствам 320
Применение модификаторов доступа в аксессорах 320
Применение индексаторов и свойств 324

Глава 11. Наследование 329

Основы наследования 329
Доступ к членам класса и наследование 333
Организация защищенного доступа 336
Конструкторы и наследование 337
Вызов конструкторов базового класса 339
Наследование и сокрытие имен 343
Применение ключевого слова base для доступа к скрытому имени 344
Создание многоуровневой иерархии классов 346
Порядок вызова конструкторов 349
Ссылки на базовый класс и объекты производных классов 351

Виртуальные методы и их переопределение 355
Что дает переопределение методов 359
Применение виртуальных методов 360
Применение абстрактных классов 363
Предотвращение наследования с помощью ключевого слова sealed 367
Класс object 368
Упаковка и распаковка 370
Класс object как универсальный тип данных 372
Глава 12. Интерфейсы, структуры и перечисления 375
Интерфейсы 375
Реализация интерфейсов 377
Применение интерфейсных ссылок 381
Интерфейсные свойства 383
Интерфейсные индексаторы 385
Наследование интерфейсов 387
Сокрытие имен при наследовании интерфейсов 388
Явные реализации 388
Выбор между интерфейсом и абстрактным классом 391
Стандартные интерфейсы для среды .NET Framework 391
Структуры 391
О назначении структур 395
Перечисления 397
Инициализация перечисления 399
Указание базового типа перечисления 399
Применение перечислений 399
Глава 13. Обработка исключительных ситуаций 403
Класс System.Exception 404
Основы обработки исключительных ситуаций 404
Применение пары ключевых слов try и catch 404
Простой пример обработки исключительной ситуации 405
Второй пример обработки исключительной ситуации 407
Последствия неперехвата исключений 408
Обработка исключительных ситуаций — "изящный" способ устранения программных ошибок 410
Применение нескольких операторов catch 411
Перехват всех исключений 412
Вложение блоков try 413
Генерирование исключений вручную 414
Повторное генерирование исключений 415
Использование блока finally 416
Подробное рассмотрение класса Exception 418
Наиболее часто используемые исключения 420

Получение производных классов исключений 422
Перехват исключений производных классов 426
Применение ключевых слов checked и unchecked 428

Глава 14. Применение средств ввода-вывода 431

Организация системы ввода-вывода в C# на потоках 431
Байтовые и символьные потоки 432
Встроенные потоки 432
Классы потоков 432
Класс Stream 432
Классы байтовых потоков 434
Классы-оболочки символьных потоков 434
Двоичные потоки 436
Консольный ввод-вывод 436
Чтение данных из потока ввода с консоли 436
Применение метода ReadKey() 438
Запись данных в поток вывода на консоль 440
Класс FileStream и байтовый ввод-вывод в файл 441
Открытие и закрытие файла 441
Чтение байтов из потока файлового ввода-вывода 444
Запись в файл 446
Использование класса FileStream для копирования файла 448
Символьный ввод-вывод в файл 449
Применение класса StreamWriter 449
Применение класса StreamReader 451
Переадресация стандартных потоков 452
Чтение и запись двоичных данных 454
Класс BinaryWriter 454
Класс BinaryReader 455
Демонстрирование двоичного ввода-вывода 457
Файлы с произвольным доступом 461
Применение класса MemoryStream 463
Применение классов StringReader и StringWriter 465
Класс File 467
Копирование файлов с помощью метода Copy() 467
Применение методов Exists() и GetLastAccessTime() 468
Преобразование числовых строк в их внутреннее представление 469

Глава 15. Делегаты, события и лямбда-выражения 473

Делегаты 473
Групповое преобразование делегируемых методов 476
Применение методов экземпляра в качестве делегатов 477
Групповая адресация 478
Ковариантность и контравариантность 481

Класс System.Delegate 483
Назначение делегатов 483
Анонимные функции 483
Анонимные методы 484
Передача аргументов анонимному методу 484
Возврат значения из анонимного метода 485
Применение внешних переменных в анонимных методах 486
Лямбда-выражения 488
Лямбда-оператор 488
Одиночные лямбда-выражения 489
Блочные лямбда-выражения 492
События 494
Пример групповой адресации события 496
Методы экземпляра в сравнении со статическими методами в качестве обработчиков событий 497
Применение аксессоров событий 500
Разнообразные возможности событий 504
Применение анонимных методов и лямбда-выражений вместе с событиями 504
Рекомендации по обработке событий в среде .NET Framework 506
Применение делегатов EventHandler<TEventArgs> и EventHandler 508
Практический пример обработки событий 509

Глава 16. Пространства имен, препроцессор и сборки 513

Пространства имен 513
Объявление пространства имен 514
Предотвращение конфликтов имен с помощью пространств имен 516
Директива using 518
Вторая форма директивы using 520
Аддитивный характер пространств имен 521
Вложенные пространства имен 523
Глобальное пространство имен 524
Применение описателя псевдонима пространства имен :: 524
Препроцессор 528
Директива #define 529
Директивы #if и #endif 529
Директивы #else и #elif 531
Директива #undef 533
Директива #error 533
Директива #warning 534
Директива #line 534
Директивы #region и #endregion 534
Директива #pragma 534
Сборки и модификатор доступа internal 535

Модификатор доступа internal 536

Глава 17. Динамическая идентификация типов, рефлексия и атрибуты 537

Динамическая идентификация типов 537
Проверка типа с помощью оператора is 538
Применение оператора as 539
Применение оператора typeof 540
Рефлексия 541
Класс System.Type — ядро подсистемы рефлексии 541
Применение рефлексии 543
Получение сведений о методах 544
Вызов методов с помощью рефлексии 548
Получение конструкторов конкретного типа 550
Получение типов данных из сборок 555
Полностью автоматизированное обнаружение типов 560
Атрибуты 562
Основы применения атрибутов 563
Сравнение позиционных и именованных параметров 566
Встроенные атрибуты 570
Атрибут AttributeUsage 570
Атрибут Conditional 571
Атрибут Obsolete 572

Глава 18. Обобщения 575

Что такое обобщения 576
Простой пример обобщений 576
Различение обобщенных типов по аргументам типа 580
Повышение типовой безопасности с помощью обобщений 580
Обобщенный класс с двумя параметрами типа 583
Общая форма обобщенного класса 585
Ограниченные типы 585
Применение ограничения на базовый класс 586
Применение ограничения на интерфейс 594
Применение ограничения new() на конструктор 598
Ограничения ссылочного типа и типа значения 599
Установление связи между двумя параметрами типа с помощью ограничения 602
Применение нескольких ограничений 603
Получение значения, присваиваемого параметру типа по умолчанию 604
Обобщенные структуры 606
Создание обобщенного метода 607
Вызов обобщенного метода с явно указанными аргументами типа 609
Применение ограничений в обобщенных методах 610
Обобщенные делегаты 610

Обобщенные интерфейсы 612
Сравнение экземпляров параметра типа 615
Иерархии обобщенных классов 620
Применение обобщенного базового класса 620
Обобщенный производный класс 622
Переопределение виртуальных методов в обобщенном классе 623
Перегрузка методов с несколькими параметрами типа 625
Ковариантность и контравариантность в параметрах обобщенного типа 626
Применение ковариантности в обобщенном интерфейсе 626
Применение контравариантности в обобщенном интерфейсе 630
Вариантные делегаты 633
Создание экземпляров объектов обобщенных типов 635
Некоторые ограничения, присущие обобщениям 636
Заключительные соображения относительно обобщений 636

Глава 19. LINQ 637

Основы LINQ 638
Простой запрос 639
Неоднократное выполнение запросов 641
Связь между типами данных в запросе 642
Общая форма запроса 643
Отбор запрашиваемых значений с помощью оператора where 644
Сортировка результатов запроса с помощью оператора orderby 646
Подробное рассмотрение оператора select 649
Применение вложенных операторов from 653
Группирование результатов с помощью оператора group 655
Продолжение запроса с помощью оператора into 657
Применение оператора let для создания временной переменной в запросе 659
Объединение двух последовательностей с помощью оператора join 660
Анонимные типы 663
Создание группового объединения 666
Методы запроса 669
Основные методы запроса 669
Формирование запросов с помощью методов запроса 670
Синтаксис запросов и методы запроса 673
Дополнительные методы расширения, связанные с запросами 673
Режимы выполнения запросов: отложенный и немедленный 675
Деревья выражений 676
Методы расширения 678
PLINQ 680

Глава 20. Небезопасный код, указатели, обнуляемые типы и разные ключевые слова 681
Небезопасный код 681
Основы применения указателей 682
Применение ключевого слова unsafe 684
Применение модификатора fixed 685
Доступ к членам структуры с помощью указателя 686
Арифметические операции над указателями 686
Сравнение указателей 688
Указатели и массивы 688
Указатели и строки 690
Многоуровневая непрямая адресация 691
Массивы указателей 692
Создание буферов фиксированного размера 693
Обнуляемые типы 695
Основы применения обнуляемых типов 695
Применение обнуляемых объектов в выражениях 697
Оператор ?? 698
Обнуляемые объекты, операторы отношения и логические операторы 699
Частичные типы 700
Частичные методы 701
Создание объектов динамического типа 703
Возможность взаимодействия с моделью COM 707
Дружественные сборки 708
Разные ключевые слова 708
Ключевое слов lock 708
Ключевое слово readonly 709
Ключевые слова const и volatile 710
Оператор using 711
Ключевое слово extern 712

ЧАСТЬ II. БИБЛИОТЕКА C# 717

Глава 21. Пространство имен System 719
Члены пространства имен System 720
Класс Math 721
Структуры .NET, соответствующие встроенным типам значений 727
Структуры целочисленных типов данных 728
Структуры типов данных с плавающей точкой 730
Структура Decimal 735
Структура Char 741
Структура Boolean 748
Класс Array 750
Сортировка и поиск в массивах 763

Обращение содержимого массива 766
Копирование массива 767
Применение предиката 768
Применение делегата Action 769
Класс BitConverter 771
Генерирование случайных чисел средствами класса Random 773
Управление памятью и класс GC 774
Класс object 776
Класс Tuple 777
Интерфейсы IComparable и IComparable<T> 778
Интерфейс IEquatable<T> 778
Интерфейс IConvertible 779
Интерфейс ICloneable 779
Интерфейсы IFormatProvider и IFormattable 781
Интерфейсы IObservable<T> и IObserver<T> 781

Глава 22. Строки и форматирование 783

Строки в C# 783
Класс String 784
Конструкторы класса String 784
Поле, индексатор и свойство класса String 785
Операторы класса String 786
Заполнение и обрезка строк 808
Вставка, удаление и замена строк 810
Смена регистра 811
Применение метода Substring() 811
Методы расширения класса String 812
Форматирование 812
Общее представление о форматировании 812
Спецификаторы формата числовых данных 814
Представление о номерах аргументов 815
Применение методов String.Format() и ToString() для форматирования данных 816
Применение метода String.Format() для форматирования значений 816
Применение метода ToString() для форматирования данных 819
Определение пользовательского формата числовых данных 820
Символы-заполнители специального формата числовых данных 820
Форматирование даты и времени 824
Определение пользовательского формата даты и времени 827
Форматирование промежутков времени 829
Форматирование перечислений 830

Глава 23. Многопоточное программирование. Часть первая: основы 833

Основы многопоточной обработки 834

Класс Thread 835
Создание и запуск потока 836
Простые способы усовершенствования многопоточной программы 838
Создание нескольких потоков 839
Определение момента окончания потока 841
Передача аргумента потоку 844
Свойство IsBackground 846
Приоритеты потоков 847
Синхронизация 849
Другой подход к синхронизации потоков 853
Класс Monitor и блокировка 855
Сообщение между потоками с помощью методов Wait(), Pulse() и PulseAll() 855
Пример использования методов Wait() и Pulse() 856
Взаимоблокировка и состояние гонки 860
Применение атрибута MethodImplAttribute 860
Применение мьютекса и семафора 862
Мьютекс 863
Семафор 867
Применение событий 870
Класс Interlocked 873
Классы синхронизации, внедренные в версии .NET Framework 4.0 874
Прерывание потока 875
Другая форма метода Abort() 876
Отмена действия метода Abort() 878
Приостановка и возобновление потока 880
Определение состояния потока 880
Применение основного потока 880
Дополнительные средства многопоточной обработки, внедренные в версии .NET Framework 4.0 882
Рекомендации по многопоточному программированию 882
Запуск отдельной задачи 882

Глава 24. Многопоточное программирование. Часть вторая: библиотека TPL 885

Два подхода к параллельному программированию 886
Класс Task 887
Создание задачи 887
Применение идентификатора задачи 890
Применение методов ожидания 891
Вызов метода Dispose() 895
Применение класса TaskFactory для запуска задачи 895
Применение лямбда-выражения в качестве задачи 896
Создание продолжения задачи 897

Возврат значения из задачи 899
Отмена задачи и обработка исключения AggregateException 901
Другие средства организации задач 905
Класс Parallel 906
Распараллеливание задач методом Invoke() 906
Применение метода For() 909
Применение метода ForEach() 915
Исследование возможностей PLINQ 917
Класс ParallelEnumerable 917
Распараллеливание запроса методом AsParallel() 918
Применение метода AsOrdered() 919
Отмена параллельного запроса 920
Другие средства PLINQ 922
Вопросы эффективности PLINQ 922
Глава 25. Коллекции, перечислители и итераторы 923
Краткий обзор коллекций 924
Необобщенные коллекции 925
Интерфейсы необобщенных коллекций 926
Структура DictionaryEntry 931
Классы необобщенных коллекций 931
Хранение отдельных битов в классе коллекции BitArray 950
Специальные коллекции 953
Обобщенные коллекции 954
Интерфейсы обобщенных коллекций 954
Структура KeyValuePair<TKey, TValue> 960
Классы обобщенных коллекций 960
Параллельные коллекции 983
Сохранение объектов, определяемых пользователем классов, в коллекции 988
Реализация интерфейса IComparable 990
Реализация интерфейса IComparable для необобщенных коллекций 990
Реализация интерфейса IComparable для обобщенных коллекций 992
Применение интерфейса IComparer 994
Применение необобщенного интерфейса IComparer 994
Применение обобщенного интерфейса IComparer<T> 996
Применение класса StringComparer 997
Доступ к коллекции с помощью перечислителя 998
Применение обычного перечислителя 999
Применение перечислителя типа IDictionaryEnumerator 1000
Реализация интерфейсов IEnumerable и IEnumerator 1001
Применение итераторов 1003
Прерывание итератора 1005
Применение нескольких операторов yield 1006

Создание именованного итератора 1006
Создание обобщенного итератора 1008
Инициализаторы коллекций 1009

Глава 26. Сетевые средства подключения к Интернету 1011
Члены пространства имен System.Net 1012
Универсальные идентификаторы ресурсов 1013
Основы организации доступа к Интернету 1014
Класс WebRequest 1015
Класс WebResponse 1017
Классы HttpWebRequest и HttpWebResponse 1018
Первый простой пример 1018
Обработка сетевых ошибок 1021
Исключения, генерируемые методом Create() 1021
Исключения, генерируемые методом GetResponse() 1022
Исключения, генерируемые методом GetResponseStream() 1022
Обработка исключений 1022
Класс Uri 1024
Доступ к дополнительной информации, получаемой в ответ по протоколу HTTP 1025
Доступ к заголовку 1026
Доступ к cookie-наборам 1027
Применение свойства LastModified 1029
Практический пример создания программы MiniCrawler 1030
Применение класса WebClient 1034

Приложение. Краткий справочник по составлению документирующих комментариев 1039
Дескрипторы XML-комментариев 1039
Компилирование документирующих комментариев 1041
Пример составления документации в формате XML 1041

Предметный указатель 1044

Об авторе

Герберт Шилдт (Herbert Schildt) является одним из самых известных специалистов по языкам программирования C#, C++, C и Java. Его книги по программированию изданы миллионными тиражами и переведены с английского на все основные иностранные языки. Его перу принадлежит целый ряд популярных книг, в том числе *Полный справочник по Java*, *Полный справочник по C++*, *Полный справочник по C* (все перечисленные книги вышли в издательстве "Вильямс" в 2007 и 2008 гг.). Несмотря на то что Герберт Шилдт интересуется всеми аспектами вычислительной техники, его основная специализация — языки программирования, в том числе компиляторы, интерпретаторы и языки программирования роботов. Он также проявляет живой интерес к стандартизации языков. Шилдт окончил Иллинойский университет и имеет степени магистра и бакалавра. Связаться с ним можно, посетив его веб-сайт по адресу `www.HerbSchildt.com`.

О научном редакторе

Майкл Ховард (Michael Howard) работает руководителем проекта программной защиты в группе техники информационной безопасности, входящей в подразделение разработки защищенных информационных систем (TwC) корпорации Microsoft, где он отвечает за внедрение надежных с точки зрения безопасности методов проектирования, программирования и тестирования информационных систем в масштабах всей корпорации. Ховард является автором методики безопасной разработки (Security Development Lifecycle — SDL) — процесса повышения безопасности программного обеспечения, выпускаемого корпорацией Microsoft.

Свою карьеру в корпорации Microsoft Ховард начал в 1992 году, проработав два первых года с ОС Windows и компиляторами в службе поддержки программных продуктов (Product Support Services) новозеландского отделения корпорации, а затем перейдя в консультационную службу (Microsoft Consulting Services), где он занимался клиентской поддержкой инфраструктуры безопасности и помогал в разработке заказных проектных решений и программного обеспечения. В 1997 году Ховард переехал в Соединенные Штаты и поступил на работу в отделение Windows веб-службы Internet Information Services, представлявшей собой веб-сервер следующего поколения в корпорации Microsoft, прежде чем перейти в 2000 году к своим текущим служебным обязанностям.

Ховард является редактором журнала *IEEE Security & Privacy*, часто выступает на конференциях, посвященных безопасности программных средств, и регулярно пишет статьи по вопросам безопасного программирования и проектирования программного обеспечения. Он является одним из авторов шести книг по безопасности информационных систем.

Благодарности

Особая благодарность выражается Майклу Ховарду за превосходное научное редактирование книги. Его знания, опыт, дельные советы и предложения оказались неоценимыми.

Предисловие

Программисты — люди требовательные, постоянно ищущие пути повышения производительности, эффективности и переносимости разрабатываемых ими программ. Они не менее требовательны к применяемым инструментальным средствам и особенно к языкам программирования. Существует немало языков программирования, но лишь немногие из них действительно хороши. Хороший язык программирования должен быть одновременно эффективным и гибким, а его синтаксис — кратким, но ясным. Он должен облегчать создание правильного кода, не мешая делать это, а также поддерживать самые современные возможности программирования, но не ультрамодные тенденции, заводящие в тупик. И наконец, хороший язык программирования должен обладать еще одним, едва уловимым качеством: вызывать у нас такое ощущение, будто мы находимся в своей стихии, когда пользуемся им. Именно таким языком и является C#.

Язык C# был создан корпорацией Microsoft для поддержки среды .NET Framework и опирается на богатое наследие в области программирования. Его главным разработчиком был Андерс Хейльсберг (Anders Hejlsberg) — известнейший специалист по программированию. C# происходит напрямую от двух самых удачных в области программирования языков: C и C++. От языка C он унаследовал синтаксис, многие ключевые слова и операторы, а от C++ — усовершенствованную объектную модель. Кроме того, C# тесно связан с Java — другим не менее удачным языком.

Имея общее происхождение, но во многом отличаясь, C# и Java похожи друг на друга как близкие, но не кровные родственники. В обоих языках поддерживается

распределенное программирование и применяется промежуточный код для обеспечения безопасности и переносимости, но отличия кроются в деталях реализации. Кроме того, в обоих языках предоставляется немало возможностей для проверки ошибок при выполнении, обеспечения безопасности и управляемого исполнения, хотя и в этом случае отличия кроются в деталях реализации. Но в отличие от Java, язык C# предоставляет доступ к указателям — средствам программирования, которые поддерживаются в C++. Следовательно, C# сочетает в себе эффективность, присущую C++, и типовую безопасность, характерную для Java. Более того, компромиссы между эффективностью и безопасностью в этом языке программирования тщательно уравновешены и совершенно прозрачны.

На протяжении всей истории вычислительной техники языки программирования развивались, приспосабливаясь к изменениям в вычислительной среде, новшествам в теории языков программирования и новым тенденциям в осмыслении и подходе к работе программистов. И в этом отношении C# не является исключением. В ходе непрерывного процесса уточнения, адаптации и нововведений C# продемонстрировал способность быстро реагировать на потребности программистов в переменах. Об этом явно свидетельствуют многие новые возможности, введенные в C# с момента выхода исходной версии 1.0 этого языка в 2000 году.

Рассмотрим для примера первое существенное исправление, внесенное в версии C# 2.0, где был введен ряд свойств, упрощавших написание более гибкого, надежного и быстро действующего кода. Без сомнения, самым важным новшеством в версии C# 2.0 явилось внедрение обобщений. Благодаря обобщениям стало возможным создание типизированного, повторно используемого кода на C#. Следовательно, внедрение обобщений позволило основательно расширить возможности и повысить эффективность этого языка.

А теперь рассмотрим второе существенное исправление, внесенное в версии C# 3.0. Не будет преувеличением сказать, что в этой версии введены свойства, переопределившие саму суть C# и поднявшие на новый уровень разработку языков программирования. Среди многих новых свойств особенно выделяются два следующих: LINQ и лябмда-выражения. Сокращение LINQ означает *язык интегрированных запросов*. Это языковое средство позволяет создавать запросы к базе данных, используя элементы C#. А лябмда-выражения — это синтаксис функционалов с помощью лямбда-оператора =>, причем лябмда-выражения часто применяются в LINQ-выражениях.

И наконец, третье существенное исправление было внесено в версии C# 4.0, описываемой в этой книге. Эта версия опирается на предыдущие и в то же время предоставляет целый ряд новых средств для рационального решения типичных задач программирования. В частности, в ней внедрены именованные и необязательные аргументы, что делает более удобным вызов некоторых видов методов; добавлено ключевое слово `dynamic`, упрощающее применение C# в тех случаях, когда тип данных создается во время выполнения, например, при сопряжении с моделью компонентных объектов (COM) или при использовании рефлексии; а средства ковариантности и контравариантности, уже поддерживавшиеся в C#, были расширены с тем, чтобы использовать параметры типа. Благодаря усовершенствованиям среды .NET Framework, представленной в виде библиотеки C#, в данной версии поддерживается параллельное программирование средствами TPL (Task Parallel Library — Библиотека распараллеливания задач) и PLINQ (Parallel LINQ — Параллельный язык интегрированных запросов). Эти подсистемы упрощают создание кода, который мас-

штабируется автоматически для более эффективного использования компьютеров с многоядерными процессорами. Таким образом, с выпуском версии C# 4.0 появилась возможность воспользоваться преимуществами высокопроизводительных вычислительных платформ.

Благодаря своей способности быстро приспосабливаться к постоянно меняющимся потребностям в области программирования C# по-прежнему остается живым и новаторским языком. А следовательно, он представляет собой один из самых эффективных и богатых своими возможностями языков в современном программировании. Это язык, пренебречь которым не может позволить себе ни один программист. И эта книга призвана помочь вам овладеть им.

Структура книги

В этой книге описывается версия 4.0 языка C#. Она разделена на две части. В части I дается подробное пояснение языка C#, в том числе новых средств, внедренных в версии 4.0. Это самая большая часть книги, в которой описываются ключевые слова, синтаксис и средства данного языка, а также операции ввода-вывода и обработки файлов, рефлексия и препроцессор.

В части II рассматривается библиотека классов C#, которая одновременно является библиотекой классов для среды .NET Framework. Эта библиотека довольно обширна, но за недостатком места в этой книге просто невозможно описать ее полностью. Поэтому в части II основное внимание уделяется корневой библиотеке, которая находится в пространстве имен `System`. Кроме того, в этой части рассматриваются коллекции, организация многопоточной обработки, сетевого подключения к Интернету, а также средства TPL и PLINQ. Это те части более обширной библиотеки классов, которыми пользуется всякий, программирующий на языке C#.

Книга для всех программирующих

Для чтения этой книги вообще не требуется иметь опыт программирования. Если вы уже знаете C++ или Java, то сможете довольно быстро продвинуться в освоении излагаемого в книге материала, поскольку у C# имеется немало общего с этими языками. Даже если вам не приходилось программировать прежде, вы сможете освоить C#, но для этого вам придется тщательно проработать примеры, приведенные в каждой главе книги.

Необходимое программное обеспечение

Для компилирования и выполнения примеров программ на C# 4.0, приведенных в этой книге, вам потребуется пакет Visual Studio 2010 (или более поздняя версия).

Код, доступный в Интернете

Не забывайте о том, что исходный код для примеров всех программ, приведенных в этой книге, свободно доступен для загрузки по адресу `www.mhprofessional.com`.

Что еще почитать

Эта книга — своеобразный "ключ" к целой серии книг по программированию, написанных Гербертом Шилдтом. Ниже перечислены другие книги, которые могут представлять для вас интерес.

Для изучения языка программирования Java рекомендуются следующие книги.

Полный справочник по Java (ИД " Вильямс", 2007 г.)

Java: руководство для начинающих (ИД " Вильямс", 2008 г.)

SWING: руководство для начинающих (ИД "Вильямс", 2007 г.)

Искусство программирования на Java (ИД "Вильямс", 2005 г.)

Java. Методики программирования Шилдта (ИД "Вильямс", 2008 г.)

Для изучения языка программирования C++ особенно полезными окажутся следующие книги.

Полный справочник по C++ (ИД "Вильямс", 2007 г.)

C++. Руководство для начинающих (ИД "Вильямс", 2005 г.)

STL Programming From the Ground Up

Искусство программирования на C++

C++. Методики программирования Шилдта (ИД "Вильямс", 2009 г.)

Если же вы стремитесь овладеть языком C, составляющим основу всех современных языков программирования, вам будет интересно прочитать книгу

Полный справочник по C (ИД "Вильямс", 2007 г.)

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

ЧАСТЬ

Язык C#

В части I рассматриваются отдельные элементы языка C#, в том числе ключевые слова, синтаксис и операторы. Описывается также ряд основополагающих методов программирования, тесно связанных с языком C#, включая организацию ввода-вывода и рефлексию.

ГЛАВА 1 Создание C#
ГЛАВА 2 Краткий обзор элементов C#
ГЛАВА 3 Типы данных, литералы и переменные
ГЛАВА 4 Операторы
ГЛАВА 5 Управляющие операторы
ГЛАВА 6 Введение в классы, объекты и методы
ГЛАВА 7 Массивы и строки
ГЛАВА 8 Подробнее о методах и классах
ГЛАВА 9 Перегрузка операторов
ГЛАВА 10 Индексаторы и свойства
ГЛАВА 11 Наследование
ГЛАВА 12 Интерфейсы, структуры и перечисления
ГЛАВА 13 Обработка исключительных ситуаций
ГЛАВА 14 Применение средств ввода-вывода
ГЛАВА 15 Делегаты, события и лямбда-выражения
ГЛАВА 16 Пространства имен, препроцессор и сборки
ГЛАВА 17 Динамическая идентификация типов, рефлексия и атрибуты
ГЛАВА 18 Обобщения
ГЛАВА 19 LINQ
ГЛАВА 20 Небезопасный код, указатели, обнуляемые типы и разные ключевые слова

1 ГЛАВА

Создание C#

C# является основным языком разработки программ на платформе .NET корпорации Microsoft. В нем удачно сочетаются испытанные средства программирования с самыми последними новшествами и предоставляется возможность для эффективного и очень практичного написания программ, предназначенных для вычислительной среды современных предприятий. Это, без сомнения, один из самых важных языков программирования XXI века.

Назначение этой главы — представить C# в его историческом контексте, упомянув и те движущие силы, которые способствовали его созданию, выработке его конструктивных особенностей и определили его влияние на другие языки программирования. Кроме того, в этой главе поясняется взаимосвязь C# со средой .NET Framework. Как станет ясно из дальнейшего материала, C# и .NET Framework совместно образуют весьма изящную среду программирования.

Генеалогическое дерево C#

Языки программирования не существуют в пустоте. Напротив, они тесно связаны друг с другом таким образом, что на каждый новый язык оказывают в той или иной форме влияние его предшественники. Этот процесс сродни перекрестному опылению, в ходе которого свойства одного языка приспосабливаются к другому языку, полезные нововведения внедряются в существующий контекст, а устаревшие конструкции удаляются. Таким путем развиваются языки программирования и совершенствуется искусство программирования. И в этом отношении C# не является исключением.

У языка программирования C# "богатое наследство". Он является прямым наследником двух самых удачных языков программирования: C и C++. Он также имеет тесные родственные связи с еще одним языком: Java. Ясное представление об этих взаимосвязях имеет решающее значение для понимания C#. Поэтому сначала определим, какое место занимает C# среди этих трех языков.

Язык C — начало современной эпохи программирования

Создание C знаменует собой начало современной эпохи программирования. Язык C был разработан Деннисом Ритчи (Dennis Ritchie) в 1970-е годы для программирования на мини-ЭВМ DEC PDP-11 под управлением операционной системы Unix. Несмотря на то что в ряде предшествовавших языков, в особенности Pascal, был достигнут значительный прогресс, именно C установил тот образец, которому до сих пор следуют в программировании.

Язык C появился в результате революции в *структурном программировании* в 1960-е годы. До появления структурного программирования писать большие программы было трудно, поскольку логика программы постепенно вырождалась в так называемый "макаронный" код — запутанный клубок безусловных переходов, вызовов и возвратов, которые трудно отследить. В структурированных языках программирования этот недостаток устранялся путем ввода строго определенных управляющих операторов, подпрограмм с локальными переменными и других усовершенствований. Благодаря применению методов структурного программирования сами программы стали более организованными, надежными и управляемыми.

И хотя в то время существовали и другие структурированные языки программирования, именно в C впервые удалось добиться удачного сочетания эффективности, изящества и выразительности. Благодаря своему краткому, но простому синтаксису в сочетании с принципом, ставившим во главу угла программиста, а не сам язык, C быстро завоевал многих сторонников. Сейчас уже нелегко представить себе, что C оказался своего рода "струей свежего воздуха", которого так не хватало программистам. В итоге C стал самым распространенным языком структурного программирования в 1980-е годы.

Но даже у такого достойного языка, как C, имелись свои ограничения. К числу самых труднопреодолимых его ограничений относится неспособность справиться с большими программами. Как только проект достигает определенного масштаба, язык C тут же ставит предел, затрудняющий понимание и сопровождение программ при их последующем разрастании. Конкретный предел зависит от самой программы, программиста и применяемых инструментальных средств, тем не менее, всегда существует "порог", за которым программа на C становится неуправляемой.

Появление ООП и C++

К концу 1970-х годов масштабы многих проектов приблизились к пределам, с которыми уже не могли справиться методики структурного программирования вообще и язык C в частности. Для решения этой проблемы было открыто новое направление в программировании — так называемое объектно-ориентированное программирование (ООП). Применяя метод ООП, программист мог работать с более "крупными" программами. Но главная трудность заключалась в том, что C, самый распространенный в то время язык, не поддерживал ООП. Стремление к созданию объектно-ориентированного варианта C в конечном итоге привело к появлению C++.

Язык C++ был разработан в 1979 году Бьярне Страуструпом (Bjarne Stroustrup), работавшим в компании Bell Laboratories, базировавшейся в Мюррей-Хилл, шт. Нью-Джерси. Первоначально новый язык назывался "C с классами", но в 1983 году он был переименован в C++. Язык C полностью входит в состав C++, а следовательно, C служит основанием, на котором зиждется C++. Большая часть дополнений, введенных Страуструпом, обеспечивала плавный переход к ООП. И вместо того чтобы изучать совершенно новый язык, программирующему на C требовалось лишь освоить ряд новых свойств, чтобы воспользоваться преимуществами методики ООП.

В течение 1980-х годов C++ все еще оставался в тени, интенсивно развиваясь, но к началу 1990-х годов, когда он уже был готов для широкого применения, его популярность в области программирования заметно возросла. К концу 1990-х годов он стал наиболее широко распространенным языком программирования и в настоящее время по-прежнему обладает неоспоримыми преимуществами языка разработки высокопроизводительных программ системного уровня.

Важно понимать, что разработка C++ не была попыткой создать совершенно новый язык программирования. Напротив, это была попытка усовершенствовать уже существовавший довольно удачный язык. Такой подход к разработке языков программирования, основанный на уже существующем языке и совершенствующий его далее, превратился в упрочившуюся тенденцию, которая продолжается до сих пор.

Появление Интернета и Java

Следующим важным шагом в развитии языков программирования стала разработка Java. Работа над языком Java, который первоначально назывался Oak (Дуб), началась в 1991 году в компании Sun Microsystems. Главной "движущей силой" в разработке Java был Джеймс Гослинг (James Gosling), но не малая роль в работе над этим языком принадлежит также Патрику Ноутону (Patrick Naughton), Крису Уорту (Chris Warth), Эду Фрэнку (Ed Frank) и Майку Шеридану (Mike Sheridan).

Java представляет собой структурированный, объектно-ориентированный язык с синтаксисом и конструктивными особенностями, унаследованными от C++. Нововведения в Java возникли не столько в результате прогресса в искусстве программирования, хотя некоторые успехи в данной области все же были, сколько вследствие перемен в вычислительной среде. До появления на широкой арене Интернета большинство программ писалось, компилировалось и предназначалось для конкретного процессора и операционной системы. Как известно, программисты всегда стремились повторно использовать свой код, но, несмотря на это, легкой переносимости программ из одной среды в другую уделялось меньше внимания, чем более насущным задачам. Тем не менее с появлением Интернета, когда в глобальную сеть связывались

разнотипные процессоры и операционные системы, застаревшая проблема переносимости программ вновь возникла с неожиданной остротой. Для решения проблемы переносимости потребовался новый язык, и им стал Java.

Самым важным свойством (и причиной быстрого признания) Java является способность создавать межплатформенный, переносимый код, тем не менее, интересно отметить, что первоначальным толчком для разработки Java послужил не Интернет, а потребность в независящем от платформы языке, на котором можно было бы разрабатывать программы для встраиваемых контроллеров. В 1993 году стало очевидно, что вопросы межплатформенной переносимости, возникавшие при создании кода для встраиваемых контроллеров, стали актуальными и при попытке написать код для Интернета. Напомним, что Интернет — это глобальная распределенная вычислительная среда, в которой работают и мирно "сосуществуют" разнотипные компьютеры. И в итоге оказалось, что теми же самыми методами, которыми решалась проблема переносимости программ в мелких масштабах, можно решать аналогичную задачу в намного более крупных масштабах Интернета.

Переносимость программ на Java достигалась благодаря преобразованию исходного кода в промежуточный, называемый *байт-кодом*. Этот байт-код затем выполнялся виртуальной машиной Java (JVM) — основной частью исполняющей системы Java. Таким образом, программа на Java могла выполняться в любой среде, для которой была доступна JVM. А поскольку JVM реализуется относительно просто, то она сразу же стала доступной для большого числа сред.

Применением байт-кода Java коренным образом отличается от C и C++, где исходный код практически всегда компилируется в исполняемый машинный код, который, в свою очередь, привязан к конкретному процессору и операционной системе. Так, если требуется выполнить программу на C или C++ в другой системе, ее придется перекомпилировать в машинный код специально для данной вычислительной среды. Следовательно, для создания программы на C или C++, которая могла был выполняться в различных средах, потребовалось бы несколько разных исполняемых версий этой программы. Это оказалось бы не только непрактично, но и дорого. Изящным и рентабельным решением данной проблемы явилось применение в Java промежуточного кода. Именно это решение было в дальнейшем приспособлено для целей языка C#.

Как упоминалось ранее, Java происходит от C и C++. В основу этого языка положен синтаксис C, а его объектная модель получила свое развитие из C++. И хотя код Java не совместим с кодом C или C++ ни сверху вниз, ни снизу вверх, его синтаксис очень похож на эти языки, что позволяет большому числу программирующих на C или C++ без особого труда перейти на Java. Кроме того, Java построен по уже существующему образцу, что позволило разработчикам этого языка сосредоточить основное внимание на новых и передовых его свойствах. Как и Страуструпу при создании C++, Гослингу и его коллегам не пришлось изобретать велосипед, т.е. разрабатывать Java как совершенно новый язык. Более того, после создания Java языки C и C++ стали признанной основой, на которой можно разрабатывать новые языки программирования.

Создание C#

Несмотря на то что в Java успешно решаются многие вопросы переносимости программ в среде Интернета, его возможности все же ограничены. Ему, в частности, недостает *межъязыковой возможности взаимодействия*, называемой также *многоязыковым программированием*. Это возможность кода, написанного на одном языке, без труда вза-

имодействовать с кодом, написанным на другом языке. Межъязыковая возможность взаимодействия требуется для построения крупных, распределенных программных систем. Она желательна также для создания отдельных компонентов программ, поскольку наиболее ценным компонентом считается тот, который может быть использован в самых разных языках программирования и в самом большом числе операционных сред.

Другой возможностью, отсутствующей в Java, является полная интеграция с платформой Windows. Несмотря на то что программы на Java могут выполняться в среде Windows, при условии, что установлена виртуальная машина Java, среды Java и Windows не являются сильно связанными. А поскольку Windows является самой распространенной операционной системой во всем мире, то отсутствие прямой поддержки Windows является существенным недостатком Java.

Для удовлетворения этих и других потребностей программирования корпорация Microsoft разработала в конце 1990-х годов язык C# как часть общей стратегии .NET. Впервые он был выпущен в виде альфа-версии в середине 2000 года. Главным разработчиком C# был Андерс Хейльсберг — один из ведущих в мире специалистов по языкам программирования, который может похвалиться рядом заметных достижений в данной области. Достаточно сказать, что в 1980-е годы он был автором очень удачной и имевшей большое значение разработки — языка Turbo Pascal, изящная реализация которого послужила образцом для создания всех последующих компиляторов.

Язык C# непосредственно связан с C, C++ и Java. И это не случайно. Ведь это три самых широко распространенных и признанных во всем мире языка программирования. Кроме того, на момент создания C# практически все профессиональные программисты уже владели C, C++ или Java. Благодаря тому что C# построен на столь прочном и понятном основании, перейти на этот язык из C, C++ или Java не представляло особого труда. А поскольку и Хейльсбергу не нужно (да и нежелательно) было изобретать велосипед, то он мог сосредоточиться непосредственно на усовершенствованиях и нововведениях в C#.

На рис. 1.1 приведено генеалогическое дерево C#. Предком C# во втором поколении является C, от которого он унаследовал синтаксис, многие ключевые слова и операторы. Кроме того, C# построен на усовершенствованной объектной модели, определенной в C++. Если вы знаете C или C++, то будете чувствовать себя уютно и с языком C#.

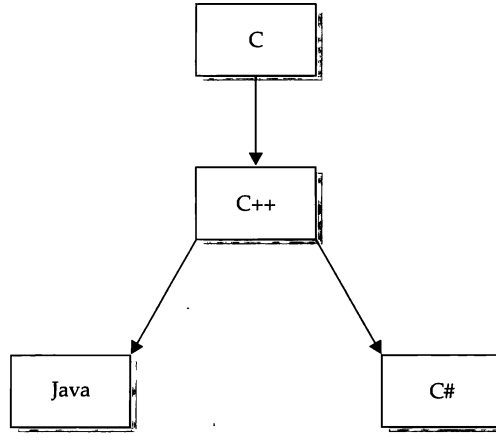


Рис. 1.1. Генеалогическое дерево C#

Родственные связи C# и Java более сложные. Как пояснялось выше, Java также происходит от C и C++ и обладает общим с ними синтаксисом и объектной моделью. Как и Java, C# предназначен для получения переносимого кода, но C# не происходит непосредственно от Java. Напротив, C# и Java — это близкие, но не кровные родственники, имеющие общих предков, но во многом отличающиеся друг от друга. Впрочем, если вы знаете Java, то многие понятия C# окажутся вам знакомыми. С другой стороны, если вам в будущем придется изучать Java, то многие понятия, усвоенные в C#, могут быть легко распространены и на Java.

В C# имеется немало новых средств, которые будут подробно рассмотрены на страницах этой книги, но самое важное из них связано со встроенной поддержкой программных компонентов. В действительности C# может считаться компонентно-ориентированным языком программирования, поскольку в него внедрена встроенная поддержка написания программных компонентов. Например, в состав C# входят средства прямой поддержки таких составных частей программных компонентов, как свойства, методы и события. Но самой важной компонентно-ориентированной особенностью этого языка, вероятно, является возможность работы в безопасной среде многоязыкового программирования.

Развитие C#

С момента выпуска исходной версии 1.0 развитие C# происходило быстро. Вскоре после версии 1.0 корпорация Microsoft выпустила версию 1.1, в которую было внесено немало корректив, но мало значительных возможностей. Однако ситуация совершенно изменилась после выпуска версии C# 2.0.

Появление версии 2.0 стало поворотным моментом в истории развития C#, поскольку в нее было введено много новых средств, в том числе обобщения, частичные типы и анонимные методы, которые основательно расширили пределы возможностей и область применения этого языка, а также повысили его эффективность. После выпуска версии 2.0 "упрочилось" положение C#. Ее появление продемонстрировало также приверженность корпорации Microsoft к поддержке этого языка в долгосрочной перспективе.

Следующей значительной вехой в истории развития C# стал выпуск версии 3.0. В связи с внедрением многих новых свойств в версии C# 2.0 можно было ожидать некоторого замедления в развитии C#, поскольку программистам требовалось время для их освоения, но этого не произошло. С появлением версии 3.0 корпорация Microsoft внедрила ряд новшеств, совершенно изменивших общее представление о программировании. К числу этих новшеств относятся, среди прочего, лямбда-выражения, язык интегрированных запросов (LINQ), методы расширения и неявно типизированные переменные. Конечно, все эти новые возможности очень важны, поскольку они оказали заметное влияние на развитие данного языка, но среди них особенно выделяются две: язык интегрированных запросов (LINQ) и лямбда-выражения. Язык LINQ и лямбда-выражения вносят совершенно новый акцент в программирование на C# и еще глубже подчеркивают его ведущую роль в непрекращающейся эволюции языков программирования.

Текущей является версия C# 4.0, о которой и пойдет речь в этой книге. Эта версия прочно опирается на три предыдущие основные версии C#, дополняя их целым рядом новых средств. Вероятно, самыми важными среди них являются именованные и необязательные аргументы. В частности, именованные аргументы позволяют связывать

аргумент с параметром по имени. А необязательные аргументы дают возможность указывать для параметра используемый по умолчанию аргумент. Еще одним важным новым средством является тип `dynamic`, применяемый для объявления объектов, которые проверяются на соответствие типов во время выполнения, а не компиляции. Кроме того, ковариантность и контравариантность параметров типа поддерживается благодаря новому применению ключевых слов `in` и `out`. Тем, кто пользуется моделью COM вообще и прикладными интерфейсами Office Automation API в частности, существенно упрощен доступ к этим средствам, хотя они и не рассматриваются в этой книге. В целом, новые средства, внедренные в версии C# 4.0, способствуют дальнейшей рационализации программирования и повышают практичность самого языка C#.

Еще два важных средства, внедренных в версии 4.0 и непосредственно связанных с программированием на C#, предоставляются не самим языком, а средой .NET Framework 4.0. Речь идет о поддержке параллельного программирования с помощью библиотеки распараллеливания задач (TPL) и параллельном варианте языка интегрированных запросов (PLINQ). Оба эти средства позволяют существенно усовершенствовать и упростить процесс создания программ, в которых применяется принцип параллелизма. И то и другое средство упрощает создание многопоточного кода, который масштабируется автоматически для использования нескольких процессоров, доступных на компьютере. В настоящее время широкое распространение получили компьютеры с многоядерными процессорами, и поэтому возможность распараллеливать выполнение кода среди всех доступных процессоров приобретает все большее значение практически для всех, кто программирует на C#. В силу этого особого обстоятельства средства TPL и PLINQ рассматриваются в данной книге.

Связь C# со средой .NET Framework

Несмотря на то что C# является самодостаточным языком программирования, у него имеется особая взаимосвязь со средой выполнения .NET Framework. Наличие такой взаимосвязи объясняется двумя причинами. Во-первых, C# первоначально предназначался для создания кода, который должен выполняться в среде .NET Framework. И во-вторых, используемые в C# библиотеки определены в среде .NET Framework. На практике это означает, что C# и .NET Framework тесно связаны друг с другом, хотя теоретически C# можно отделить от среды .NET Framework. В связи с этим очень важно иметь хотя бы самое общее представление о среде .NET Framework и ее значении для C#.

О среде NET Framework

Назначение .NET Framework — служить средой для поддержки разработки и выполнения сильно распределенных компонентных приложений. Она обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows. Что же касается взаимосвязи с C#, то среда .NET Framework определяет два очень важных элемента. Первым из них является *общеязыковая среда выполнения* (Common Language Runtime — CLR). Это система, управляющая выполнением программ. Среди прочих преимуществ — CLR как составная часть среды .NET Framework поддерживает многоязыковое программирование, а также обеспечивает переносимость и безопасное выполнение программ.

Вторым элементом среды .NET Framework является *библиотека классов*. Эта библиотека предоставляет программе доступ к среде выполнения. Так, если требуется выполнить операцию ввода-вывода, например вывести что-нибудь на экран, то для этой цели используется библиотека классов .NET. Для тех, кто только начинает изучать программирование, понятие *класса* может оказаться незнакомым. Оно подробно разъясняется далее в этой книге, а пока достаточно сказать, что класс — это объектно-ориентированная конструкция, помогающая организовать программы. Если программа ограничивается средствами, определяемыми в библиотеке классов .NET, то такая программа может выполняться везде, где поддерживается среда выполнения .NET. А поскольку в C# библиотека классов .NET используется автоматически, то программы на C# заведомо оказываются переносимыми во все имеющиеся среды .NET Framework.

Принцип действия CLR

Среда CLR управляет выполнением кода .NET. Действует она по следующему принципу. Результатом компиляции программы на C# является не исполняемый код, а файл, содержащий особого рода псевдокод, называемый *Microsoft Intermediate Language*, MSIL (промежуточный язык Microsoft). Псевдокод MSIL определяет набор переносимых инструкций, не зависящих от конкретного процессора. По существу, MSIL определяет переносимый язык ассемблера. Следует, однако, иметь в виду, что, несмотря на кажущееся сходство псевдокода MSIL с байт-кодом Java, это все же разные понятия.

Назначение CLR — преобразовать промежуточный код в исполняемый код по ходу выполнения программы. Следовательно, всякая программа, скомпилированная в псевдокод MSIL, может быть выполнена в любой среде, где имеется реализация CLR. Именно таким образом отчасти достигается переносимость в среде .NET Framework.

Псевдокод MSIL преобразуется в исполняемый код с помощью *JIT-компилятора*. Сокращение JIT означает *точно в срок* и отражает оперативный характер данного компилятора. Процесс преобразования кода происходит следующим образом. При выполнении программы среда CLR активизирует JIT-компилятор, который преобразует псевдокод MSIL в собственный код системы по требованию для каждой части программы. Таким образом, программа на C# фактически выполняется как собственный код, несмотря на то, что первоначально она скомпилирована в псевдокод MSIL. Это означает, что такая программа выполняется так же быстро, как и в том случае, когда она исходно скомпилирована в собственный код, но в то же время она приобретает все преимущества переносимости псевдокода MSIL.

Помимо псевдокода MSIL, при компилировании программы на C# получаются также *метаданные*, которые служат для описания данных, используемых в программе, а также обеспечивают простое взаимодействие одного кода с другим. Метаданные содержатся в том же файле, что и псевдокод MSIL.

Управляемый и неуправляемый код

Как правило, при написании программы на C# формируется так называемый *управляемый код*. Как пояснялось выше, такой код выполняется под управлением среды CLR, и поэтому на него накладываются определенные ограничения, хотя это и дает ряд преимуществ. Ограничения накладываются и удовлетворятся довольно просто: компи-

лятор должен сформировать файл MSIL, предназначенный для выполнения в среде CLR, используя при этом библиотеку классов .NET, — и то и другое обеспечивается средствами C#. Ко многим преимуществам управляемого кода относятся, в частности, современные способы управления памятью, возможность программирования на разных языках, повышение безопасности, поддержка управления версиями и четкая организация взаимодействия программных компонентов.

В отличие от управляемого кода, неуправляемый код не выполняется в среде CLR. Следовательно, до появления среды .NET Framework во всех программах для Windows применялся неуправляемый код. Впрочем, управляемый и неуправляемый коды могут взаимодействовать друг с другом, а значит, формирование управляемого кода в C# совсем не означает, что на его возможность взаимодействия с уже существующими программами накладываются какие-то ограничения.

Общеязыковая спецификация

Несмотря на все преимущества, которые среда CLR дает управляемому коду, для максимального удобства его использования вместе с программами, написанными на других языках, он должен подчинятся *общеязыковой спецификации* (Common Language Specification — CLS), которая определяет ряд общих свойств для разных .NET-совместимых языков. Соответствие CLS особенно важно при создании программных компонентов, предназначенных для применения в других языках. В CLS в качестве подмножества входит *общая система типов* (Common Type System — CTS), в которой определяются правила, касающиеся типов данных. И разумеется, в C# поддерживается как CLS, так и CTS.

ГЛАВА 2

Краткий обзор элементов C#

Наибольшие трудности в изучении языка программирования вызывает то обстоятельство, что ни один из его элементов не существует обособленно. Напротив, все элементы языка действуют совместно. Такая взаимосвязанность затрудняет рассмотрение одного аспекта C# безотносительно к другому. Поэтому для преодоления данного затруднения в этой главе дается краткий обзор нескольких средств языка C#, включая общую форму программы на C#, ряд основных управляющих и прочих операторов. Вместо того чтобы углубляться в детали, в этой главе основное внимание уделяется лишь самым общим принципам написания любой программы на C#. А большинство вопросов, затрагиваемых по ходу изложения материала этой главы, более подробно рассматриваются в остальных главах части I.

Объектно-ориентированное программирование

Основным понятием C# является объектно-ориентированное программирование (ООП). Методика ООП неотделима от C#, и поэтому все программы на C# являются объектно-ориентированными хотя бы в самой малой степени. В связи с этим очень важно и полезно усвоить основополагающие принципы ООП, прежде чем приступать к написанию самой простой программы на C#.

ООП представляет собой эффективный подход к программированию. Методики программирования претерпели существенные изменения с момента изобретения

компьютера, постепенно приспосабливаясь, главным образом, к повышению сложности программ. Когда, например, появились первые ЭВМ, программирование заключалось в ручном переключении на разные двоичные машинные команды с переднего пульта управления ЭВМ. Такой подход был вполне оправданным, поскольку программы состояли всего из нескольких сотен команд. Дальнейшее усложнение программ привело к разработке языка ассемблера, который давал программистам возможность работать с более сложными программами, используя символическое представление отдельных машинных команд. Постоянное усложнение программ вызвало потребность в разработке и внедрении в практику программирования таких языков высокого уровня, как, например, FORTRAN и COBOL, которые предоставляли программистам больше средств для того, чтобы как-то справиться с постоянно растущей сложностью программ. Но как только возможности этих первых языков программирования были полностью исчерпаны, появились разработки языков структурного программирования, в том числе и C.

На каждом этапе развития программирования появлялись методы и инструментальные средства для "обуздания" растущей сложности программ. И на каждом таком этапе новый подход вбирал в себя все самое лучшее из предыдущих, знаменуя собой прогресс в программировании. Это же можно сказать и об ООП. До ООП многие проекты достигали (а иногда и превышали) предел, за которым структурный подход к программированию оказывался уже неработоспособным. Поэтому для преодоления трудностей, связанных с усложнением программ, и возникла потребность в ООП.

ООП вобрало в себя все самые лучшие идеи структурного программирования, объединив их с рядом новых понятий. В итоге появился новый и лучший способ организации программ. В самом общем виде программа может быть организована одним из двух способов: вокруг кода (т.е. того, что фактически происходит) или же вокруг данных (т.е. того, что подвергается воздействию). Программы, созданные только методами структурного программирования, как правило, организованы вокруг кода. Такой подход можно рассматривать "как код, воздействующий на данные".

Совсем иначе работают объектно-ориентированные программы. Они организованы вокруг данных, исходя из главного принципа: "данные управляют доступом к коду". В объектно-ориентированном языке программирования определяются данные и код, которому разрешается воздействовать на эти данные. Следовательно, тип данных точно определяет операции, которые могут быть выполнены над данными.

Для поддержки принципов ООП все объектно-ориентированные языки программирования, в том числе и C#, должны обладать тремя общими свойствами: инкапсуляцией, полиморфизмом и наследованием. Рассмотрим каждое из этих свойств в отдельности.

Инкапсуляция

Инкапсуляция — это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. В объектно-ориентированном языке данные и код могут быть объединены в совершенно автономный *черный ящик*. Внутри такого ящика находятся все необходимые данные и код. Когда код и данные связываются вместе подобным образом, создается объект. Иными словами, *объект* — это элемент, поддерживающий инкапсуляцию.

В объекте код, данные или же и то и другое могут быть *закрытыми* или же *открытыми*. Закрытые данные или код известны и доступны только остальной части объекта. Это означает, что закрытые данные или код недоступны части программы, находящейся за пределами объекта. Если же данные или код оказываются открытыми, то они доступны другим частям программы, хотя и определены внутри объекта. Как правило, открытые части объекта служат для организации управляемого интерфейса с закрытыми частями.

Основной единицей инкапсуляции в C# является *класс*, который определяет форму объекта. Он описывает данные, а также код, который будет ими оперировать. В C# описание класса служит для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта.

Код и данные, составляющие вместе класс, называют *членами*. Данные, определяемые классом, называют *полями*, или *переменными экземпляра*. А код, оперирующий данными, содержится в *функциях-членах*, самым типичным представителем которых является *метод*. В C# метод служит в качестве аналога подпрограммы. (К числу других функций-членов относятся свойства, события и конструкторы.) Таким образом, методы класса содержат код, воздействующий на поля, определяемые этим классом.

Полиморфизм

Полиморфизм, что по-гречески означает "множество форм", — это свойство, позволяющее одному интерфейсу получать доступ к общему классу действий. Простым примером полиморфизма может служить руль автомашины, который выполняет одни и те же функции своеобразного интерфейса независимо от вида применяемого механизма управления автомашиной. Это означает, что руль действует одинаково независимо от вида рулевого управления: прямого действия, с усилением или реечной передачей. Следовательно, при вращении руля влево автомашина всегда поворачивает влево, какой бы вид управления в ней ни применялся. Главное преимущество единообразного интерфейса заключается в том, что, зная, как обращаться с рулем, вы сумеете водить автомашину любого типа.

Тот же самый принцип может быть применен и в программировании. Рассмотрим для примера *стек*, т.е. область памяти, функционирующую по принципу "последним пришел — первым обслужен". Допустим, что в программе требуются три разных типа стеков: один — для целых значений, другой — для значений с плавающей точкой, третий — для символьных значений. В данном примере алгоритм, реализующий все эти стеки, остается неизменным, несмотря на то, что в них сохраняются разнотипные данные. В языке, не являющемся объектно-ориентированным, для этой цели пришлось бы создать три разных набора стековых подпрограмм с разными именами. Но благодаря полиморфизму для реализации всех трех типов стеков в C# достаточно создать лишь один общий набор подпрограмм. Зная, как пользоваться одним стеком, вы сумеете воспользоваться и остальными.

В более общем смысле понятие полиморфизма нередко выражается следующим образом: "один интерфейс — множество методов". Это означает, что для группы взаимосвязанных действий можно разработать общий интерфейс. Полиморфизм помогает упростить программу, позволяя использовать один и тот же интерфейс для описания *общего класса действий*. Выбрать конкретное действие (т.е. метод) в каждом отдельном случае — это задача компилятора. Программисту не нужно делать это самому. Ему достаточно запомнить и правильно использовать общий интерфейс.

Наследование

Наследование представляет собой процесс, в ходе которого один объект приобретает свойства другого объекта. Это очень важный процесс, поскольку он обеспечивает принцип иерархической классификации. Если вдуматься, то большая часть знаний поддается систематизации благодаря иерархической классификации по нисходящей. Например, сорт яблок "Джонатан" входит в общую классификацию сортов *яблок*, которые, в свою очередь, относятся к классу *фруктов*, а те — к еще более крупному классу *пищевых продуктов*. Это означает, что класс пищевых продуктов обладает рядом свойств (съедобности, питательности и т.д.), которые по логике вещей распространяются и на его подкласс фруктов. Помимо этих свойств, класс фруктов обладает своими собственными свойствами (сочностью, сладостью и т.д.), которыми он отличается от других пищевых продуктов. У класса яблок имеются свои характерные особенности (растут на деревьях, не в тропиках и т.д.). Таким образом, сорт яблок "Джонатан" наследует свойства всех предшествующих классов, обладая в то же время свойствами, присущими только этому сорту яблок, например красной окраской кожицы с желтым бочком и характерным ароматом и вкусом.

Если не пользоваться иерархиями, то для каждого объекта пришлось бы явно определять все его свойства. А если воспользоваться наследованием, то достаточно определить лишь те свойства, которые делают объект особенным в его классе. Он может также наследовать общие свойства своего родителя. Следовательно, благодаря механизму наследования один объект становится отдельным экземпляром более общего класса.

Первая простая программа

А теперь самое время перейти к примеру конкретной программы на C#. Для начала скомпилируем и выполним короткую программу.

```
/*
  Это простая программа на C#.
  Назовем ее Example.cs.
*/

using System;

class Example {

  // Любая программа на C# начинается с вызова метода Main().
  static void Main() {
    Console.WriteLine("Простая программа на C#.");
  }
}
```

Основной средой для разработки программ на C# служит Visual Studio корпорации Microsoft. Для компилирования примеров всех программ, приведенных для примера в этой книге, в том числе и тех, где используются новые средства C# 4.0, вам потребуется Visual Studio 2010 или же более поздняя версия, поддерживающая C#.

Создавать, компилировать и выполнять программы на C#, используя Visual Studio, можно двумя способами: пользуясь, во-первых, интегрированной средой разработки Visual Studio, а во-вторых, — компилятором командной строки `csc.exe`. Далее описываются оба способа.

Применение компилятора командной строки csc.exe

Для коммерческой разработки программ вам, скорее всего, придется пользоваться интегрированной средой Visual Studio, хотя для некоторых читателей более удобным может оказаться компилятор, работающий в режиме командной строки, особенно для компилирования и выполнения примеров программ, приведенных в этой книге. Объясняется это тем, что для работы над отдельной программой не нужно создавать целый проект. Для этого достаточно написать программу, а затем скомпилировать и выполнить ее, причем все это делается из командной строки. Таким образом, если вы умеете пользоваться окном Командная строка (Command Prompt) и его интерфейсом в Windows, то компилятор командной строки окажется для вас более простым и оперативным инструментальным средством, чем интегрированная среда разработки.

ПРЕДОСТЕРЕЖЕНИЕ

Если вы не знаете, как пользоваться окном Командная строка, то вам лучше работать в интегрированной среде разработки Visual Studio. Ведь пытаться усвоить одновременно команды интерфейса Командная строка и элементы языка C# не так-то просто, несмотря на то, что запомнить эти команды совсем нетрудно.

Для написания и выполнения программ на C# с помощью компилятора командной строки выполните следующую несложную процедуру.

1. Введите исходный текст программы, используя текстовый редактор.
2. Скомпилируйте программу с помощью компилятора `csc.exe`.
3. Выполните программу.

Ввод исходного текста программы

Исходный текст примеров программ, приведенных в этой книге, доступен для загрузки по адресу `www.mhprofessional.com`. Но при желании вы можете сами ввести исходный текст этих программ вручную. Для этого воспользуйтесь избранным текстовым редактором, например Notepad. Но не забывайте, что вы должны создать файлы, содержащие простой, а не отформатированный текст, поскольку информация форматирования текста, сохраняемая в файле для обработки текста, может помешать нормальной работе компилятора C#. Введя исходный текст программы, присвойте ее файлу имя `Example.cs`.

Компилирование программы

Для компилирования программы на C# запустите на выполнение компилятор `csc.exe`, указав имя исходного файла в командной строке.

```
C:\>csc Example.cs
```

Компилятор `csc` создаст файл `Example.exe`, содержащий версию MSIL данной программы. Несмотря на то что псевдокод MSIL не является исполняемым кодом, он содержится в исполняемом файле с расширением `.exe`. Среда CLR автоматически вызывает JIT-компилятор при попытке выполнить файл `Example.exe`. Следует, однако, иметь в виду, что если попытаться выполнить файл `Example.exe` (или любой другой

исполняемый файл, содержащий псевдокод MSIL) на том компьютере, где среда .NET Framework не установлена, то программа не будет выполнена, поскольку на этом компьютере отсутствует среда CLR.

ПРИМЕЧАНИЕ

Прежде чем запускать на выполнение компилятор `csc.exe`, откройте окно Командная строка, настроенное под Visual Studio. Для этого проще всего выбрать команду Visual Studio⇨Инструменты Visual Studio⇨Командная строка Visual Studio (Visual Studio⇨Visual Studio Tools⇨Visual Studio Command Prompt) из меню Пуск⇨Все программы (Start⇨All Programs) на панели задач Windows. Кроме того, вы можете открыть ненастроенное окно Командная строка, а затем выполнить командный файл `vsvars32.bat`, входящий в состав Visual Studio.

Выполнение программы

Для выполнения программы введите ее имя в командной строке следующим образом.

```
C:\>Example
```

В результате выполнения программы на экране появится такая строка.

```
Простая программа на C#.
```

Применение интегрированной среды разработки Visual Studio

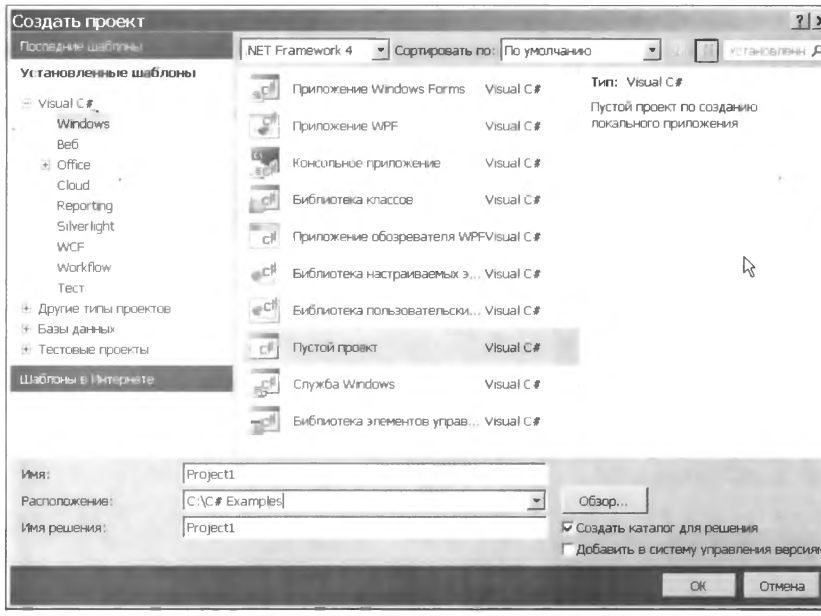
Visual Studio представляет собой интегрированную среду разработки программ, созданную корпорацией Microsoft. Такая среда дает возможность править, компилировать, выполнять и отлаживать программы на C#, не покидая эту грамотно организованную среду. Visual Studio предоставляет не только все необходимые средства для работы с программами, но и помогает правильно организовать их. Она оказывается наиболее эффективной для работы над крупными проектами, хотя может быть с тем же успехом использована и для разработки небольших программ, например, тех, которые приведены в качестве примера в этой книге.

Ниже приведена краткая процедура правки, компилирования и выполнения программы на C# в интегрированной среде разработки Visual Studio 2010. При этом предполагается, что интегрированная среда разработки входит в состав пакета Visual Studio 2010 Professional. В других версиях Visual Studio возможны незначительные отличия.

1. Создайте новый (пустой) проект C#, выбрав команду Файл⇨Создать⇨Проект (File⇨New⇨Project). Затем выберите элемент Windows из списка Установленные шаблоны (Installed Templates) и далее — шаблон Пустой проект (Empty Project), как показано на рисунке.

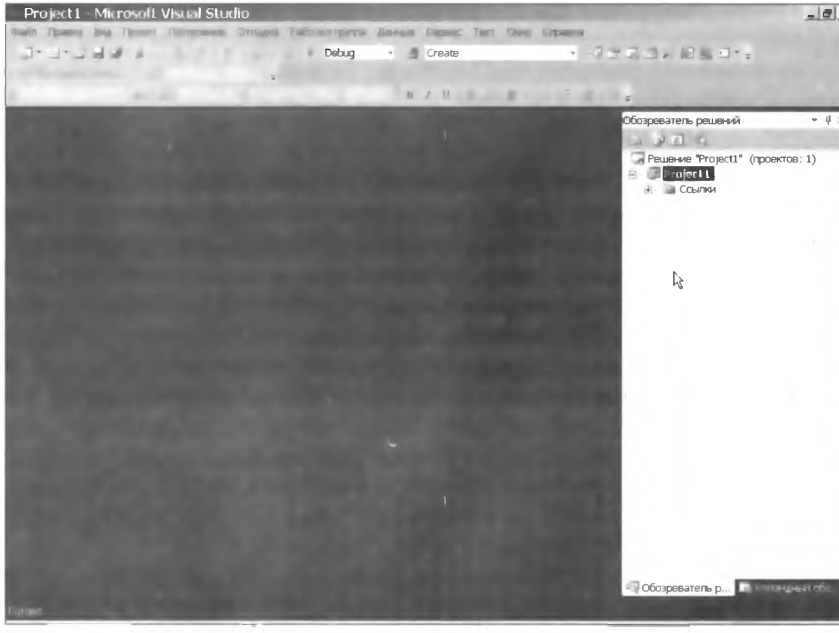
ПРИМЕЧАНИЕ

Имя и местоположение вашего проекта может отличаться от того, что показано здесь.



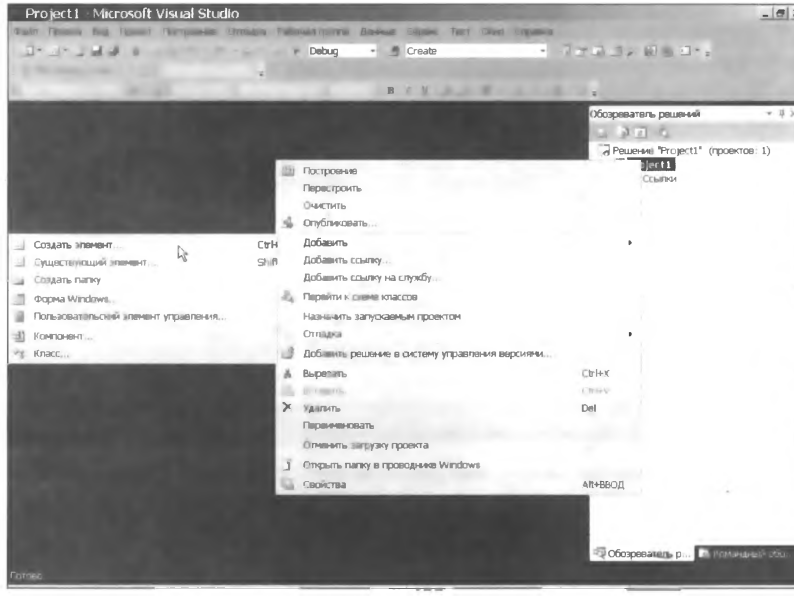
Щелкните на кнопке OK, чтобы создать проект.

2. После создания нового проекта среда Visual Studio будет выглядеть так, как показано на рисунке.

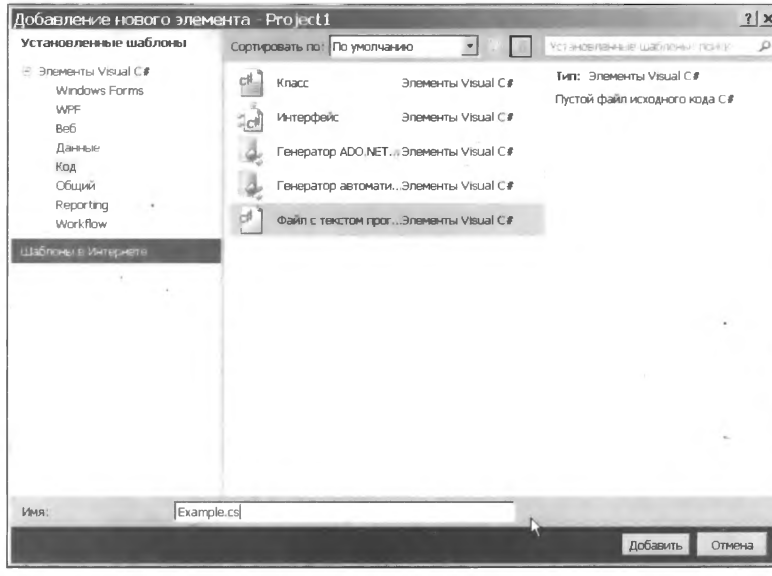


Если по какой-либо причине окно Обозреватель решений (Solution Explorer) будет отсутствовать, откройте его с помощью команды Вид⇨Обозреватель решений (View⇨Solution Explorer).

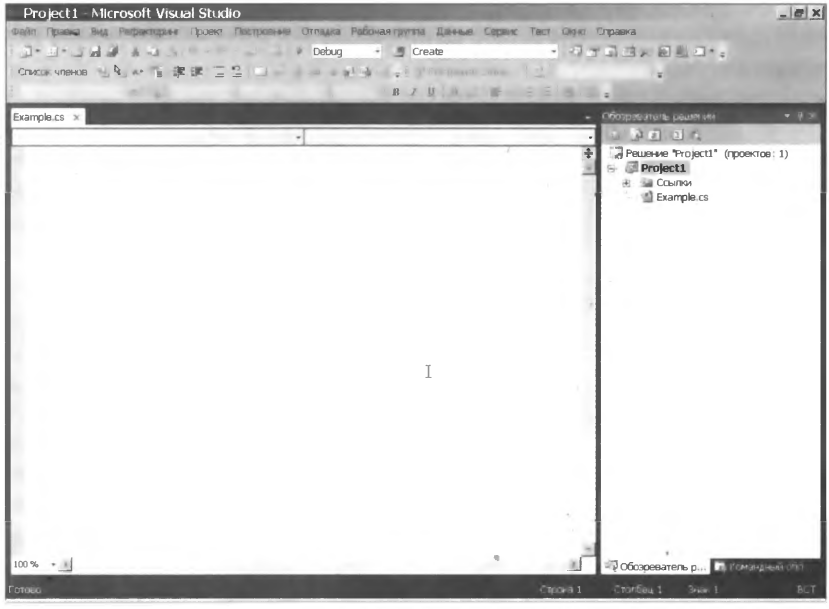
3. На данном этапе проект пуст, и вам нужно ввести в него файл с исходным текстом программы на C#. Для этого щелкните правой кнопкой мыши на имени проекта (в данном случае — `Project1`) в окне **Обозреватель решений**, а затем выберите команду **Добавить** (Add) из контекстного меню. В итоге появится подменю, показанное на рисунке.



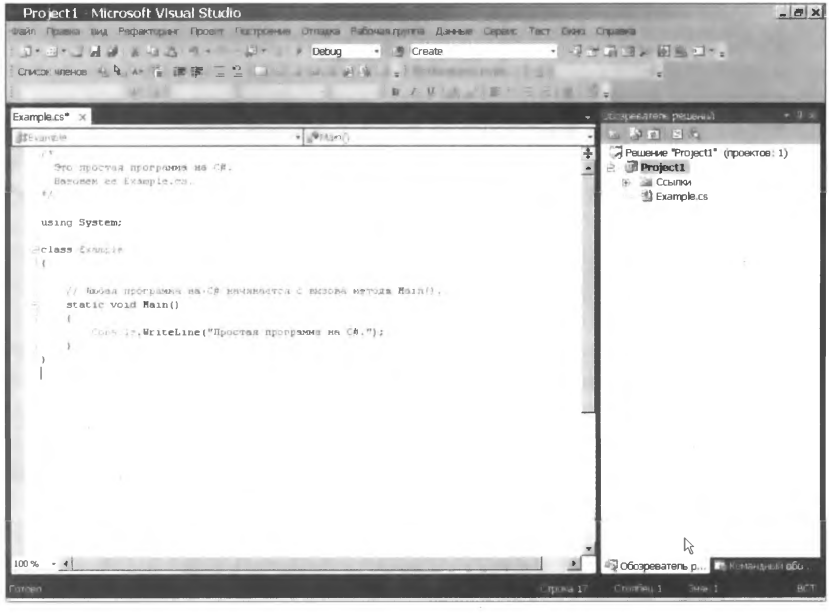
4. Выберите команду **Создать элемент** (New Item), чтобы открыть диалоговое окно **Добавление нового элемента** (Add New Item). Выберите сначала элемент **Код** (Code) из списка **Установленные шаблоны**, а затем шаблон **Файл с текстом программы** (Code File) и измените имя файла на `Example.cs`, как показано на рисунке.



5. Введите выбранный файл в проект, щелкнув на кнопке **Добавить**. После этого экран будет выглядеть так, как показано на рисунке.



6. Введите исходный текст программы в окне с меткой `Example.cs`, после чего сохраните этот текст в файле. (Исходный текст примеров программ, приведенных в этой книге, можно свободно загрузить по адресу `www.mhprofessional.com`, чтобы не вводить его каждый раз вручную.) По завершении ввода исходного текста программы экран будет выглядеть так, как показано на рисунке.



7. Скомпилируйте программу, выбрав команду Построение⇨Построить решение (Build⇨Build Solution).
8. Выполните программу, выбрав команду Отладка⇨Запуск без отладки (Debug⇨Start Without Debugging). В результате выполнения программы откроется окно, показанное на рисунке.



Как следует из приведенной выше процедуры, компилирование коротких программ в интегрированной среде разработки требует выполнения немалого числа шагов. Но для каждого примера программы из этой книги вам совсем не обязательно создавать новый проект. Вместо этого вы можете пользоваться одним и тем же проектом C#. С этой целью удалите текущий исходный файл и введите новый. Затем перекомпилируйте и выполните программу. Благодаря этому существенно упрощается весь процесс разработки коротких программ. Однако для разработки реальных приложений каждой программе потребуется отдельный проект.

ПРИМЕЧАНИЕ

Приведенных выше инструкций достаточно для компилирования и выполнения примеров программ, представленных в этой книге, но если вы собираетесь пользоваться Visual Studio как основной средой для разработки программ, вам придется более подробно ознакомиться с ее возможностями и средствами. Это весьма эффективная среда разработки программ, помогающая поддерживать крупные проекты на поддающемся управлению организационном уровне. Данная интегрированная среда разработки позволяет также правильно организовать файлы и связанные с проектом ресурсы. Поэтому целесообразно потратить время и приложить усилия, чтобы приобрести необходимые навыки работы в среде Visual Studio.

Построчный анализ первого примера программы

Несмотря на то что пример программы `Example.cs` довольно краток, в нем демонстрируется ряд ключевых средств, типичных для всех программ на C#. Проанализируем более подробно каждую строку этой программы, начиная с ее имени.

В отличие от ряда других языков программирования, и в особенности Java, где имя файла программы имеет большое значение, имя программы на C# может быть произвольным. Ранее вам было предложено присвоить программе из первого примера имя `Example.cs`, чтобы успешно скомпилировать и выполнить ее, но в C# файл с исходным текстом этой программы можно было бы назвать как угодно. Например, его можно было назвать `Sample.cs`, `Test.cs` или даже `X.cs`.

В файлах с исходным текстом программ на C# условно принято расширение `.cs`, и это условие вы должны соблюдать. Кроме того, многие программисты называют файлы с исходным текстом своих программ по имени основного класса, определенного в программе. Именно поэтому в рассматриваемом здесь примере было выбрано имя файла `Example.cs`. Но поскольку имена программ на C# могут быть произвольными, то они не указываются в большинстве примеров программ, приведенных в настоящей книге. Поэтому вы вольны сами выбирать для них имена.

Итак, анализируемая программа начинается с таких строк.

```
/*
 Это простая программа на C#.
 Назовем ее Example.cs.
*/
```

Эти строки образуют *комментарий*. Как и в большинстве других языков программирования, в C# допускается вводить комментарии в файл с исходным текстом программы. Содержимое комментария игнорируется компилятором. Но, с другой стороны, в комментарии дается краткое описание или пояснение работы программы для всех, кто читает ее исходный текст. В данном случае в комментарии дается описание программы и напоминание о том, что ее исходный файл называется `Example.cs`. Разумеется, в комментариях к реальным приложениям обычно поясняется работа отдельных частей программы или же функции конкретных средств.

В C# поддерживаются три стиля комментариев. Один из них приводится в самом начале программы и называется *многострочным комментарием*. Этот стиль комментария должен начинаться символами `/*` и оканчиваться символами `*/`. Все, что находится между этими символами, игнорируется компилятором. Как следует из его названия, многострочный комментарий может состоять из нескольких строк.

Рассмотрим следующую строку программы.

```
using System;
```

Эта строка означает, что в программе используется пространство имен `System`. В C# *пространство имен* определяет область объявлений. Подробнее о пространстве имен речь пойдет далее в этой книге, а до тех пор поясним вкратце его назначение. Благодаря пространству имен одно множество имен отделяется от других. По существу, имена, объявляемые в одном пространстве имен, не вступают в конфликт с именами, объявляемыми в другом пространстве имен. В анализируемой программе используется пространство имен `System`, которое зарезервировано для элементов, связанных с библиотекой классов среды .NET Framework, применяемой в C#. Ключевое слово `using` просто констатирует тот факт, что в программе используются имена в заданном пространстве имен. (Попутно обратим внимание на весьма любопытную возможность создавать собственные пространства имен, что особенно полезно для работы над крупными проектами.)

Перейдем к следующей строке программы.

```
class Example {
```

В этой строке ключевое слово `class` служит для объявления вновь определяемого класса. Как упоминалось выше, класс является основной единицей инкапсуляции в C#, а `Example` — это имя класса. Определение класса начинается с открывающей фигурной скобки (`{`) и оканчивается закрывающей фигурной скобкой (`}`). Элементы, заключенные в эти фигурные скобки, являются членами класса. Не вдаваясь пока что

в подробности, достаточно сказать, что в C# большая часть действий, выполняемых в программе, происходит именно в классе.

Следующая строка программы содержит *однострочный комментарий*.

```
// Любая программа на C# начинается с вызова метода Main().
```

Это второй стиль комментариев, поддерживаемых в C#. Однострочный комментарий начинается и оканчивается символами //. Несмотря на различие стилей комментариев, программисты нередко пользуются многострочными комментариями для более длинных примечаний и однострочными комментариями для коротких, построчных примечаний к программе. (Третий стиль комментариев, поддерживаемых в C#, применяется при создании документации и описывается в приложении А.)

Перейдем к анализу следующей строки программы.

```
static void Main() {
```

Эта строка начинается с метода `Main()`. Как упоминалось выше, в C# подпрограмма называется методом. И, как поясняется в предшествующем комментарии, именно с этой строки начинается выполнение программы. Выполнение всех приложений C# начинается с вызова метода `Main()`. Разбирать полностью значение каждого элемента данной строки пока что не имеет смысла, потому что для этого нужно знать ряд других средств C#. Но поскольку данная строка используется во многих примерах программ, приведенных в этой книге, то проанализируем ее вкратце.

Данная строка начинается с ключевого слова `static`. Метод, определяемый ключевым словом `static`, может вызываться до создания объекта его класса. Необходимость в этом объясняется тем, что метод `Main()` вызывается при запуске программы. Ключевое слово `void` указывает на то, что метод `Main()` не возвращает значение. В дальнейшем вы узнаете, что методы могут также возвращать значения. Пустые круглые скобки после имени метода `Main` означают, что этому методу не передается никакой информации. Теоретически методу `Main()` можно передать информацию, но в данном примере этого не делается. И последним элементом анализируемой строки является символ `{`, обозначающий начало тела метода `Main()`. Весь код, составляющий тело метода, находится между открывающими и закрывающими фигурными скобками.

Рассмотрим следующую строку программы. Обратите внимание на то, что она находится внутри метода `Main()`.

```
Console.WriteLine("Простая программа на C#.");
```

В этой строке осуществляется вывод на экран текстовой строки `"Простая программа на C#."`. Сам вывод выполняется встроенным методом `WriteLine()`. В данном примере метод `WriteLine()` выводит на экран строку, которая ему передается. Информация, передаваемая методу, называется *аргументом*. Помимо текстовых строк, метод `WriteLine()` позволяет выводить на экран другие виды информации. Анализируемая строка начинается с `Console` — имени предопределенного класса, поддерживающего ввод-вывод на консоль. Сочетание обозначений `Console` и `WriteLine()` указывает компилятору на то, что метод `WriteLine()` является членом класса `Console`. Применение в C# объекта для определения вывода на консоль служит еще одним свидетельством объектно-ориентированного характера этого языка программирования.

Обратите внимание на то, что оператор, содержащий вызов метода `WriteLine()`, оканчивается точкой с запятой, как, впрочем, и рассматривавшаяся ранее директива `using System`. Как правило, операторы в C# оканчиваются точкой с запятой. Исключением из этого правила служат блоки, которые начинаются символом `{`

и оканчиваются символом }. Именно поэтому строки программы с этими символами не оканчиваются точкой с запятой. Блоки обеспечивают механизм группирования операторов и рассматриваются далее в этой главе.

Первый символ } в анализируемой программе завершает метод Main(), а второй — определение класса Example.

И наконец, в C# различаются прописные и строчные буквы. Несоблюдение этого правила может привести к серьезным осложнениям. Так, если вы неумышленно наберете main вместо Main или же writeline вместо WriteLine, анализируемая программа окажется ошибочной. Более того, компилятор C# не предоставит возможность выполнить классы, которые не содержат метод Main(), хотя и *скомпилирует* их. Поэтому если вы неверно наберете имя метода Main, то получите от компилятора сообщение об ошибке, уведомляющее о том, что в исполняемом файле Example.exe не определена точка входа.

Обработка синтаксических ошибок

Если вы только начинаете изучать программирование, то вам следует научиться правильно истолковывать (и реагировать на) ошибки, которые могут появиться при попытке скомпилировать программу. Большинство ошибок компиляции возникает в результате опечаток при наборе исходного текста программы. Все программисты рано или поздно обнаруживают, что при наборе исходного текста программы очень легко сделать опечатку. Правда, если вы наберете что-нибудь неправильно, компилятор выдаст соответствующее сообщение о *синтаксической ошибке* при попытке скомпилировать вашу программу. В таком сообщении обычно указывается номер строки исходного текста программы, где была обнаружена ошибка, а также кратко описывается характер ошибки.

Несмотря на всю полезность сообщений о синтаксических ошибках, выдаваемых компилятором, они иногда вводят в заблуждение. Ведь компилятор C# пытается извлечь какой-то смысл из исходного текста, как бы он ни был набран. Именно по этой причине ошибка, о которой сообщает компилятор, не всегда отражает настоящую причину возникшего затруднения. Неумышленный пропуск открывающей фигурной скобки после метода Main() в рассмотренном выше примере программы приводит к появлению приведенной ниже последовательности сообщений об ошибках при компиляции данной программы компилятором командной строки csc. (Аналогичные ошибки появляются при компиляции в интегрированной среде разработки Visual Studio.)

```
EX1.CS(12,21): ошибка CS1002: ; ожидалось
EX1.CS(13,22): ошибка CS1519: Недопустимая лексема '(' в
объявлении члена класса, структуры или интерфейса
EX1.CS(15,1): ошибка CS1022: Требуется определение типа
или пространства имен либо признак конца файла
```

Очевидно, что первое сообщение об ошибке нельзя считать верным, поскольку пропущена не точка с запятой, а фигурная скобка. Два других сообщения об ошибках вносят такую же путаницу.

Из всего изложенного выше следует, что если программа содержит синтаксическую ошибку, то сообщения компилятора не следует понимать буквально, поскольку они могут ввести в заблуждение. Для выявления истинной причины ошибки может

потребоваться критический пересмотр сообщения об ошибке. Кроме того, полезно проанализировать несколько строк кода, предшествующих той строке, в которой обнаружена сообщаемая ошибка. Иногда об ошибке сообщается лишь через несколько строк после того места, где она действительно произошла.

Незначительное изменение программы

Несмотря на то что приведенная ниже строка указывается во всех примерах программ, рассматриваемых в этой книге, формально она не нужна.

```
using System;
```

Тем не менее она указывается ради удобства. Эта строка не нужна потому, что в C# можно всегда *полностью определить* имя с помощью пространства имен, к которому оно принадлежит. Например, строку

```
Console.WriteLine("Простая программа на C#.");
```

можно переписать следующим образом.

```
System.Console.WriteLine("Простая программа на C#.");
```

Таким образом, первый пример программы можно видоизменить так.

```
// В эту версию не включена строка "using System;".

class Example {

  // Любая программа на C# начинается с вызова метода Main().
  static void Main() {

    // Здесь имя Console.WriteLine полностью определено.
    System.Console.WriteLine("Простая программа на C#.");
  }
}
```

Указывать пространство имен `System` всякий раз, когда используется член этого пространства, — довольно утомительное занятие, и поэтому большинство программистов на C# вводят директиву `using System` в начале своих программ, как это сделано в примерах всех программ, приведенных в данной книге. Следует, однако, иметь в виду, что любое имя можно всегда определить, явно указав его пространство имен, если в этом есть необходимость.

Вторая простая программа

В языке программирования, вероятно, нет более важной конструкции, чем переменная. *Переменная* — это именованная область памяти, для которой может быть установлено значение. Она называется переменной потому, что ее значение может быть изменено по ходу выполнения программы. Иными словами, содержимое переменной подлежит изменению и не является постоянным.

В приведенной ниже программе создаются две переменные — `x` и `y`.

```
// Эта программа демонстрирует применение переменных.

using System;

class Example2 {
  static void Main() {
      int x; // здесь объявляется переменная
      int y; // здесь объявляется еще одна переменная

      x = 100; // здесь переменной x присваивается значение 100

      Console.WriteLine("x содержит " + x);

      y = x / 2;

      Console.Write("y содержит x / 2: ");
      Console.WriteLine(y);
  }
}
```

Выполнение этой программы дает следующий результат.

```
x содержит 100
y содержит x / 2: 50
```

В этой программе вводится ряд новых понятий. Прежде всего, в операторе

```
int x; // здесь объявляется переменная
```

объявляется переменная целочисленного типа с именем x. В C# все переменные должны объявляться до их применения. Кроме того, нужно обязательно указать тип значения, которое будет храниться в переменной. Это так называемый *тип* переменной. В данном примере в переменной x хранится целочисленное значение, т.е. целое число. Для объявления в C# переменной целочисленного типа перед ее именем указывается ключевое слово `int`. Таким образом, в приведенном выше операторе объявляется переменная x типа `int`.

В следующей строке объявляется вторая переменная с именем y.

```
int y; // здесь объявляется еще одна переменная
```

Как видите, эта переменная объявляется таким же образом, как и предыдущая, за исключением того, что ей присваивается другое имя.

В целом, для объявления переменной служит следующий оператор:

тип имя_переменной;

где *тип* — это конкретный тип объявляемой переменной, а *имя_переменной* — имя самой переменной. Помимо типа `int`, в C# поддерживается ряд других типов данных.

В следующей строке программы переменной x присваивается значение 100.

```
x = 100; // здесь переменной x присваивается значение 100
```

В C# оператор присваивания обозначается одиночным знаком равенства (=). Данный оператор выполняет копирование значения, расположенного справа от знака равенства, в переменную, находящуюся слева от него.

В следующей строке программы осуществляется вывод на экран текстовой строки "x содержит " и значения переменной x.

```
Console.WriteLine("x содержит " + x);
```

В этом операторе знак + обозначает, что значение переменной x выводится вслед за предшествующей ему текстовой строкой. Если обобщить этот частный случай, то с помощью знака операции + можно организовать сцепление какого угодно числа элементов в одном операторе с вызовом метода `WriteLine()`.

В следующей строке программы переменной y присваивается значение переменной x, деленное на 2.

```
y = x / 2;
```

В этой строке значение переменной x делится на 2, а полученный результат сохраняется в переменной y. Таким образом, после выполнения данной строки в переменной y содержится значение 50. При этом значение переменной x не меняется. Как и в большинстве других языков программирования, в C# поддерживаются все арифметические операции, в том числе и перечисленные ниже.

+	Сложение
-	Вычитание
*	Умножение
/	Деление

Рассмотрим две оставшиеся строки программы.

```
Console.Write("y содержит x / 2: ");
Console.WriteLine(y);
```

В этих строках обнаруживаются еще две особенности. Во-первых, для вывода текстовой строки "y содержит x / 2: " на экран используется встроенный метод `Write()`. После этой текстовой строки новая строка не следует. Это означает, что последующий вывод будет осуществлен в той же самой строке. Метод `Write()` подобен методу `WriteLine()`, за исключением того, что после каждого его вызова вывод не начинается с новой строки. И во-вторых, обратите внимание на то, что в вызове метода `WriteLine()` указывается только переменная y. Оба метода, `Write()` и `WriteLine()`, могут быть использованы для вывода значений любых встроенных в C# типов.

Прежде чем двигаться дальше, следует упомянуть еще об одной особенности объявления переменных. Две или более переменных можно указать в одном операторе объявления. Нужно лишь разделить их запятой. Например, переменные x и y могут быть объявлены следующим образом.

```
int x, y; // обе переменные объявляются в одном операторе
```

ПРИМЕЧАНИЕ

В C# внедрено средство, называемое **неявно типизированной переменной**. Неявно типизированными являются такие переменные, тип которых автоматически определяется компилятором. Подробнее неявно типизированные переменные рассматриваются в главе 3.

Другие типы данных

В предыдущем примере программы использовались переменные типа `int`. Но в переменных типа `int` могут храниться только целые числа. Их нельзя использовать в операциях с числами, имеющими дробную часть. Например, переменная типа `int` может содержать значение 18, но не значение 18,3. Правда, `int` — далеко не единственный тип данных, определяемых в C#. Для операций с числами, имеющими дробную часть, в C# предусмотрены два типа данных с плавающей точкой: `float` и `double`. Они обозначают числовые значения с одинарной и двойной точностью соответственно. Из этих двух типов чаще всего используется тип `double`.

Для объявления переменной типа `double` служит оператор

```
double result;
```

где `result` — это имя переменной типа `double`. А поскольку переменная `result` имеет тип данных с плавающей точкой, то в ней могут храниться такие числовые значения, как, например, 122,23, 0,034 или –19,0.

Для лучшего понимания отличий между типами данных `int` и `double` рассмотрим такой пример программы.

```
/*
  Эта программа демонстрирует отличия
  между типами данных int и double.
*/

using System;

class Example3 {
  static void Main() {
    int ivar; // объявить целочисленную переменную
    double dvar; // объявить переменную с плавающей точкой

    ivar = 100; // присвоить переменной ivar значение 100

    dvar = 100.0; // присвоить переменной dvar значение 100.0

    Console.WriteLine("Исходное значение ivar: " + ivar);
    Console.WriteLine("Исходное значение dvar: " + dvar);

    Console.WriteLine(); // вывести пустую строку

    // Разделить значения обеих переменных на 3.
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    Console.WriteLine("Значение ivar после деления: " + ivar);
    Console.WriteLine("Значение dvar после деления: " + dvar);
  }
}
```

Ниже приведен результат выполнения приведенной выше программы.

```
Исходное значение ivar: 100
Исходное значение dvar: 100
```

```
Значение ivar после деления: 33
Значение dvar после деления: 33.3333333333333
```

Как видите, при делении значения переменной `ivar` типа `int` на 3 остается лишь целая часть результата — 33, а дробная его часть теряется. В то же время при делении значения переменной `dvar` типа `double` на 3 дробная часть результата сохраняется.

Как демонстрирует данный пример программы, в числовых значениях с плавающей точкой следует использовать обозначение самой десятичной точки. Например, значение 100 в C# считается целым, а значение 100,0 — с плавающей точкой.

В данной программе обнаруживается еще одна особенность. Для вывода пустой строки достаточно вызвать метод `WriteLine()` без аргументов.

Типы данных с плавающей точкой зачастую используются в операциях с реальными числовыми величинами, где обычно требуется дробная часть числа. Так, приведенная ниже программа вычисляет площадь круга, используя значение 3,1416 числа "пи".

```
// Вычислить площадь круга.

using System;

class Circle {
  static void Main() {
    double radius;
    double area;

    radius = 10.0;
    area = radius * radius * 3.1416;

    Console.WriteLine("Площадь равна " + area);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Площадь равна 314.16
```

Очевидно, что вычисление площади круга не дало бы удовлетворительного результата, если бы при этом не использовались данные с плавающей точкой.

Два управляющих оператора

Выполнение программы внутри метода (т.е. в его теле) происходит последовательно от одного оператора к другому, т.е. по цепочке сверху вниз. Этот порядок выполнения программы можно изменить с помощью различных управляющих операторов, поддерживаемых в C#. Более подробно управляющие операторы будут рассмотрены в дальнейшем, а здесь они представлены вкратце, поскольку используются в последующих примерах программ.

Условный оператор

С помощью условного оператора `if` в C# можно организовать выборочное выполнение части программы. Оператор `if` действует в C# практически так же, как и оператор `IF` в любом другом языке программирования. В частности, с точки зрения синтак-

синтаксиса он тождествен операторам `if` в C, C++ и Java. Ниже приведена простейшая форма этого оператора.

```
if(условие) оператор;
```

Здесь *условие* представляет собой булево, т.е. логическое, выражение, принимающее одно из двух значений: "истина" или "ложь". Если условие истинно, то *оператор* выполняется. А если условие ложно, то выполнение программы происходит, минуя *оператор*. Ниже приведен пример применения условного оператора.

```
if(10 < 11) Console.WriteLine("10 меньше 11");
```

В данном примере условное выражение принимает истинное значение, поскольку 10 меньше 11, и поэтому метод `WriteLine()` выполняется. А теперь рассмотрим другой пример.

```
if(10 < 9) Console.WriteLine("не подлежит выводу");
```

В данном примере 10 не меньше 9. Следовательно, вызов метода `WriteLine()` не произойдет.

В C# определен полный набор операторов отношения, которые можно использовать в условных выражениях. Ниже перечислены все эти операторы и их обозначения.

Операция	Значение
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно

Далее следует пример еще одной программы, демонстрирующей применение условного оператора `if`.

```
// Продемонстрировать применение условного оператора if.

using System;

class IfDemo {
  static void Main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) Console.WriteLine("a меньше b");

    // Не подлежит выводу.
    if(a == b) Console.WriteLine("этого никто не увидит");

    Console.WriteLine();

    c = a - b; // c содержит -1
```

```
    Console.WriteLine("c содержит -1");
    if(c >= 0) Console.WriteLine("значение c неотрицательно");
    if(c < 0) Console.WriteLine("значение c отрицательно");

    Console.WriteLine();

    c = b - a; // теперь c содержит 1
    Console.WriteLine("c содержит 1");
    if(c >= 0) Console.WriteLine("значение c неотрицательно");
    if(c < 0) Console.WriteLine("значение c отрицательно ");
  }
}
```

Вот к какому результату приводит выполнение данной программы.

```
a меньше b

c содержит -1
значение c отрицательно

c содержит 1
значение c неотрицательно
```

Обратите внимание на еще одну особенность этой программы. В строке

```
int a, b, c;
```

три переменные, `a`, `b` и `c`, объявляются списком, разделяемым запятыми. Как упоминалось выше, если требуется объявить две или более переменные одного и того же типа, это можно сделать в одном операторе, разделив их имена запятыми.

Оператор цикла

Для повторного выполнения последовательности операций в программе можно организовать *цикл*. Язык C# отличается большим разнообразием циклических конструкций. Здесь будет рассмотрен оператор цикла `for`. Как и у оператора `if`, у оператора `for` в C# имеются аналоги в C, C++ и Java. Ниже приведена простейшая форма этого оператора.

```
for(инициализация; условие; итерация) оператор;
```

В самой общей форме в части *инициализация* данного оператора задается начальное значение переменной управления циклом. Часть *условие* представляет собой булево выражение, проверяющее значение переменной управления циклом. Если результат проверки истинен, то цикл продолжается. Если же он ложен, то цикл завершается. В части `итерация` определяется порядок изменения переменной управления циклом на каждом шаге цикла, когда он повторяется. Ниже приведен пример краткой программы, демонстрирующей применение оператора цикла `for`.

```
// Продемонстрировать применение оператора цикла for.

using System;

class ForDemo {
```

```
  static void Main() {
    int count;
    for(count = 0; count < 5; count = count+1)

      Console.WriteLine("Это подсчет: " + count);

    Console.WriteLine("Готово!");
  }
}
```

Вот как выглядит результат выполнения данной программы.

```
Это подсчет: 0
Это подсчет: 1
Это подсчет: 2
Это подсчет: 3
Это подсчет: 4
Готово!
```

В данном примере `count` выполняет роль переменной управления циклом. В инициализирующей части оператора цикла `for` задается нулевое значение этой переменной. В начале каждого шага цикла, включая и первый, проверяется условие `count < 5`. Если эта проверка дает истинный результат, то выполняется оператор, содержащий метод `WriteLine()`. Далее выполняется итерационная часть оператора цикла `for`, где значение переменной `count` увеличивается на 1. Этот процесс повторяется до тех пор, пока значение переменной `count` не достигнет величины 5. В этот момент проверка упомянутого выше условия дает ложный результат, что приводит к завершению цикла. Выполнение программы продолжается с оператора, следующего после цикла.

Любопытно, что в программах, профессионально написанных на C#, вы вряд ли увидите итерационную часть оператора цикла в том виде, в каком она представлена в приведенном выше примере программы, т.е. вы редко встретите следующую строку.

```
count = count + 1;
```

Дело в том, что в C# имеется специальный оператор инкремента, выполняющий приращение на 1 значение переменной, или так называемого операнда. Этот оператор обозначается двумя знаками + (++). Используя оператор инкремента, можно переписать приведенную выше строку следующим образом.

```
count++;
```

Таким образом, оператор цикла `for` из приведенного выше примера программы обычно записывается в следующем виде.

```
for(count = 0; count < 5; count++)
```

Опробуйте этот более краткий способ записи итерационной части цикла. Вы сами можете убедиться, что данный цикл выполняется так же, как и прежде.

В C# имеется также оператор декремента, обозначаемый двумя дефисами (--). Этот оператор уменьшает значение операнда на 1.

Использование кодовых блоков

Еще одним важным элементом C# является *кодовый блок*, который представляет собой группу операторов. Для его организации достаточно расположить операторы

между открывающей и закрывающей фигурными скобками. Как только кодовый блок будет создан, он станет логическим элементом, который можно использовать в любом месте программы, где применяется одиночный оператор. В частности, кодовый блок может служить адресатом операторов `if` и `for`. Рассмотрим следующий оператор `if`.

```
if(w < h) {
  v = w * h;
  w = 0;
}
```

Если в данном примере кода значение переменной `w` меньше значения переменной `h`, то оба оператора выполняются в кодовом блоке. Они образуют внутри кодового блока единый логический элемент, причем один не может выполняться без другого. Таким образом, если требуется логически связать два (или более) оператора, то для этой цели следует создать кодовый блок. С помощью кодовых блоков можно более эффективно и ясно реализовать многие алгоритмы.

Ниже приведен пример программы, в которой кодовый блок служит для того, чтобы исключить деление на нуль.

```
// Продемонстрировать применение кодового блока.

using System;

class BlockDemo {
  static void Main() {
    int i, j, d;

    i = 5;
    j = 10;

    // Адресатом этого оператора if служит кодовый блок.
    if(i != 0) {
        Console.WriteLine("i не равно нулю");
        d = j / i;
        Console.WriteLine("j / i равно " + d);
    }
  }
}
```

Вот к какому результату приводит выполнение данной программы.

```
i не равно нулю
j / i равно 2
```

В данном примере адресатом оператора `if` служит кодовый блок, а не единственный оператор. Если условие, управляющее оператором `if`, оказывается истинным, то выполняются три оператора в кодовом блоке. Попробуйте задать нулевое значение переменной `i`, чтобы посмотреть, что из этого получится.

Рассмотрим еще один пример, где кодовый блок служит для вычисления суммы и произведения чисел от 1 до 10.

```
// Вычислить сумму и произведение чисел от 1 до 10.

using System;
```

```
class ProdSum {
  static void Main() {
    int prod;
    int sum;
    int i;

    sum = 0;
    prod = 1;

    for(i=1; i <= 10; i++) {
        sum = sum + i;
        prod = prod * i;
    }

    Console.WriteLine("Сумма равна " + sum);
    Console.WriteLine("Произведение равно " + prod);
  }
}
```

Ниже приведен результат выполнения данной программы.

```
Сумма равна 55
Произведение равно 3628800
```

В данном примере внутри кодового блока организуется цикл для вычисления суммы и произведения. В отсутствие такого блока для достижения того же самого результата пришлось бы организовать два отдельных цикла.

И последнее: кодовые блоки не снижают эффективность программ во время их выполнения. Иными словами, наличие символов { и }, обозначающих кодовый блок, никоим образом не замедляет выполнение программы. В действительности применение кодовых блоков, как правило, приводит к повышению быстродействия и эффективности программ, поскольку они упрощают программирование определенных алгоритмов.

Точка с запятой и оформление исходного текста программы

В C# точка с запятой обозначает конец оператора. Это означает, что каждый оператор в отдельности должен оканчиваться точкой с запятой.

Как вы уже знаете, кодовый блок представляет собой набор логически связанных операторов, заключенных в фигурные скобки. Блок не оканчивается точкой с запятой, поскольку он состоит из группы операторов. Вместо этого окончание кодового блока обозначается закрывающей фигурной скобкой.

В C# конец строки не означает конец оператора — о его окончании свидетельствует только точка с запятой. Именно поэтому оператор можно поместить в любой части строки. Например, на языке C# строки кода

```
x = y;
y = y + 1;
Console.WriteLine(x + " " + y);
```

означают то же самое, что и строка кода

```
x = y; y = y + 1; Console.WriteLine(x + " " + y);
```

Более того, составные элементы оператора можно располагать в отдельных строках. Например, следующий фрагмент кода считается в C# вполне допустимым.

```
Console.WriteLine("Это длинная строка вывода" +
                  x + y + z +
                  "дополнительный вывод");
```

Такое разбиение длинных строк нередко применяется для того, чтобы сделать исходный текст программы более удобным для чтения. Оно помогает также исключить заворачивание слишком длинных строк.

Возможно, вы уже обратили внимание на то, что в предыдущих примерах программ некоторые операторы были набраны с отступом. В C# допускается свободная форма записи. Это означает, что взаимное расположение операторов в строке не имеет особого значения. Но с годами в программировании сложился общепринятый стиль оформления исходного текста программ с отступами, что существенно облегчает чтение этого текста. Именно этому стилю следуют примеры программ в данной книге, что рекомендуется делать и вам. В соответствии с этим стилем следует делать отступ (в виде нескольких пробелов) после каждой открывающей фигурной скобки и возвращаться назад после закрывающей фигурной скобки. А для некоторых операторов даже требуется дополнительный отступ, но об этом речь пойдет далее.

Ключевые слова C#

Основу любого языка программирования составляют его ключевые слова, поскольку они определяют средства, встроенные в этот язык. В C# определены два общих типа ключевых слов: *зарезервированные* и *контекстные*. Зарезервированные ключевые слова нельзя использовать в именах переменных, классов или методов. Их можно использовать только в качестве ключевых слов. Именно поэтому они и называются *зарезервированными*. Их иногда еще называют *зарезервированными словами*, или *зарезервированными идентификаторами*. В настоящее время в версии 4.0 языка C# определено 77 зарезервированных ключевых слов (табл. 2.1).

Таблица 2.1. Ключевые слова, зарезервированные в языке C#

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	volatile
void	while			

Кроме того, в версии C# 4.0 определены 18 контекстных ключевых слов, которые приобретают особое значение в определенном контексте. В таком контексте они выполняют роль ключевых слов, а вне его они могут использоваться в именах других элементов программы, например в именах переменных. Следовательно, контекстные ключевые слова формально не являются зарезервированными. Но, как правило, их следует считать зарезервированными, избегая их применения в любых других целях. Ведь применение контекстного ключевого слова в качестве имени какого-нибудь другого элемента программы может привести к путанице, и поэтому считается многими программистами плохой практикой. Контекстные ключевые слова приведены в табл. 2.2.

Таблица 2.2. Контекстные ключевые слова в C#

`add`	`dynamic`	`from`	`get`	`global`
`group`	`into`	`join`	`let`	`orderby`
`partial`	`remove`	`select`	`set`	`value`
`var`	`where`	`yield`		

Идентификаторы

В C# идентификатор представляет собой имя, присваиваемое методу, переменной или любому другому определяемому пользователем элементу программы. Идентификаторы могут состоять из одного или нескольких символов. Имена переменных могут начинаться с любой буквы алфавита или знака подчеркивания. Далее может следовать буква, цифра или знак подчеркивания. С помощью знака подчеркивания можно повысить удобочитаемость имени переменной, как, например, `line_count`. Но идентификаторы, содержащие два знака подчеркивания подряд, например, `max__value`, зарезервированы для применения в компиляторе. Прописные и строчные буквы в C# различаются. Так, например `myvar` и `MyVar` — это разные имена переменных. Ниже приведены некоторые примеры допустимых идентификаторов.

`Test`	`x`	`y2`	`MaxLoad`
`up`	`_top`	`my_var`	`sample23`

Помните, что идентификатор не может начинаться с цифры. Например, `12x` — недействительный идентификатор. Хорошая практика программирования требует выбирать идентификаторы, отражающие назначение или применение именуемых элементов.

Несмотря на то что зарезервированные ключевые слова нельзя использовать в качестве идентификаторов, в C# разрешается применять ключевое слово с предшествующим знаком `@` в качестве допустимого идентификатора. Например, `@for` — действительный идентификатор. В этом случае в качестве идентификатора фактически служит ключевое слово `for`, а знак `@` просто игнорируется. Ниже приведен пример программы, демонстрирующей применение идентификатора со знаком `@`.

```
// Продемонстрировать применение идентификатора со знаком @.

using System;

class IdTest {
   static void Main() {
    int @if; // применение ключевого слова if
```

```
            //в качестве идентификатора

    for(@if = 0; @if < 10; @if++)
        Console.WriteLine("@if равно " + @if);
  }
}
```

Приведенный ниже результат выполнения этой программы подтверждает, что `@if` правильно интерпретируется в качестве идентификатора.

```
@if равно 0
@if равно 1
@if равно 2
@if равно 3
@if равно 4
@if равно 5
@if равно 6
@if равно 7
@if равно 8
@if равно 9
```

Откровенно говоря, применять ключевые слова со знаком `@` в качестве идентификаторов не рекомендуется, кроме особых случаев. Помимо того, знак `@` может предшествовать любому идентификатору, но такая практика программирования считается плохой.

Библиотека классов среды .NET Framework

В примерах программ, представленных в этой главе, применялись два встроенных метода: `WriteLine()` и `Write()`. Как упоминалось выше, эти методы являются членами класса `Console`, относящегося к пространству имен `System`, которое определяется в библиотеке классов для среды .NET Framework. Ранее в этой главе пояснялось, что среда C# опирается на библиотеку классов, предназначенную для среды .NET Framework, чтобы поддерживать операции ввода-вывода, обработку строк, работу в сети и графические пользовательские интерфейсы. Поэтому, вообще говоря, C# представляет собой определенное сочетание самого языка C# и стандартных классов .NET. Как будет показано далее, библиотека классов обеспечивает функциональные возможности, являющиеся неотъемлемой частью любой программы на C#. Для того чтобы научиться программировать на C#, нужно знать не только сам язык, но и уметь пользоваться стандартными классами. Различные элементы библиотеки классов для среды .NET Framework рассматриваются в части I этой книги, а в части II — сама библиотека по отдельным ее составляющим.

ГЛАВА 3

Типы данных, литералы и переменные

В этой главе рассматриваются три основополагающих элемента C#: типы данных, литералы и переменные. В целом, типы данных, доступные в языке программирования, определяют те виды задач, для решения которых можно применять данный язык. Как и следовало ожидать, в C# предоставляется богатый набор встроенных типов данных, что делает этот язык пригодным для самого широкого применения. Любой из этих типов данных может служить для создания переменных и констант, которые в языке C# называются *литералами*.

О значении типов данных

Типы данных имеют особенное значение в C#, поскольку это строго типизированный язык. Это означает, что все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Следовательно, строгий контроль типов позволяет исключить ошибки и повысить надежность программ. Для обеспечения контроля типов все переменные, выражения и значения должны принадлежать к определенному типу. Такого понятия, как "бестиповая" переменная, в данном языке программирования вообще не существует. Более того, тип значения определяет те операции, которые разрешается выполнять над ним. Операция, разрешенная для одного типа данных, может оказаться недопустимой для другого.

ПРИМЕЧАНИЕ

В версии C# 4.0 внедрен новый тип данных, называемый `dynamic` и приводящий к отсрочке контроля типов до времени выполнения, вместо того чтобы производить подобный контроль во время компиляции. Поэтому тип `dynamic` является исключением из обычного правила контроля типов во время компиляции. Подробнее о типе `dynamic` речь пойдет в главе 17.

Типы значений в C#

В C# имеются две общие категории встроенных типов данных: *типы значений* и *ссылочные типы*. Они отличаются по содержимому переменной. Если переменная относится к типу значения, то она содержит само значение, например 3,1416 или 212. А если переменная относится к ссылочному типу, то она содержит ссылку на значение. Наиболее распространенным примером использования ссылочного типа является класс, но о классах и ссылочных типах речь пойдет далее в этой книге. А здесь рассматриваются типы значений.

В основу языка C# положены 13 типов значений, перечисленных в табл. 3.1. Все они называются *простыми типами*, поскольку состоят из единственного значения. (Иными словами, они не состоят из двух или более значений.) Они составляют основу системы типов C#, предоставляя простейшие, низкоуровневые элементы данных, которыми можно оперировать в программе. Простые типы данных иногда еще называют *примитивными*.

Таблица. 3.1. Типы значений в C#

Тип	Значение
`bool`	Логический, предоставляет два значения: "истина" или "ложь"
`byte`	8-разрядный целочисленный без знака
`char`	Символьный
`decimal`	Десятичный (для финансовых расчетов)
`double`	С плавающей точкой двойной точности
`float`	С плавающей точкой одинарной точности
`int`	Целочисленный
`long`	Длинный целочисленный
`sbyte`	8-разрядный целочисленный со знаком
`short`	Короткий целочисленный
`uint`	Целочисленный без знака
`ulong`	Длинный целочисленный без знака
`ushort`	Короткий целочисленный без знака

В C# строго определены пределы и характер действия каждого типа значения. Исходя из требований к переносимости программ, C# не допускает в этом отношении никаких компромиссов. Например, тип `int` должен быть одинаковым во всех средах выполнения. Но в этом случае отпадает необходимость переписывать код для кон-

кретной платформы. И хотя строгое определение размерности типов значений может стать причиной незначительного падения производительности в некоторых средах, эта мера необходима для достижения переносимости программ.

ПРИМЕЧАНИЕ

Помимо простых типов, в C# определены еще три категории типов значений: перечисления, структуры и обнуляемые типы. Все они рассматриваются далее в этой книге.

Целочисленные типы

В C# определены девять целочисленных типов: `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`. Но тип `char` применяется, главным образом, для представления символов и поэтому рассматривается далее в этой главе. Остальные восемь целочисленных типов предназначены для числовых расчетов. Ниже представлены их диапазон представления чисел и разрядность в битах.

Тип	Разрядность в битах	Диапазон представления чисел
`byte`	8	0-255
`sbyte`	8	-128-127
`short`	16	-32 768-32 767
`ushort`	16	0-65 535
`int`	32	-2 147 483 648-2 147 483 647
`uint`	32	0-4 294 967 295
`long`	64	-9 223 372 036 854 775 808-9 223 372 036 854 775 807
`ulong`	64	0-18 446 744 073 709 551 615

Как следует из приведенной выше таблицы, в C# определены оба варианта различных целочисленных типов: со знаком и без знака. Целочисленные типы со знаком отличаются от аналогичных типов без знака способом интерпретации старшего разряда целого числа. Так, если в программе указано целочисленное значение со знаком, то компилятор C# сгенерирует код, в котором старший разряд целого числа используется в качестве *флага знака*. Число считается положительным, если флаг знака равен 0, и отрицательным, если он равен 1. Отрицательные числа практически всегда представляются методом дополнения до двух, в соответствии с которым все двоичные разряды отрицательного числа сначала инвертируются, а затем к этому числу добавляется 1.

Целочисленные типы со знаком имеют большое значение для очень многих алгоритмов, но по абсолютной величине они наполовину меньше своих аналогов без знака. Вот как, например, выглядит число 32 767 типа `short` в двоичном представлении.

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Если установить старший разряд этого числа равным 1, чтобы получить значение со знаком, то оно будет интерпретировано как –1, принимая во внимание формат дополнения до двух. Но если объявить его как значение типа `ushort`, то после установки в 1 старшего разряда оно станет равным 65 535.

Вероятно, самым распространенным в программировании целочисленным типом является тип `int`. Переменные типа `int` нередко используются для управления циклами, индексирования массивов и математических расчетов общего назначения. Когда же требуется целочисленное значение с большим диапазоном представления чисел, чем у типа `int`, то для этой цели имеется целый ряд других целочисленных типов. Так, если значение нужно сохранить без знака, то для него можно выбрать тип `uint`, для больших значений со знаком — тип `long`, а для больших значений без знака — тип `ulong`. В качестве примера ниже приведена программа, вычисляющая расстояние от Земли до Солнца в дюймах. Для хранения столь большого значения в ней используется переменная типа `long`.

```
// Вычислить расстояние от Земли до Солнца в дюймах.

using System;

class Inches {
  static void Main() {
    long inches;
    long miles;

    miles = 93000000; // 93 000 000 миль до Солнца

    // 5 280 футов в миле, 12 дюймов в футе.
    inches = miles * 5280 * 12;

    Console.WriteLine("Расстояние до Солнца: " +
                      inches + " дюймов.");

  }
}
```

Вот как выглядит результат выполнения этой программы.

```
Расстояние до Солнца: 5892480000000 дюймов.
```

Очевидно, что этот результат нельзя было бы сохранить в переменной типа `int` или `uint`.

Самыми мелкими целочисленными типами являются `byte` и `sbyte`. Тип `byte` представляет целые значения без знака в пределах от 0 до 255. Переменные типа `byte` особенно удобны для обработки исходных двоичных данных, например байтового потока, поступающего от некоторого устройства. А для представления мелких целых значений со знаком служит тип `sbyte`. Ниже приведен пример программы, в которой переменная типа `byte` используется для управления циклом, где суммируются числа от 1 до 100.

```
// Использовать тип byte.

using System;

class Use_byte {
  static void Main() {
    byte x;
    int sum;

    sum = 0;
```

```
    for(x = 1; x <= 100; x++)
        sum = sum + x;

        Console.WriteLine("Сумма чисел от 1 до 100 равна " + sum);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Сумма чисел от 1 до 100 равна 5050
```

В приведенном выше примере программы цикл выполняется только от 1 до 100, что не превышает диапазон представления чисел для типа `byte`, и поэтому для управления этим циклом не требуется переменная более крупного типа.

Если же требуется целое значение, большее, чем значение типа `byte` или `sbyte`, но меньшее, чем значение типа `int` или `uint`, то для него можно выбрать тип `short` или `ushort`.

Типы для представления чисел с плавающей точкой

Типы с плавающей точкой позволяют представлять числа с дробной частью. В C# имеются две разновидности типов данных с плавающей точкой: `float` и `double`. Они представляют числовые значения с одинарной и двойной точностью соответственно. Так, разрядность типа `float` составляет 32 бита, что приближенно соответствует диапазону представления чисел от 5E–45 до 3,4E+38. А разрядность типа `double` составляет 64 бита, что приближенно соответствует диапазону представления чисел от 5E–324 до 1,7E+308.

В программировании на C# чаще применяется тип `double`, в частности, потому, что во многих математических функциях из библиотеки классов C#, которая одновременно является библиотекой классов для среды .NET Framework, используются числовые значения типа `double`. Например, метод `Sqrt()`, определенный в библиотеке классов `System.Math`, возвращает значение типа `double`, которое представляет собой квадратный корень из аргумента типа `double`, передаваемого данному методу. В приведенном ниже примере программы метод `Sqrt()` используется для вычисления радиуса окружности по площади круга.

```
// Определить радиус окружности по площади круга.

using System;

class FindRadius {
  static void Main() {
    Double r;
    Double area;

    area = 10.0;

    r = Math.Sqrt(area / 3.1416);

    Console.WriteLine("Радиус равен " + r);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Радиус равен 1.78412203012729
```

В приведенном выше примере программы следует обратить внимание на вызов метода `Sqrt()`. Как упоминалось выше, метод `Sqrt()` относится к классу `Math`, поэтому в его вызове имя `Math` предшествует имени самого метода. Аналогичным образом имя класса `Console` предшествует имени метода `WriteLine()` в его вызове. При вызове некоторых, хотя и не всех, стандартных методов обычно указывается имя их класса, как показано в следующем примере.

В следующем примере программы демонстрируется применение нескольких тригонометрических функций, которые относятся к классу `Math` и входят в стандартную библиотеку классов C#. Они также оперируют данными типа `double`. В этом примере на экран выводятся значения синуса, косинуса и тангенса угла, измеряемого в пределах от 0,1 до 1,0 радиана.

```
// Продемонстрировать применение тригонометрических функций.

using System;

class Trigonometry {
  static void Main() {
    Double theta; // угол в радианах

    for(theta = 0.1; theta <= 1.0;
                     theta = theta + 0.1) {
      Console.WriteLine("Синус угла " + theta +
                        " равен " + Math.Sin(theta));
      Console.WriteLine("Косинус угла " + theta +
                        "  равен " + Math.Cos(theta));
      Console.WriteLine("Тангенс угла " + theta +
                        "  равен " + Math.Tan(theta));
      Console.WriteLine();
    }
  }
}
```

Ниже приведена лишь часть результата выполнения данной программы.

```
Синус угла 0.1  равен 0.0998334166468282
Косинус угла 0.1  равен 0.995004165278026
Тангенс угла 0.1  равен 0.100334672085451

Синус угла 0.2  равен 0.198669330795061
Косинус угла 0.2  равен 0.980066577841242
Тангенс угла 0.2  равен 0.202710035508673

Синус угла 0.3  равен 0.29552020666134
Косинус угла 0.3  равен 0.955336489125606
Тангенс угла 0.3  равен 0.309336249609623
```

Для вычисления синуса, косинуса и тангенса угла в приведенном выше примере были использованы стандартные методы `Math.Sin()`, `Math.Cos()` и `Math.Tan()`. Как и метод `Math.Sqrt()`, эти тригонометрические методы вызываются с аргументом типа `double` и возвращают результат того же типа. Вычисляемые углы должны быть указаны в радианах.

Десятичный тип данных

Вероятно, самым интересным среди всех числовых типов данных в C# является тип `decimal`, который предназначен для применения в финансовых расчетах. Этот тип имеет разрядность 128 бит для представления числовых значений в пределах от 1E–28 до 7,9E+28. Вам, вероятно, известно, что для обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа `decimal`, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов. Благодаря тому что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами.

Ниже приведен пример программы, в которой тип `decimal` используется в конкретном финансовом расчете. В этой программе цена со скидкой рассчитывается на основании исходной цены и скидки в процентах.

```
// Использовать тип decimal для расчета скидки.

using System;

class UseDecimal {
  static void Main() {
    decimal price;
    decimal discount;
    decimal discounted_price;

    // Рассчитать цену со скидкой.
    price = 19.95m;
    discount = 0.15m; // норма скидки составляет 15%

    discounted_price = price - ( price * discount);

    Console.WriteLine("Цена со скидкой: $" + discounted_price);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Цена со скидкой: $16.9575
```

Обратите внимание на то, что значения констант типа `decimal` в приведенном выше примере программы указываются с суффиксом `m`. Дело в том, что без суффикса `m` эти значения интерпретировались бы как стандартные константы с плавающей точкой, которые несовместимы с типом данных `decimal`. Тем не менее переменной типа `decimal` можно присвоить целое значение без суффикса `m`, например 10. (Подробнее о числовых константах речь пойдет далее в этой главе.)

Рассмотрим еще один пример применения типа `decimal`. В этом примере рассчитывается будущая стоимость капиталовложений с фиксированной нормой прибыли в течение ряда лет.

```
/*
  Применить тип decimal для расчета будущей стоимости
  капиталовложений.
*/
```

```
using System;

class FutVal {
  static void Main() {
    decimal amount;
    decimal rate_of_return;
    int years, i;

    amount = 1000.0M;
    rate_of_return = 0.07M;
    years = 10;

    Console.WriteLine("Первоначальные капиталовложения: $" + amount);
    Console.WriteLine("Норма прибыли: " + rate_of_return);
    Console.WriteLine("В течение " + years + " лет");

    for(i = 0; i < years; i++)
        amount = amount + (amount * rate_of_return);

    Console.WriteLine("Будущая стоимость равна $" + amount);
  }
}
```

Вот как выглядит результат выполнения этой программы.

```
Первоначальные капиталовложения: $1000
Норма прибыли: 0.07
В течение 10 лет
Будущая стоимость равна $1967.151357289565322490000
```

Обратите внимание на то, что результат выполнения приведенной выше программы представлен с точностью до целого ряда десятичных разрядов, т.е. с явным избытком по сравнению с тем, что обычно требуется! Далее в этой главе будет показано, как подобный результат приводится к более "привлекательному" виду.

Символы

В C# символы представлены не 8-разрядным кодом, как во многих других языках программирования, например C++, а 16-разрядным кодом, который называется *уникодом* (Unicode). В уникоде набор символов представлен настолько широко, что он охватывает символы практически из всех естественных языков на свете. Если для многих естественных языков, в том числе английского, французского и немецкого, характерны относительно небольшие алфавиты, то в ряде других языков, например китайском, употребляются довольно обширные наборы символов, которые нельзя представить 8-разрядным кодом. Для преодоления этого ограничения в C# определен тип `char`, представляющий 16-разрядные значения без знака в пределах от 0 до 65 535. При этом стандартный набор символов в 8-разрядном коде ASCII является подмножеством уникода в пределах от 0 до 127. Следовательно, символы в коде ASCII по-прежнему остаются действительными в C#.

Для того чтобы присвоить значение символьной переменной, достаточно заключить это значение (т.е. символ) в одинарные кавычки. Так, в приведенном ниже фрагменте кода переменной `ch` присваивается символ `X`.

```
char ch;
ch = 'X';
```

Значение типа `char` можно вывести на экран с помощью метода `WriteLine()`. Например, в следующей строке кода на экран выводится значение переменной `ch`.

```
Console.WriteLine("Значение ch равно: " + ch);
```

Несмотря на то что тип `char` определен в C# как целочисленный, его не следует путать со всеми остальными целочисленными типами. Дело в том, что в C# отсутствует автоматическое преобразование символьных значений в целочисленные и обратно. Например, следующий фрагмент кода содержит ошибку.

```
char ch;
ch = 88; // ошибка, не выйдет
```

Ошибочность приведенного выше фрагмента кода объясняется тем, что `88` — это целое значение, которое не преобразуется автоматически в символьное. При попытке скомпилировать данный фрагмент кода будет выдано соответствующее сообщение об ошибке. Для того чтобы операция присваивания целого значения символьной переменной оказалась допустимой, необходимо осуществить приведение типа, о котором речь пойдет далее в этой главе.

Логический тип данных

Тип `bool` представляет два логических значения: "истина" и "ложь". Эти логические значения обозначаются в C# зарезервированными словами `true` и `false` соответственно. Следовательно, переменная или выражение типа `bool` будет принимать одно из этих логических значений. Кроме того, в C# не определено взаимное преобразование логических и целых значений. Например, 1 не преобразуется в значение `true`, а 0 — в значение `false`.

В приведенном ниже примере программы демонстрируется применение типа `bool`.

```
// Продемонстрировать применение типа bool.

using System;

class BoolDemo {
  static void Main() {
    bool b;
    b = false;

    Console.WriteLine("b равно " + b);
    b = true;
    Console.WriteLine("b равно " + b);

    // Логическое значение может управлять оператором if.
    if(b) Console.WriteLine("Выполняется.");

    b = false;
    if(b) Console.WriteLine("Не выполняется.");
```

```
    // Результатом выполнения оператора отношения
    // является логическое значение.
    Console.WriteLine("10 > 9 равно " + (10 > 9));
  }
}
```

Эта программа дает следующий результат.

```
b равно False
b равно True
Выполняется.
10 > 9 равно True
```

В приведенной выше программе обнаруживаются три интересные особенности. Во-первых, при выводе логического значения типа `bool` с помощью метода `WriteLine()` на экране появляется значение "True" или "False". Во-вторых, самого значения переменной типа `bool` достаточно для управления оператором `if`. Для этого не нужно, например, записывать оператор `if` следующим образом.

```
if(b == true) ...
```

И в-третьих, результатом выполнения оператора отношения является логическое значение. Именно поэтому в результате вычисления выражения `10 > 9` на экран выводится значение "True." Кроме того, выражение `10 > 9` следует заключить в скобки, поскольку оператор + имеет более высокий приоритет, чем оператор >.

Некоторые возможности вывода

До сих пор при выводе с помощью метода `WriteLine()` данные отображались в формате, используемом по умолчанию. Но в среде .NET Framework определен достаточно развитый механизм форматирования, позволяющий во всех деталях управлять выводом данных. Форматированный ввод-вывод подробнее рассматривается далее в этой книге, а до тех пор полезно ознакомиться с некоторыми возможностями форматирования. Они позволяют указать, в каком именно виде следует выводить значения с помощью метода `WriteLine()`. Благодаря этому выводимый результат выглядит более привлекательно. Следует, однако, иметь в виду, что механизм форматирования поддерживает намного больше возможностей, а не только те, которые рассматриваются в этом разделе.

При выводе списков данных в предыдущих примерах программ каждый элемент списка приходилось отделять знаком +, как в следующей строке.

```
Console.WriteLine("Вы заказали " + 2 +
                  " предмета по цене $" + 3 + " каждый.");
```

Конечно, такой способ вывода числовой информации удобен, но он не позволяет управлять внешним видом выводимой информации. Например, при выводе значения с плавающей точкой нельзя определить количество отображаемых десятичных разрядов. Рассмотрим оператор

```
Console.WriteLine("Деление 10/3 дает: " + 10.0/3.0);
```

который выводит следующий результат.

```
Деление 10/3 дает: 3.33333333333333
```

В одних случаях такого вывода может оказаться достаточно, а в других — он просто недопустим. Например, в финансовых расчетах после десятичной точки принято указывать лишь два десятичных разряда.

Для управления форматированием числовых данных служит другая форма метода `WriteLine()`, позволяющая встраивать информацию форматирования, как показано ниже.

```
WriteLine("форматирующая строка", arg0, arg1, ... , argN);
```

В этой форме аргументы метода `WriteLine()` разделяются запятой, а не знаком +. А *форматирующая строка* состоит из двух элементов: обычных печатаемых символов, предназначенных для вывода в исходном виде, а также спецификаторов формата. Последние указываются в следующей общей форме:

```
{argnum, width: fmt}
```

где *argnum* — номер выводимого аргумента, начиная с нуля; *width* — минимальная ширина поля; *fmt* — формат. Параметры *width* и *fmt* являются необязательными.

Если во время выполнения в форматирующей строке встречается спецификатор формата, то вместо него подставляется и отображается соответствующий аргумент, обозначаемый параметром *argnum*. Таким образом, местоположение спецификатора формата в форматирующей строке определяет место отображения соответствующих данных. Параметры *width* и *fmt* указывать необязательно. Это означает, что в своей простейшей форме спецификатор формата обозначает конкретный отображаемый аргумент. Например, спецификатор `{0}` обозначает аргумент *arg0*, спецификатор `{1}` — аргумент *arg1* и т.д.

Начнем с самого простого примера. При выполнение оператора

```
Console.WriteLine("В феврале {0} или {1} дней.", 28, 29);
```

получается следующий результат.

```
В феврале 28 или 29 дней
```

Как видите, значение `28` подставляется вместо спецификатора `{0}`, а значение `29` — вместо спецификатора `{1}`. Следовательно, спецификаторы формата обозначают место в строке, где отображаются соответствующие аргументы (в данном случае — значения `28` и `29`). Кроме того, обратите внимание на то, что дополнительные значения разделяются запятой, а не знаком +.

Ниже приведен видоизмененный вариант предыдущего оператора, в котором указывается ширина полей.

```
Console.WriteLine("В феврале {0,10} или {1,5} дней.", 28, 29);
```

Выполнение этого оператора дает следующий результат.

```
В феврале         28 или    29 дней.
```

Как видите, неиспользуемые части полей заполнены пробелами. Напомним, что *минимальная* ширина поля определяется параметром *width*. Если требуется, она может быть превышена при выводе результата.

Разумеется, аргументы, связанные с командой форматирования, не обязательно должны быть константами. Ниже приведен пример программы, которая выводит таблицу результатов возведения чисел в квадрат и куб. В ней команды форматирования используются для вывода соответствующих значений.

```
// Применить команды форматирования.

using System;

class DisplayOptions {
  static void Main() {
    int i;

    Console.WriteLine("Число\tКвадрат\tКуб");

    for(i = 1; i < 10; i++)
      Console.WriteLine("{0}\t{1}\t{2}", i, i*i, i*i*i);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Число   Квадрат Куб
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
```

В приведенных выше примерах сами выводимые значения не форматировались. Но ведь основное назначение спецификаторов формата — управлять внешним видом выводимых данных. Чаще всего форматированию подлежат следующие типы данных: с плавающей точкой и десятичный. Самый простой способ указать формат данных — описать шаблон, который будет использоваться в методе `WriteLine()`. Для этого указывается образец требуемого формата с помощью символов #, обозначающих разряды чисел. Кроме того, можно указать десятичную точку и запятые, разделяющие цифры. Ниже приведен пример более подходящего вывода результата деления 10 на 3.

```
Console.WriteLine("Деление 10/3 дает: {0:#.##}", 10.0/3.0);
```

Выполнение этого оператора приводит к следующему результату.

```
Деление 10/3 дает: 3.33
```

В данном примере шаблон `#.##` указывает методу `WriteLine()` отобразить два десятичных разряда в дробной части числа. Следует, однако, иметь в виду, что метод `WriteLine()` может отобразить столько цифр слева от десятичной точки, сколько потребуется для правильной интерпретации выводимого значения.

Рассмотрим еще один пример. Оператор

```
Console.WriteLine("{0:###,###.##}", 123456.56);
```

дает следующий результат.

```
123,456.56
```

Для вывода денежных сумм, например, рекомендуется использовать спецификатор формата `C`.

```
decimal balance;

balance = 12323.09m;
Console.WriteLine("Текущий баланс равен {0:C}", balance);
```

Результат выполнения этого фрагмента кода выводится в формате денежных сумм, указываемых в долларах США.

```
Текущий баланс равен $12,323.09
```

Форматом C можно также воспользоваться, чтобы представить в более подходящем виде результат выполнения рассматривавшейся ранее программы расчета цены со скидкой.

```
// Использовать спецификатор формата C для вывода
// результата в местной валюте.

using System;

class UseDecimal {
  static void Main() {
    decimal price;
    decimal discount;
    decimal discounted_price;

    // рассчитать цену со скидкой.
    price = 19.95m;
    discount = 0.15m; // норма скидки составляет 15%

    discounted_price = price - ( price * discount);

    Console.WriteLine("Цена со скидкой: {0:C}", discounted_price);
  }
}
```

Вот как теперь выглядит результат выполнения этой программы.

```
Цена со скидкой: 16,96 грн.
```

Литералы

В C# *литералами* называются постоянные значения, представленные в удобной для восприятия форме. Например, число 100 является литералом. Сами литералы и их назначение настолько понятны, что они применялись во всех предыдущих примерах программ без всяких пояснений. Но теперь настало время дать им формальное объяснение.

В C# литералы могут быть любого простого типа. Представление каждого литерала зависит от конкретного типа. Как пояснялось ранее, символьные литералы заключаются в одинарные кавычки. Например, `'a'` и `'%'` являются символьными литералами.

Целочисленные литералы указываются в виде чисел без дробной части. Например, `10` и `-100` — это целочисленные литералы. Для обозначения литералов с плавающей точкой требуется указывать десятичную точку и дробную часть числа. Например, `11.123` — это литерал с плавающей точкой. Для вещественных чисел с плавающей точкой в C# допускается также использовать экспоненциальное представление.

У литералов должен быть также конкретный тип, поскольку C# является строго типизированным языком. В этой связи возникает естественный вопрос: к какому типу следует отнести числовой литерал, например `2,123987` или `0.23`? К счастью, для ответа на этот вопрос в C# установлен ряд простых для соблюдения правил.

Во-первых, у целочисленных литералов должен быть самый мелкий целочисленный тип, которым они могут быть представлены, начиная с типа `int`. Таким образом, у целочисленных литералов может быть один из следующих типов: `int`, `uint`, `long` или `ulong` в зависимости от значения литерала. И во-вторых, литералы с плавающей точкой относятся к типу `double`.

Если вас не устраивает используемый по умолчанию тип литерала, вы можете явно указать другой его тип с помощью суффикса. Так, для указания типа `long` к литералу присоединяется суффикс `l` или `L`. Например, `12` — это литерал типа `int`, а `12L` — литерал типа `long`. Для указания целочисленного типа без знака к литералу присоединяется суффикс `u` или `U`. Следовательно, `100` — это литерал типа `int`, а `100U` — литерал типа `uint`. А для указания длинного целочисленного типа без знака к литералу присоединяется суффикс `ul` или `UL`. Например, `984375UL` — это литерал типа `ulong`.

Кроме того, для указания типа `float` к литералу присоединяется суффикс `F` или `f`. Например, `10.19F` — это литерал типа `float`. Можете даже указать тип `double`, присоединив к литералу суффикс `d` или `D`, хотя это излишне. Ведь, как упоминалось выше, по умолчанию литералы с плавающей точкой относятся к типу `double`.

И наконец, для указания типа `decimal` к литералу присоединяется суффикс `m` или `M`. Например, `9.95M` — это десятичный литерал типа `decimal`.

Несмотря на то что целочисленные литералы образуют по умолчанию значения типа `int`, `uint`, `long` или `ulong`, их можно присваивать переменным типа `byte`, `sbyte`, `short` или `ushort`, при условии, что присваиваемое значение может быть представлено целевым типом.

Шестнадцатеричные литералы

Вам, вероятно, известно, что в программировании иногда оказывается проще пользоваться системой счисления по основанию 16, чем по основанию 10. Система счисления по основанию 16 называется *шестнадцатеричной*. В ней используются числа от 0 до 9, а также буквы от A до F, которыми обозначаются десятичные числа 10, 11, 12, 13, 14 и 15. Например, десятичному числу 16 соответствует шестнадцатеричное число 10. Вследствие того что шестнадцатеричные числа применяются в программировании довольно часто, в C# разрешается указывать целочисленные литералы в шестнадцатеричном формате. Шестнадцатеричные литералы должны начинаться с символов `0x`, т.е. нуля и последующей латинской буквы "икс". Ниже приведены некоторые примеры шестнадцатеричных литералов.

```
count = 0xFF;  // 255 в десятичной системе
incr = 0x1a;   // 26 в десятичной системе
```

Управляющие последовательности символов

Большинство печатаемых символов достаточно заключить в одинарные кавычки, но набор в текстовом редакторе некоторых символов, например возврата каретки, вызывает особые трудности. Кроме того, ряд других символов, в том числе одинарные и двойные кавычки, имеют специальное назначение в C#, поэтому их нельзя исполь-

зовать непосредственно. По этим причинам в C# предусмотрены специальные *управляющие последовательности символов*, иногда еще называемые *константами с обратной косой чертой* (табл. 3.2). Такие последовательности применяются вместо тех символов, которых они представляют.

Таблица 3.2. Управляющие последовательности символов

Управляющая последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Пустой символ
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Например, в следующей строке кода переменной ch присваивается символ табуляции.

```
ch = '\t';
```

А в приведенном ниже примере кода переменной ch присваивается символ одинарной кавычки.

```
ch = '\'';
```

Строковые литералы

В C# поддерживается еще один тип литералов — *строковый*. Строковый литерал представляет собой набор символов, заключенных в двойные кавычки. Например следующий фрагмент кода:

```
"это тест"
```

представляет собой текстовую строку. Образцы подобных строк не раз встречались в приведенных выше примерах программ.

Помимо обычных символов, строковый литерал может содержать одну или несколько управляющих последовательностей символов, о которых речь шла выше. Рассмотрим для примера программу, в которой используются управляющие последовательности \n и \t.

```
// Продемонстрировать применение управляющих
// последовательностей символов в строковых литералах.

using System;

class StrDemo {
  static void Main() {
```

```
    Console.WriteLine("Первая строка\nВторая строка\nТретья строка");
    Console.WriteLine("Один\tДва\tТри");
    Console.WriteLine("Четыре\tПять\tШесть");

    // Вставить кавычки.
    Console.WriteLine("\"Зачем?\", спросил он.");
  }
}
```

Результат выполнения этой программы приведен ниже.

```
Первая  строка
Вторая  строка
Третья  строка
Один    Два     Три
Четыре  Пять    Шесть
"Зачем?", спросил он.
```

В приведенном выше примере программы обратите внимание на то, что для перехода на новую строку используется управляющая последовательность `\n`. Для вывода нескольких строк совсем не обязательно вызывать метод `WriteLine()` несколько раз — достаточно вставить управляющую последовательность `\n` в тех местах удлиненной текстовой строки (или строкового литерала), где должен происходить переход на новую строку. Обратите также внимание на то, как в текстовой строке формируется знак кавычек.

Помимо описанной выше формы строкового литерала, можно также указать *буквальный строковый литерал*. Такой литерал начинается с символа `@`, после которого следует строка в кавычках. Содержимое строки в кавычках воспринимается без изменений и может быть расширено до двух и более строк. Это означает, что в буквальный строковый литерал можно включить символы новой строки, табуляции и прочие, не прибегая к управляющим последовательностям. Единственное исключение составляют двойные кавычки ("), для указания которых необходимо использовать две двойные кавычки подряд (""). В приведенном ниже примере программы демонстрируется применение буквальных строковых литералов.

```
// Продемонстрировать применение буквальных строковых литералов.

using System;

class Verbatim {
  static void Main() {
    Console.WriteLine(@"Это буквальный
строковый литерал,
занимающий несколько строк.
");
    Console.WriteLine(@"А это вывод с табуляцией:
1  2   3      4
5  6   7      8
");
    Console.WriteLine(@"Отзыв программиста: ""Мне нравится C#.""");
  }
}
```

Результат выполнения этой программы приведен ниже.

```
Это буквальный
строковый литерал,
занимающий несколько строк.

А это вывод с табуляцией:
1       2       3       4
5       6       7       8

Отзыв программиста: "Мне нравится C#."
```

Следует особо подчеркнуть, что буквальные строковые литералы выводятся в том же виде, в каком они введены в исходном тексте программы.

Преимущество буквальных строковых литералов заключается в том, что они позволяют указать в программе выводимый результат именно так, как он должен выглядеть на экране. Но если выводится несколько строк, то переход на новую строку может нарушить порядок набора исходного текста программы с отступами. Именно по этой причине в примерах программ, приведенных в этой книге, применение буквальных строковых литералов ограничено. Тем не менее они приносят немало замечательных выгод во многих случаях, когда требуется форматирование выводимых результатов.

И последнее замечание: не путайте строки с символами. Символьный литерал, например `'X'`, обозначает одиночную букву типа `char`. А строка, состоящая из одного символа, например `"X"`, по-прежнему остается текстовой строкой.

Более подробное рассмотрение переменных

Переменные объявляются с помощью оператора следующей формы:

тип имя_переменной;

где *тип* — это тип данных, хранящихся в переменной; а *имя_переменной* — это ее имя. Объявить можно переменную любого действительного типа, в том числе и описанных выше типов значений. Важно подчеркнуть, что возможности переменной определяются ее типом. Например, переменную типа `bool` нельзя использовать для хранения числовых значений с плавающей точкой. Кроме того, тип переменной нельзя изменять в течение срока ее существования. В частности, переменную типа `int` нельзя преобразовать в переменную типа `char`.

Все переменные в C# должны быть объявлены до их применения. Это нужно для того, чтобы уведомить компилятор о типе данных, хранящихся в переменной, прежде чем он попытается правильно скомпилировать любой оператор, в котором используется переменная. Это позволяет также осуществлять строгий контроль типов в C#.

В C# определено несколько различных видов переменных. Так, в предыдущих примерах программ использовались переменные, называемые *локальными*, поскольку они объявляются внутри метода.

Инициализация переменной

Задать значение переменной можно, в частности, с помощью оператора присваивания, как было не раз продемонстрировано ранее. Кроме того, задать начальное значение переменной можно при ее объявлении. Для этого после имени переменной указывается знак равенства (=) и присваиваемое значение. Ниже приведена общая форма инициализации переменной:

```
тип имя_переменной = значение;
```

где *значение* — это конкретное значение, задаваемое при создании переменной. Оно должно соответствовать указанному типу переменной.

Ниже приведены некоторые примеры инициализации переменных.

```
int count = 10;  // задать начальное значение 10 переменной count.
char ch = 'X';   // инициализировать переменную ch буквенным значением X.
float f = 1.2F   // переменная f инициализируется числовым значением 1,2.
```

Если две или более переменные одного и того же типа объявляются списком, разделяемым запятыми, то этим переменным можно задать, например, начальное значение.

```
int a, b = 8, c = 19, d; // инициализировать переменные b и c
```

В данном примере инициализируются только переменные `b` и `c`.

Динамическая инициализация

В приведенных выше примерах в качестве инициализаторов переменных использовались только константы, но в C# допускается также динамическая инициализация переменных с помощью любого выражения, действительного на момент объявления переменной. Ниже приведен пример краткой программы для вычисления гипотенузы прямоугольного треугольника по длине его противоположных сторон.

```
// Продемонстрировать динамическую инициализацию.

using System;

class DynInit {
  static void Main() {
    // Длина сторон прямоугольного треугольника.
    double s1 = 4.0;
    double s2 = 5.0;

    // Инициализировать переменную hypot динамически.
    double hypot = Math.Sqrt( (s1 * s1) + (s2 * s2) );

    Console.Write("Гипотенуза треугольника со сторонами " +
                  s1 + " и " + s2 + " равна ");

    Console.WriteLine("{0:#.###}.", hypot);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Гипотенуза треугольника со сторонами 4 и 5 равна 6.403
```

В данном примере объявляются три локальные переменные: `s1`, `s2` и `hypot`. Две из них (`s1` и `s2`) инициализируются константами, А третья (`hypot`) динамически инициализируется вычисляемой длиной гипотенузы. Для такой инициализации используется выражение, указываемое в вызываемом методе `Math.Sqrt()`. Как пояснялось выше, для динамической инициализации пригодно любое выражение, действительное на момент объявления переменной. А поскольку вызов метода `Math.Sqrt()` (или любого другого библиотечного метода) является действительным на данный момент, то

его можно использовать для инициализации переменной hypot. Следует особо подчеркнуть, что в выражении для инициализации можно использовать любой элемент, действительный на момент самой инициализации переменной, в том числе вызовы методов, другие переменные или литералы.

Неявно типизированные переменные

Как пояснялось выше, все переменные в C# должны быть объявлены. Как правило, при объявлении переменной сначала указывается тип, например int или bool, а затем имя переменной. Но начиная с версии C# 3.0, компилятору предоставляется возможность самому определить тип локальной переменной, исходя из значения, которым она инициализируется. Такая переменная называется *неявно типизированной*.

Неявно типизированная переменная объявляется с помощью ключевого слова var и должна быть непременно инициализирована. Для определения типа этой переменной компилятору служит тип ее инициализатора, т.е. значения, которым она инициализируется. Рассмотрим такой пример.

```
var e = 2.7183;
```

В данном примере переменная e инициализируется литералом с плавающей точкой, который по умолчанию имеет тип double, и поэтому она относится к типу double. Если бы переменная e была объявлена следующим образом:

```
var e = 2.7183F;
```

то она была бы отнесена к типу float.

В приведенном ниже примере программы демонстрируется применение неявно типизированных переменных. Он представляет собой вариант программы из предыдущего раздела, измененной таким образом, чтобы все переменные были типизированы неявно.

```
// Продемонстрировать применение неявно типизированных переменных.

using System;

class ImplicitlyTypedVar {
  static void Main() {
    // Эти переменные типизированы неявно. Они отнесены
    // к типу double, поскольку инициализирующие их
    // выражения сами относятся к типу double.
    var s1 = 4.0;
    var s2 = 5.0;

    // Итак, переменная hypot типизирована неявно и
    // относится к типу double, поскольку результат,
    // возвращаемый методом Sqrt(), имеет тип double.
    var hypot = Math.Sqrt( (s1 * s1) + (s2 * s2) );

    Console.Write("Гипотенуза треугольника со сторонами " +
                  s1 + " by " + s2 + " равна ");

    Console.WriteLine("{0:#.###}.", hypot);

    // Следующий оператор не может быть скомпилирован,
```

```
    // поскольку переменная s1 имеет тип double и
    // ей нельзя присвоить десятичное значение.
    // s1 = 12.2M; // Ошибка!
  }
}
```

Результат выполнения этой программы оказывается таким же, как и прежде.

Важно подчеркнуть, что неявно типизированная переменная по-прежнему остается строго типизированной. Обратите внимание на следующую закомментированную строку из приведенной выше программы.

```
// s1 = 12.2M; // Ошибка!
```

Эта операция присваивания недействительна, поскольку переменная `s1` относится к типу `double`. Следовательно, ей нельзя присвоить десятичное значение. Единственное отличие неявно типизированной переменной от обычной, явно типизированной переменной, — в способе определения ее типа. Как только этот тип будет определен, он закрепляется за переменной до конца ее существования. Это, в частности, означает, что тип переменной `s1` не может быть изменен по ходу выполнения программы.

Неявно типизированные переменные внедрены в C# не для того, чтобы заменить собой обычные объявления переменных. Напротив, неявно типизированные переменные предназначены для особых случаев, и самый примечательный из них имеет отношение к языку интегрированных запросов (LINQ), подробно рассматриваемому в главе 19. Таким образом, большинство объявлений переменных должно и впредь оставаться явно типизированными, поскольку они облегчают чтение и понимание исходного текста программы.

И последнее замечание: одновременно можно объявить только одну неявно типизированную переменную. Поэтому объявление

```
var s1 = 4.0, s2 = 5.0; // Ошибка!
```

является неверным и не может быть скомпи́лировано. Ведь в нем предпринимается попытка объявить обе переменные, `s1` и `s2`, одновременно.

Область действия и время существования переменных

Все переменные, использовавшиеся в предыдущих примерах программ, объявлялись в самом начале метода `Main()`. Но в C# локальную переменную разрешается объявлять в любом кодовом блоке. Как пояснялось в главе 2, кодовый блок начинается открывающей фигурной скобкой и оканчивается закрывающей фигурной скобкой. Этот блок и определяет *область действия*. Следовательно, всякий раз, когда начинается блок, образуется новая область действия. Прежде всего область действия определяет видимость имен отдельных элементов, в том числе и переменных, в других частях программы без дополнительного уточнения. Она определяет также время существования локальных переменных.

В C# к числу наиболее важных относятся области действия, определяемые классом и методом. Рассмотрение области действия класса (и объявляемых в ней переменных) придется отложить до того момента, когда в этой книге будут описываться классы. А до тех пор будут рассматриваться только те области действия, которые определяются методом или же в самом методе.

Область действия, определяемая методом, начинается открывающей фигурной скобкой и оканчивается закрывающей фигурной скобкой. Но если у этого метода имеются параметры, то и они входят в область действия, определяемую данным методом.

Как правило, локальные переменные объявляются в области действия, невидимой для кода, находящегося вне этой области. Поэтому, объявляя переменную в определенной области действия, вы тем самым защищаете ее от доступа или видоизменения вне данной области. Разумеется, правила области действия служат основанием для инкапсуляции.

Области действия могут быть вложенными. Например, всякий раз, когда создается кодовый блок, одновременно образуется и новая, вложенная область действия. В этом случае внешняя область действия охватывает внутреннюю область. Это означает, что локальные переменные, объявленные во внешней области действия, будут видимы для кода во внутренней области действия. Но обратное не справедливо: локальные переменные, объявленные во внутренней области действия, не будут видимы вне этой области.

Для того чтобы стала более понятной сущность вложенных областей действия, рассмотрим следующий пример программы.

```
// Продемонстрировать область действия кодового блока.

using System;

class ScopeDemo {
  static void Main() {
    int x; // Эта переменная доступна для всего кода внутри метода Main().

    x = 10;
    if(x == 10) {  // начать новую область действия
      int y = 20; // Эта переменная доступна только в данном кодовом блоке.

      // Здесь доступны обе переменные, x и y.

      Console.WriteLine("x и y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Ошибка! Переменна y здесь недоступна.

    // А переменная x здесь по-прежнему доступна.
    Console.WriteLine("x равно " + x);
  }
}
```

Как поясняется в комментариях к приведенной выше программе, переменная `x` объявляется в начале области действия метода `Main()`, и поэтому она доступна для всего последующего кода в пределах этого метода. В блоке условного оператора `if` объявляется переменная `y`. А поскольку этот кодовый блок определяет свою собственную область действия, то переменная `y` видима только для кода в пределах данного блока. Именно поэтому строка `line y = 100;`, находящаяся за пределами этого блока, закомментирована. Если удалить находящиеся перед ней символы комментария (`//`), то во время компиляции программы произойдет ошибка, поскольку переменная `y` невидима за пределами своего кодового блока. В то же время переменная `x` может использоваться в блоке условного оператора `if`, поскольку коду из этого блока,

находящемуся во вложенной области действия, доступны переменные, объявленные в охватывающей его внешней области действия.

Переменные могут быть объявлены в любом месте кодового блока, но они становятся действительными только после своего объявления. Так, если объявить переменную в начале метода, то она будет доступна для всего остального кода в пределах этого метода. А если объявить переменную в конце блока, то она окажется, по существу, бесполезной, поскольку не будет доступной ни одному коду.

Если в объявление переменной включается инициализатор, то такая переменная инициализируется повторно при каждом входе в тот блок, в котором она объявлена. Рассмотрим следующий пример программы.

```
// Продемонстрировать время существования переменной.

using System;

class VarInitDemo {
  static void Main() {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // Переменная y инициализируется при каждом входе в блок.
      Console.WriteLine("y равно: " + y); // Здесь всегда выводится -1

      y = 100;
      Console.WriteLine("y теперь равно: " + y);
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
y равно: -1
y теперь равно: 100
y равно: -1
y теперь равно: 100
y равно: -1
y теперь равно: 100
```

Как видите, переменная y повторно инициализируется одним и тем же значением `-1` при каждом входе во внутренний цикл `for`. И несмотря на то, что после этого цикла ей присваивается значение `100`, оно теряется при повторной ее инициализации.

В языке C# имеется еще одна особенность соблюдения правил области действия: несмотря на то, что блоки могут быть вложены, ни у одной из переменных из внутренней области действия не должно быть такое же имя, как и у переменной из внешней области действия. В приведенном ниже примере программы предпринимается попытка объявить две разные переменные с одним и тем же именем, и поэтому программа не может быть скомпилирована.

```
/*
  В этой программе предпринимается попытка объявить во внутренней
  области действия переменную с таким же самым именем, как и у
  переменной, определенной во внешней области действия.
```

```
  *** Эта программа не может быть скомпилирована. ***
*/

using System;

class NestVar {
  static void Main() {
    int count;

    for(count = 0; count < 10; count = count+1) {
        Console.WriteLine("Это подсчет: " + count);

        int count; // Недопустимо!!!
        for(count = 0; count < 2; count++)
            Console.WriteLine("В этой программе есть ошибка!");
    }
  }
}
```

Если у вас имеется некоторый опыт программирования на C или C++, то вам должно быть известно, что на присваивание имен переменным, объявляемым во внутренней области действия, в этих языках не существует никаких ограничений. Следовательно, в C и C++ объявление переменной `count` в кодовом блоке, входящем во внешний цикл `for`, как в приведенном выше примере, считается вполне допустимым. Но в C и C++ такое объявление одновременно означает сокрытие внешней переменной. Разработчики C# посчитали, что такого рода *сокрытие имен* может легко привести к программным ошибкам, и поэтому решили запретить его.

Преобразование и приведение типов

В программировании нередко значения переменных одного типа присваиваются переменным другого типа. Например, в приведенном ниже фрагменте кода целое значение типа `int` присваивается переменной с плавающей точкой типа `float`.

```
int i;
float f;

i = 10;
f = i; // присвоить целое значение переменной типа float
```

Если в одной операции присваивания смешиваются совместимые типы данных, то значение в правой части оператора присваивания автоматически преобразуется в тип, указанный в левой его части. Поэтому в приведенном выше фрагменте кода значение переменной `i` сначала преобразуется в тип `float`, а затем присваивается переменной `f`. Но вследствие строгого контроля типов далеко не все типы данных в C# оказываются полностью совместимыми, а следовательно, не все преобразования типов разрешены в неявном виде. Например, типы `bool` и `int` несовместимы. Правда, преобразование несовместимых типов все-таки может быть осуществлено путем *приведения*. Приведение типов, по существу, означает явное их преобразование. В этом разделе рассматривается как автоматическое преобразование, так и приведение типов.

Автоматическое преобразование типов

Когда данные одного типа присваиваются переменной другого типа, *неявное* преобразование типов происходит автоматически при следующих условиях:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Если оба эти условия удовлетворяются, то происходит *расширяющее преобразование*. Например, тип `int` достаточно крупный, чтобы вмещать в себя все действительные значения типа `byte`, а кроме того, оба типа, `int` и `byte`, являются совместимыми целочисленными типами, и поэтому для них вполне возможно неявное преобразование.

Числовые типы, как целочисленные, так и с плавающей точкой, вполне совместимы друг с другом для выполнения расширяющих преобразований. Так, приведенная ниже программа составлена совершенно правильно, поскольку преобразование типа `long` в тип `double` является расширяющим и выполняется автоматически.

```
// Продемонстрировать неявное преобразование типа long в тип double.

using System;

class LtoD {
  static void Main() {
    long L;
    double D;

    L = 100123285L;
    D = L;
    Console.WriteLine("L и D: " + L + " " + D);
  }
}
```

Если тип `long` может быть преобразован в тип `double` неявно, то обратное преобразование типа `double` в тип `long` неявным образом невозможно, поскольку оно не является расширяющим. Следовательно, приведенный ниже вариант предыдущей программы составлен неправильно.

```
// *** Эта программа не может быть скомпилирована. ***

using System;

class LtoD {
  static void Main() {
    long L;
    double D;

    D = 100123285.0;
    L = D; // Недопустимо!!!

    Console.WriteLine("L и D: " + L + " " + D);
  }
}
```

Помимо упомянутых выше ограничений, не допускается неявное взаимное преобразование типов `decimal` и `float` или `double`, а также числовых типов и `char` или `bool`. Кроме того, типы `char` и `bool` несовместимы друг с другом.

Приведение несовместимых типов

Несмотря на всю полезность неявных преобразований типов, они неспособны удовлетворить все потребности в программировании, поскольку допускают лишь расширяющие преобразования совместимых типов. А во всех остальных случаях приходится обращаться к приведению типов. *Приведение* — это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого требуется явное преобразование типов. Ниже приведена общая форма приведения типов.

```
(целевой_тип) выражение
```

Здесь *целевой_тип* обозначает тот тип, в который желательно преобразовать указанное *выражение*. Рассмотрим для примера следующее объявление переменных.

```
double x, y;
```

Если результат вычисления выражения `x/y` должен быть типа `int`, то следует записать следующее.

```
(int) (x / y)
```

Несмотря на то что переменные `x` и `y` относятся к типу `double`, результат вычисления выражения `x/y` преобразуется в тип `int` благодаря приведению. В данном примере выражение `x/y` следует непременно указывать в скобках, иначе приведение к типу `int` будет распространяться только на переменную `x`, а не на результат ее деления на переменную `y`. Приведение типов в данном случае требуется потому, что неявное преобразование типа `double` в тип `int` невозможно.

Если приведение типов приводит к *сужающему преобразованию*, то часть информации может быть потеряна. Например, в результате приведения типа `long` к типу `int` часть информации потеряется, если значение типа `long` окажется больше диапазона представления чисел для типа `int`, поскольку старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному, то в результате усечения теряется дробная часть этого числового значения. Так, если присвоить значение 1,23 целочисленной переменной, то в результате в ней останется лишь целая часть исходного числа (1), а дробная его часть (0,23) будет потеряна.

В следующем примере программы демонстрируется ряд преобразований типов, требующих приведения. В этом примере показан также ряд ситуаций, в которых приведение типов становится причиной потери данных.

```
// Продемонстрировать приведение типов.

using System;

class CastDemo {
  static void Main() {
    double x, y;
    byte b;
    int i;
    char ch;
    uint u;
    short s;
    long l;

    x = 10.0;
```

```
    y = 3.0;

    // Приведение типа double к типу int, дробная часть числа теряется.
    i = (int) (x / y);
    Console.WriteLine("Целочисленный результат деления x / y: " + i);
    Console.WriteLine();

    // Приведение типа int к типу byte без потери данных.
    i = 255;
    b = (byte) i;
    Console.WriteLine("b после присваивания 255: " + b +
                      " -- без потери данных.");

    // Приведение типа int к типу byte с потерей данных.
    i = 257;
    b = (byte) i;
    Console.WriteLine("b после присваивания 257: " + b +
                      " -- с потерей данных.");
    Console.WriteLine();

    // Приведение типа uint к типу short без потери данных.
    u = 32000;
    s = (short) u;
    Console.WriteLine("s после присваивания 32000: " +
                      s + " -- без потери данных.");
    // Приведение типа uint к типу short с потерей данных.
    u = 64000;
    s = (short) u;
    Console.WriteLine("s после присваивания 64000: " +
                      s + " -- с потерей данных.");
    Console.WriteLine();

    // Приведение типа long к типу uint без потери данных.
    l = 64000;
    u = (uint) l;
    Console.WriteLine("u после присваивания 64000: " + u +
                      " -- без потери данных.");

    // Приведение типа long к типу uint с потерей данных.
    l = -12;
    u = (uint) l;
    Console.WriteLine("u после присваивания -12: " + u +
                      " -- с потерей данных.");
    Console.WriteLine();

    // Приведение типа int к типу char.
    b = 88; // код ASCII символа X
    ch = (char) b;
    Console.WriteLine("ch после присваивания 88: " + ch);
  }
}
```

Вот какой результат дает выполнение этой программы.

```
Целочисленный результат деления x / y: 3

b после присваивания 255: 255 -- без потери данных.
```

```
b после присваивания 257: 1 -- с потерей данных.

s после присваивания 32000: 32000 -- без потери данных.
s после присваивания 64000: -1536 -- с потерей данных.

u после присваивания 64000: 64000 -- без потери данных.
u после присваивания -12: 4294967284 -- с потерей данных.

ch после присваивания 88: X
```

Рассмотрим каждую операцию присваивания в представленном выше примере программы по отдельности. Вследствие приведения результата деления `x/y` к типу `int` отбрасывается дробная часть числа, а следовательно, теряется часть информации.

Когда переменной `b` присваивается значение 255, то информация не теряется, поскольку это значение входит в диапазон представления чисел для типа `byte`. Но когда переменной `b` присваивается значение 257, то часть информации теряется, поскольку это значение превышает диапазон представления чисел для типа `byte`. Приведение типов требуется в обоих случаях, поскольку неявное преобразование типа `int` в тип `byte` невозможно.

Когда переменной `s` типа `short` присваивается значение 32 000 переменной `u` типа `uint`, потери данных не происходит, поскольку это значение входит в диапазон представления чисел для типа `short`. Но в следующей операции присваивания переменная `u` имеет значение 64 000, которое оказывается вне диапазона представления чисел для типа `short`, и поэтому данные теряются. Приведение типов требуется в обоих случаях, поскольку неявное преобразование типа `uint` в тип `short` невозможно.

Далее переменной `u` присваивается значение 64 000 переменной `l` типа `long`. В этом случае данные не теряются, поскольку значение 64 000 оказывается вне диапазона представления чисел для типа `uint`. Но когда переменной `u` присваивается значение -12, данные теряются, поскольку отрицательные числа также оказываются вне диапазона представления чисел для типа `uint`. Приведение типов требуется в обоих случаях, так как неявное преобразование типа `long` в тип `uint` невозможно.

И наконец, когда переменной `char` присваивается значение типа `byte`, информация не теряется, но приведение типов все же требуется.

Преобразование типов в выражениях

Помимо операций присваивания, преобразование типов происходит и в самих выражениях. В выражении можно свободно смешивать два или более типа данных, при условии их совместимости друг с другом. Например, в одном выражении допускается применение типов `short` и `long`, поскольку оба типа являются числовыми. Когда в выражении смешиваются разные типы данных, они преобразуются в один и тот же тип по порядку следования операций.

Преобразования типов выполняются по принятым в C# *правилам продвижения типов*. Ниже приведен алгоритм, определяемый этими правилами для операций с двумя операндами.

> ЕСЛИ один операнд имеет тип `decimal`, ТО и второй операнд продвигается к типу `decimal` (но если второй операнд имеет тип `float` или `double`, результат будет ошибочным).

ЕСЛИ один операнд имеет тип `double`, ТО и второй операнд продвигается к типу `double`.

ЕСЛИ один операнд имеет тип `float`, ТО и второй операнд продвигается к типу `float`.

ЕСЛИ один операнд имеет тип `ulong`, ТО и второй операнд продвигается к типу `ulong` (но если второй операнд имеет тип `sbyte`, `short`, `int` или `long`, результат будет ошибочным).

ЕСЛИ один операнд имеет тип `long`, ТО и второй операнд продвигается к типу `long`.

ЕСЛИ один операнд имеет тип `uint`, а второй — тип `sbyte`, `short` или `int`, ТО оба операнда продвигаются к типу `long`.

ЕСЛИ один операнд имеет тип `uint`, ТО и второй операнд продвигается к типу `uint`.

ИНАЧЕ оба операнда продвигаются к типу `int`.

Относительно правил продвижения типов необходимо сделать ряд важных замечаний. Во-первых, не все типы могут смешиваться в выражении. В частности, неявное преобразование типа `float` или `double` в тип `decimal` невозможно, как, впрочем, и смешение типа `ulong` с любым целочисленным типом со знаком. Для смешения этих типов требуется явное их приведение.

Во-вторых, особого внимания требует последнее из приведенных выше правил. Оно гласит: если ни одно из предыдущих правил не применяется, то все операнды продвигаются к типу `int`. Следовательно, все значения типа `char`, `sbyte`, `byte`, `ushort` и `short` продвигаются к типу `int` в целях вычисления выражения. Такое продвижение типов называется *целочисленным*. Это также означает, что результат выполнения всех арифметических операций будет иметь тип не ниже `int`.

Следует иметь в виду, что правила продвижения типов применяются только к значениям, которыми оперируют при вычислении выражения. Так, если значение переменной типа `byte` продвигается к типу `int` внутри выражения, то вне выражения эта переменная по-прежнему относится к типу `byte`. Продвижение типов затрагивает только вычисление выражения.

Но продвижение типов может иногда привести к неожиданным результатам. Если, например, в арифметической операции используются два значения типа `byte`, то происходит следующее. Сначала операнды типа `byte` продвигаются к типу `int`. А затем выполняется операция, дающая результат типа `int`. Следовательно, результат выполнения операции, в которой участвуют два значения типа `byte`, будет иметь тип `int`. Но ведь это не тот результат, который можно было бы с очевидностью предположить. Рассмотрим следующий пример программы.

```
// Пример неожиданного результата продвижения типов!

using System;

class PromDemo {
  static void Main() {
    byte b;

    b = 10;
```

```
    b = (byte) (b * b); // Необходимо приведение типов!!
    Console.WriteLine("b: "+ b);
  }
}
```

Как ни странно, но когда результат вычисления выражения `b*b` присваивается обратно переменной `b`, то возникает потребность в приведении к типу `byte`! Объясняется это тем, что в выражении `b*b` значение переменной `b` продвигается к типу `int` и поэтому не может быть присвоено переменной типа `byte` без приведения типов. Имейте это обстоятельство в виду, если получите неожиданное сообщение об ошибке несовместимости типов в выражениях, которые, на первый взгляд, кажутся совершенно правильными.

Аналогичная ситуация возникает при выполнении операций с символьными операндами. Например, в следующем фрагменте кода требуется обратное приведение к типу `char`, поскольку операнды `ch1` и `ch2` в выражении продвигаются к типу `int`.

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

Без приведения типов результат сложения операндов `ch1` и `ch2` будет иметь тип `int`, и поэтому его нельзя присвоить переменной типа `char`.

Продвижение типов происходит и при выполнении унарных операций, например с унарным минусом. Операнды унарных операций более мелкого типа, чем `int` (`byte`, `sbyte`, `short` и `ushort`), т.е. с более узким диапазоном представления чисел, продвигаются к типу `int`. То же самое происходит и с операндом типа `char`. Кроме того, если выполняется унарная операция отрицания значения типа `uint`, то результат продвигается к типу `long`.

Приведение типов в выражениях

Приведение типов можно применять и к отдельным частям крупного выражения. Это позволяет точнее управлять преобразованиями типов при вычислении выражения. Рассмотрим следующий пример программы, в которой выводятся квадратные корни чисел от 1 до 10 и отдельно целые и дробные части каждого числового результата. Для этого в данной программе применяется приведение типов, благодаря которому результат, возвращаемый методом `Math.Sqrt()`, преобразуется в тип `int`.

```
// Пример приведения типов в выражениях.

using System;

class CastExpr {
  static void Main() {
    double n;

    for(n = 1.0; n <= 10; n++) {
      Console.WriteLine("Квадратный корень из {0} равен {1}",
                        n, Math.Sqrt(n));

      Console.WriteLine("Целая часть числа: {0}" ,
                        (int) Math.Sqrt(n));

      Console.WriteLine("Дробная часть числа: {0}",
                        Math.Sqrt(n) - (int) Math.Sqrt(n) );
```

```
        Console.WriteLine();
    }
  }
}
```

Вот как выглядит результат выполнения этой программы.

```
Квадратный корень из 1 равен 1
Целая часть числа: 1
Дробная часть числа: 0

Квадратный корень из 2 равен 1.4142135623731
Целая часть числа: 1
Дробная часть числа: 0.414213562373095

Квадратный корень из 3 равен 1.73205080756888
Целая часть числа: 1
Дробная часть числа: 0.732050807568877

Квадратный корень из 4 равен 2
Целая часть числа: 2
Дробная часть числа: 0

Квадратный корень из 5 равен 2.23606797749979
Целая часть числа: 2
Дробная часть числа: 0.23606797749979

Квадратный корень из 6 равен 2.44948974278318
Целая часть числа: 2
Дробная часть числа: 0.449489742783178

Квадратный корень из 7 равен 2.64575131106459
Целая часть числа: 2
Дробная часть числа: 0.645751311064591

Квадратный корень из 8 равен 2.82842712474619
Целая часть числа: 2
Дробная часть числа: 0.82842712474619

Квадратный корень из 9 равен 3
Целая часть числа: 3
Дробная часть числа: 0

Квадратный корень из 10 равен 3.16227766016838
Целая часть числа: 3
Дробная часть числа: 0.16227766016838
```

Как видите, приведение результата, возвращаемого методом `Math.Sqrt()`, к типу `int` позволяет получить целую часть числа. Так, в выражении

```
Math.Sqrt(n) - (int) Math.Sqrt(n)
```

приведение к типу `int` дает целую часть числа, которая затем вычитается из всего числа, а в итоге получается дробная его часть. Следовательно, результат вычисления данного выражения имеет тип `double`. Но к типу `int` приводится только значение, возвращаемое вторым методом `Math.Sqrt()`.

4 ГЛАВА

Операторы

В языке C# предусмотрен обширный ряд операторов, предоставляющих программирующему возможность полного контроля над построением и вычислением выражений. Большинство операторов в C# относится к следующим категориям: *арифметические, поразрядные, логические* и операторы *отношения*. Все перечисленные категории операторов рассматриваются в этой главе. Кроме того, в C# предусмотрен ряд других операторов для особых случаев, включая индексирование массивов, доступ к членам класса и обработку лямбда-выражений. Эти специальные операторы рассматриваются далее в книге вместе с теми средствами, в которых они применяются.

Арифметические операторы

Арифметические операторы, представленные в языке C#, приведены ниже.

Оператор	Действие
+	Сложение
–	Вычитание, унарный минус
*	Умножение
/	Деление
%	Деление по модулю
--	Декремент
++	Инкремент

Операторы +, –, * и / действуют так, как предполагает их обозначение. Их можно применять к любому встроенному числовому типу данных.

Действие арифметических операторов не требует особых пояснений, за исключением следующих особых случаев. Прежде всего, не следует забывать, что когда оператор / применяется к целому числу, то любой остаток от деления отбрасывается; например, результат целочисленного деления 10/3 будет равен 3. Остаток от этого деления можно получить с помощью оператора деления по модулю (%), который иначе называется *оператором вычисления остатка*. Он дает остаток от целочисленного деления. Например, `10 % 3` равно `1`. В C# оператор % можно применять как к целочисленным типам данных, так и к типам с плавающей точкой. Поэтому `10.0 % 3.0` также равно `1`. В этом отношении C# отличается от языков C и C++, где операции деления по модулю разрешаются только для целочисленных типов данных. В приведенном ниже примере программы демонстрируется применение оператора деления по модулю.

```
// Продемонстрировать применение оператора %.

using System;

class ModDemo {
  static void Main() {
    int iresult, irem;
    double dresult, drem;

    iresult = 10 / 3;
    irem = 10 % 3;

    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;

    Console.WriteLine("Результат и остаток от деления 10 / 3: " +
                      iresult + " " + irem);
    Console.WriteLine("Результат и остаток от деления 10.0 / 3.0: " +
                      dresult + " " + drem);
  }
}
```

Результат выполнения этой программы приведен ниже.

```
Результат и остаток от деления 10 / 3: 3 1
Результат и остаток от деления 10.0 / 3.0: 3.33333333333333 1
```

Как видите, обе операции, % целочисленного типа и с плавающей точкой, дают один и тот же остаток, равный 1.

Операторы инкремента и декремента

Операторы инкремента (++) и декремента (--) были представлены в главе 2. Как станет ясно в дальнейшем, они обладают рядом особых и довольно интересных свойств. Но сначала выясним основное назначение этих операторов.

Оператор инкремента увеличивает свой операнд на 1, а оператор декремента уменьшает операнд на 1. Следовательно, оператор

```
x++;
```

равнозначен оператору

```
x = x + 1;
```

а оператор

```
x--;
```

равносилен оператору

```
x = x - 1;
```

Следует, однако, иметь в виду, что в инкрементной или декрементной форме значение переменной x вычисляется только один, а не два раза. В некоторых случаях это позволяет повысить эффективность выполнения программы.

Оба оператора инкремента и декремента можно указывать до операнда (в префиксной форме) или же после операнда (в постфиксной форме). Например, оператор

```
x = x + 1;
```

может быть записан в следующем виде:

```
++x; // префиксная форма
```

или же в таком виде:

```
x++; // постфиксная форма
```

В приведенном выше примере форма инкремента (префиксная или постфиксная) особого значения не имеет. Но если оператор инкремента или декремента используется в длинном выражении, то отличие в форме его записи уже имеет значение. Когда оператор инкремента или декремента *предшествует* своему операнду, то результатом операции становится значение операнда *после* инкремента или декремента. А когда оператор инкремента или декремента следует *после* своего операнда, то результатом операции становится значение операнда *до* инкремента или декремента. Рассмотрим следующий фрагмент кода.

```
x = 10;
y = ++x;
```

В данном случае значение переменной y будет установлено равным 11, поскольку значение переменной x сначала увеличивается на 1, а затем присваивается переменной y. Но во фрагменте кода

```
x = 10;
y = x++;
```

значение переменной y будет установлено равным 10, так как в этом случае значение переменной x сначала присваивается переменной y, а затем увеличивается на 1. В обоих случаях значение переменной x оказывается равным 11. Отличие состоит лишь том, когда именно это значение станет равным 11: до или после его присваивания переменной y.

Возможность управлять моментом инкремента или декремента дает немало преимуществ при программировании. Обратимся к следующему примеру программы, в которой формируется последовательный ряд чисел.

```
// Продемонстрировать отличие между префиксной
// и постфиксной формами оператора инкремента (++).
```

```
using System;

class PrePostDemo {
  static void Main() {
    int x, y;
    int i;

    x = 1;
    y = 0;
    Console.WriteLine("Ряд чисел, полученных " +
                      "с помощью оператора y = y + x++;");
    for(i = 0; i < 10; i++) {
      y = y + x++; // постфиксная форма оператора ++

      Console.WriteLine(y + " ");
    }
    Console.WriteLine();

    x = 1;
    y = 0;
    Console.WriteLine("Ряд чисел, полученных " +
                      "с помощью оператора y = y + ++x;");
    for(i = 0; i < 10; i++) {
      y = y + ++x; // префиксная форма оператора ++

      Console.WriteLine(y + " ");
    }
    Console.WriteLine();
  }
}
```

Выполнение этой программы дает следующий результат.

```
Ряд чисел, полученных с помощью оператора y = y + x++
1
3
6
10
15
14
21
28
36
45
55

Ряд чисел, полученных с помощью оператора y = y + ++x;
2
5
9
14
20
```

```
27
35
44
54
65
```

Как подтверждает приведенный выше результат, в операторе

```
y = y + x++;
```

первоначальное значение переменной x складывается с самим собой, а полученный результат присваивается переменной y. После этого значение переменной x увеличивается на 1. Но в операторе

```
y = y + ++x;
```

значение переменной x сначала увеличивается на 1, затем складывается с первоначальным значением этой же переменной, а полученный результат присваивается переменной y. Как следует из приведенного выше результата, простая замена префиксной формы записи оператора ++x постфиксной формой x++ приводит к существенному изменению последовательного ряда получаемых чисел.

И еще одно замечание по поводу приведенного выше примера: не пугайтесь выражений, подобных следующему:

```
y + ++x
```

Такое расположение рядом двух операторов может показаться не совсем привычным, но компилятор воспримет их в правильной последовательности. Нужно лишь запомнить, что в данном выражении значение переменной y складывается с увеличенным на 1 значением переменной x.

Операторы отношения и логические операторы

В обозначениях *оператор отношения* и *логический оператор* термин *отношения* означает взаимосвязь, которая может существовать между двумя значениями, а термин *логический* — взаимосвязь между логическими значениями "истина" и "ложь". И поскольку операторы отношения дают истинные или ложные результаты, то они нередко применяются вместе с логическими операторами. Именно по этой причине они и рассматриваются совместно в данном разделе.

Ниже перечислены операторы отношения.

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

К числу логических относятся операторы, приведенные ниже.

Оператор	Значение
&	И
|	ИЛИ
^	Исключающее ИЛИ
&&	Укороченное И
||	Укороченное ИЛИ
!	НЕ

Результатом выполнения оператора отношения или логического оператора является логическое значение типа `bool`.

В целом, объекты можно сравнивать на равенство или неравенство, используя операторы отношения == и !=. А операторы сравнения <, >, <= или >= могут применяться только к тем типам данных, которые поддерживают отношение порядка. Следовательно, операторы отношения можно применять ко всем числовым типам данных. Но значения типа `bool` могут сравниваться только на равенство или неравенство, поскольку истинные (`true`) и ложные (`false`) значения не упорядочиваются. Например, сравнение `true > false` в C# не имеет смысла.

Операнды логических операторов должны относиться к типу `bool`, а результат выполнения логической операции также относится к типу `bool`. Логические операторы &, |, ^ и ! поддерживают основные логические операции И, ИЛИ, исключающее ИЛИ и НЕ в соответствии с приведенной ниже таблицей истинности.

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Как следует из приведенной выше таблицы, результатом выполнения логической операции исключающее ИЛИ будет истинное значение (`true`), если один и только один ее операнд имеет значение `true`.

Ниже приведен пример программы, демонстрирующий применение нескольких операторов отношения и логических операторов.

```
// Продемонстрировать применение операторов
// отношения и логических операторов.

using System;

class RelLogOps {
  static void Main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) Console.WriteLine("i < j");
    if(i <= j) Console.WriteLine("i <= j");
    if(i != j) Console.WriteLine("i != j");
```

```
    if(i == j) Console.WriteLine("Нельзя выполнить");
    if(i >= j) Console.WriteLine("Нельзя выполнить");
    if(i > j) Console.WriteLine("Нельзя выполнить");

    b1 = true;
    b2 = false;
    if(b1 & b2) Console.WriteLine("Нельзя выполнить");
    if(!(b1 & b2)) Console.WriteLine("!(b1 & b2) — true");
    if(b1 | b2) Console.WriteLine("b1 | b2 — true");
    if(b1 ^ b2) Console.WriteLine("b1 ^ b2 — true");
  }
}
```

Выполнение этой программы дает следующий результат.

```
i < j
i <= j
i != j
!(b1 & b2) — true
b1 | b2 — true
b1 ^ b2 — true
```

Логические операторы в C# выполняют наиболее распространенные логические операции. Тем не менее существует ряд операций, выполняемых по правилам формальной логики. Эти логические операции могут быть построены с помощью логических операторов, поддерживаемых в C#. Следовательно, в C# предусмотрен такой набор логических операторов, которого достаточно для построения практически любой логической операции, в том числе импликации. *Импликация* — это двоичная операция, результатом которой является ложное значение только в том случае, если левый ее операнд имеет истинное значение, а правый — ложное. (Операция импликации отражает следующий принцип: истина не может подразумевать ложь.) Ниже приведена таблица истинности для операции импликации.

p	q	Результат импликации p и q
true	true	true
true	false	false
false	false	true
false	true	true

Операция импликации может быть построена на основе комбинации логических операторов ! и |, как в приведенной ниже строке кода.

```
!p | q
```

В следующем примере программы демонстрируется подобная реализация операции импликации.

```
// Построение операции импликации в C#.

using System;

class Implication {
  static void Main() {
    bool p=false, q=false;
```

```
    int i, j;

    for(i = 0; i < 2; i++) {
      for(j = 0; j < 2; j++) {
        if(i==0) p = true;
        if(i==1) p = false;
        if(j==0) q = true;
        if(j==1) q = false;

        Console.WriteLine("p равно " + p + ", q равно " + q);
        if(!p | q)
           Console.WriteLine("Результат импликации " + p +
                             " и " + q + " равен " + true);
        Console.WriteLine();
      }
    }
  }
}
```

Результат выполнения этой программы выглядит так.

```
p равно True, q равно True
Результат импликации True и True равен True

p равно True, q равно False

p равно False, q равно False
Результат импликации False и True равен True

p равно False, q равно False
Результат импликации False и False равен True
```

Укороченные логические операторы

В C# предусмотрены также специальные, *укороченные*, варианты логических операторов И и *ИЛИ*, предназначенные для получения более эффективного кода. Поясним это на следующих примерах логических операций. Если первый операнд логической операции И имеет ложное значение (`false`), то ее результат будет иметь ложное значение независимо от значения второго операнда. Если же первый операнд логической операции *ИЛИ* имеет истинное значение (`true`), то ее результат будет иметь истинное значение независимо от значения второго операнда. Благодаря тому что значение второго операнда в этих операциях вычислять не нужно, экономится время и повышается эффективность кода.

Укороченная логическая операция И выполняется с помощью оператора `&&`, а укороченная логическая операция *ИЛИ* — с помощью оператора `||`. Этим укороченным логическим операторам соответствуют обычные логические операторы `&` и `|`. Единственное отличие укороченного логического оператора от обычного заключается в том, что второй его операнд вычисляется только по мере необходимости.

В приведенном ниже примере программы демонстрируется применение укороченного логического оператора И. В этой программе с помощью операции деления по модулю определяется следующее: делится ли значение переменной `d` на значение переменной `n` нацело. Если остаток от деления `n/d` равен нулю, то `n` делится на `d` нацело.

Но поскольку данная операция подразумевает деление, то для проверки условия деления на нуль служит укороченный логический оператор И.

```
// Продемонстрировать применение укороченных логических операторов.

using System;

class SCops {
  static void Main() {
    int n, d;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(n + " делится нацело на " + d);

    d = 0; // задать нулевое значение переменной d

    // d равно нулю, поэтому второй операнд не вычисляется
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(n + " делится нацело на " + d);

    // Если теперь попытаться сделать то же самое без укороченного
    // логического оператора, то возникнет ошибка из-за деления на нуль.
    if(d != 0 & (n % d) == 0)
      Console.WriteLine(n + " делится нацело на " + d);
  }
}
```

Для исключения ошибки из-за деления на нуль в операторе `if` сначала проверяется условие: равно ли нулю значение переменной `d`. Если оно равно нулю, то на этом выполнение укороченного логического оператора И завершается, а последующая операция деления по модулю не выполняется. Так, при первой проверке значение переменной `d` оказывается равным 2, поэтому выполняется операция деления по модулю. А при второй проверке это значение оказывается равным нулю, следовательно, операция деления по модулю пропускается, чтобы исключить деление на нуль. И наконец, выполняется обычный логический оператор И, когда вычисляются оба операнда. Если при этом происходит деление на нуль, то возникает ошибка при выполнении.

Укороченные логические операторы иногда оказываются более эффективными, чем их обычные аналоги. Так зачем же нужны обычные логические операторы И и ИЛИ? Дело в том, что в некоторых случаях требуется вычислять оба операнда логической операции И либо ИЛИ из-за возникающих побочных эффектов. Рассмотрим следующий пример программы.

```
// Продемонстрировать значение побочных эффектов.

using System;

class SideEffects {
  static void Main() {
    int i;
    bool someCondition = false;

    i = 0;
```

```
    // Значение переменной i инкрементируется,
    // несмотря на то, что оператор if не выполняется.
    if(someCondition & (++i < 100))
      Console.WriteLine("Не выводится");
    Console.WriteLine("Оператор if выполняется: " + i); // выводится 1

    // В данном случае значение переменной i не инкрементируется,
    // поскольку инкремент в укороченном логическом операторе опускается.
    if(someCondition && (++i < 100))
       Console.WriteLine("Не выводится");
    Console.WriteLine("Оператор if выполняется: " + i); // по-прежнему 1 !!
  }
}
```

Прежде всего обратим внимание на то, что переменная `someCondition` типа `bool` инициализируется значением `false`. Далее проанализируем каждый оператор `if`. Как следует из комментариев к данной программе, в первом операторе `if` переменная `i` инкрементируется, несмотря на то что значение переменной `someCondition` равно `false`. Когда применяется логический оператор `&`, как это имеет место в первом операторе `if`, выражение в правой части этого оператора вычисляется независимо от значения выражения в его левой части. А во втором операторе `if` применяется укороченный логический оператор. В этом случае значение переменной `i` не инкрементируется, поскольку левый операнд (переменная `someCondition`) имеет значение `false`, следовательно, выражение в правой части данного оператора пропускается. Из этого следует вывод: если в коде предполагается вычисление правого операнда логической операции И либо ИЛИ, то необходимо пользоваться неукороченными формами логических операций, доступных в C#.

И последнее замечание: укороченный оператор И называется также *условным логическим оператором И*, а укороченный оператор ИЛИ — *условным логическим оператором ИЛИ*.

Оператор присваивания

Оператор присваивания обозначается одиночным знаком равенства (=). В C# оператор присваивания действует таким же образом, как и в других языках программирования. Ниже приведена его общая форма.

```
имя_переменной = выражение
```

Здесь *имя_переменной* должно быть совместимо с типом *выражения*.

У оператора присваивания имеется одна интересная особенность, о которой вам будет полезно знать: он позволяет создавать цепочку операций присваивания. Рассмотрим, например, следующий фрагмент кода.

```
int x, y, z;
x = y = z = 100; // присвоить значение 100 переменным x, y и z
```

В приведенном выше фрагменте кода одно и то же значение 100 задается для переменных `x`, `y` и `z` с помощью единственного оператора присваивания. Это значение присваивается сначала переменной `z`, затем переменной `y` и, наконец, переменной `x`. Такой способ присваивания "по цепочке" удобен для задания общего значения целой группе переменных.

Составные операторы присваивания

В C# предусмотрены специальные составные операторы присваивания, упрощающие программирование некоторых операций присваивания. Обратимся сначала к простому примеру. Приведенный ниже оператор присваивания

```
x = x + 10;
```

можно переписать, используя следующий составной оператор присваивания.

```
x += 10;
```

Пара операторов += указывает компилятору на то, что переменной x должно быть присвоено ее первоначальное значение, увеличенное на 10.

Рассмотрим еще один пример. Оператор

```
x = x - 100;
```

и оператор

```
x -= 100;
```

выполняют одни и те же действия. Оба оператора присваивают переменной x ее первоначальное значение, уменьшенное на 100.

Для многих двоичных операций, т.е. операций, требующих наличия двух операндов, существуют отдельные составные операторы присваивания. Общая форма всех этих операторов имеет следующий вид:

имя_переменной op = выражение

где *op* — арифметический или логический оператор, применяемый вместе с оператором присваивания.

Ниже перечислены составные операторы присваивания для арифметических и логических операций.

+=	-=	*=	/=
%=	&=	|=	^=

Составные операторы присваивания записываются более кратко, чем их несоставные эквиваленты. Поэтому их иногда еще называют *укороченными операторами присваивания*.

У составных операторов присваивания имеются два главных преимущества. Во-первых, они более компактны, чем их "несокращенные" эквиваленты. И во-вторых, они дают более эффективный исполняемый код, поскольку левый операнд этих операторов вычисляется только один раз. Именно по этим причинам составные операторы присваивания чаще всего применяются в программах, профессионально написанных на C#.

Поразрядные операторы

В C# предусмотрен ряд *поразрядных* операторов, расширяющих круг задач, для решения которых можно применять C#. Поразрядные операторы воздействуют на отдельные двоичные разряды (биты) своих операндов. Они определены только для целочисленных операндов, поэтому их нельзя применять к данным типа `bool`, `float` или `double`.

Эти операторы называются *поразрядными*, поскольку они служат для проверки, установки или сдвига двоичных разрядов, составляющих целое значение. Среди прочего поразрядные операторы применяются для решения самых разных задач программирования на уровне системы, включая, например, анализ информации состояния устройства. Все доступные в C# поразрядные операторы приведены в табл. 4.1.

Таблица 4.1. Поразрядные операторы

Оператор	Значение
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ обозначаются следующим образом: &, |, ^ и ~. Они выполняют те же функции, что и их логические аналоги, рассмотренные выше. Но в отличие от логических операторов, поразрядные операторы действуют на уровне отдельных двоичных разрядов. Ниже приведены результаты поразрядных операций с двоичными единицами и нулями.

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

С точки зрения наиболее распространенного применения поразрядную операцию И можно рассматривать как способ подавления отдельных двоичных разрядов. Это означает, что если какой-нибудь бит в любом из операндов равен 0, то соответствующий бит результата будет сброшен в 0. Например:

```
  1101 0011
  1010 1010
&  _________
  1000 0010
```

В приведенном ниже примере программы демонстрируется применение поразрядного оператора & для преобразования нечетных чисел в четные. Для этой цели достаточно сбросить младший разряд числа. Например, число 9 имеет следующий двоичный вид: 0000 1001. Если сбросить младший разряд этого числа, то оно станет числом 8, а в двоичной форме — 0000 1000.

```
// Применить поразрядный оператор И, чтобы сделать число четным.

using System;
```

```
class MakeEven {
  static void Main() {
    ushort num;
    ushort i;

    for(i = 1; i <= 10; i++) {
      num = i;

      Console.WriteLine("num: " + num);

      num = (ushort) (num & 0xFFFE);

      Console.WriteLine("num после сброса младшего разряда: "
                        + num + "\n");
    }
  }
}
```

Результат выполнения этой программы приведен ниже.

```
num: 1
num после сброса младшего разряда: 0

num: 2
num после сброса младшего разряда: 2

num: 3
num после сброса младшего разряда: 2

num: 4
num после сброса младшего разряда: 4

num: 5
num после сброса младшего разряда: 4

num: 6
num после сброса младшего разряда: 6

num: 7
num после сброса младшего разряда: 6

num: 8
num после сброса младшего разряда: 8

num: 9
num после сброса младшего разряда: 8

num: 10
num после сброса младшего разряда: 10
```

Шестнадцатеричное значение `0xFFFE`, используемое в поразрядном операторе И, имеет следующую двоичную форму: 1111 1111 1111 1110. Таким образом, поразрядная операция И оставляет без изменения все двоичные разряды в числовом значении переменной `num`, кроме младшего разряда, который сбрасывается в нуль. В итоге

четные числа не претерпевают никаких изменений, а нечетные уменьшаются на 1 и становятся четными.

Поразрядным оператором И удобно также пользоваться для определения установленного или сброшенного состояния отдельного двоичного разряда. В следующем примере программы определяется, является ли число нечетным.

```
// Применить поразрядный оператор И, чтобы определить,
// является ли число нечетным.

using System;

class IsOdd {
  static void Main() {
    ushort num;

    num = 10;

    if((num & 1) == 1)
      Console.WriteLine("Не выводится.");

    num = 11;

    if((num & 1) == 1)
      Console.WriteLine(num + " — нечетное число.");

  }
}
```

Вот как выглядит результат выполнения этой программы.

```
11 — нечетное число.
```

В обоих операторах `if` из приведенной выше программы выполняется поразрядная операция И над числовыми значениями переменной `num` и 1. Если младший двоичный разряд числового значения переменной `num` установлен, т.е. содержит двоичную 1, то результат поразрядной операции `num & 1` оказывается равным 1. В противном случае он равен нулю. Поэтому оператор `if` может быть выполнен успешно лишь в том случае, если проверяемое число оказывается нечетным.

Возможностью проверять состояние отдельных двоичных разрядов с помощью поразрядного оператора `&` можно воспользоваться для написания программы, в которой отдельные двоичные разряды проверяемого значения типа `byte` приводятся в двоичной форме. Ниже показан один из способов написания такой программы.

```
// Показать биты, составляющие байт.

using System;

class ShowBits {
  static void Main() {
    int t;
    byte val;

    val = 123;
    for(t=128; t > 0; t = t/2) {
```

```
      if((val & t) != 0) Console.Write("1 ");
      if((val & t) == 0) Console.Write("0 ");
    }
  }
}
```

Выполнение этой программы дает следующий результат.

```
0 1 1 1 1 0 1 1
```

В цикле `for` из приведенной выше программы каждый бит значения переменной `val` проверяется с помощью поразрядного оператора И, чтобы выяснить, установлен ли этот бит или сброшен. Если он установлен, то выводится цифра 1, а если сброшен, то выводится цифра 0.

Поразрядный оператор ИЛИ может быть использован для установки отдельных двоичных разрядов. Если в 1 установлен какой-нибудь бит в любом из операндов этого оператора, то в 1 будет установлен и соответствующий бит в другом операнде. Например:

```
  1101 0011
| 1010 1010
  ---------
  1111 1011
```

Используя поразрядный оператор ИЛИ, можно без особого труда превратить упоминавшийся выше пример программы, преобразующей нечетные числа в четные, в приведенный ниже обратный пример, где четные числа преобразуются в нечетные.

```
// Применить поразрядный оператор ИЛИ, чтобы сделать число нечетным.

using System;

class MakeOdd {
  static void Main() {
    ushort num;
    ushort i;

    for(i = 1; i <= 10; i++) {
      num = i;

      Console.WriteLine("num: " + num);

      num = (ushort) (num | 1);

      Console.WriteLine("num после установки младшего разряда: " +
                        num + "\n");
    }
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
num: 1
num после установки младшего разряда: 1

num: 2
```

```
num после установки младшего разряда: 3

num: 3
num после установки младшего разряда: 3

num: 4
num после установки младшего разряда: 5

num: 5
num после установки младшего разряда: 5

num: 6
num после установки младшего разряда: 7

num: 7
num после установки младшего разряда: 7

num: 8
num после установки младшего разряда: 9

num: 9
num после установки младшего разряда: 9

num: 10
num после установки младшего разряда: 11
```

В приведенной выше программе выполняется поразрядная операция *ИЛИ* над каждым числовым значением переменной `num` и 1, поскольку 1 дает двоичное значение, в котором установлен младший разряд. В результате поразрядной операции *ИЛИ* над 1 и любым другим значением младший разряд последнего устанавливается, тогда как все остальные разряды остаются без изменения. Таким образом, результирующее числовое значение получается нечетным, если исходное значение было четным.

Поразрядный оператор исключающее ИЛИ устанавливает двоичный разряд операнда в том и только в том случае, если двоичные разряды сравниваемых операндов оказываются разными, как в приведенном ниже примере.

```
  0111 1111
  1011 1001
^ ---------
  1100 0110
```

У поразрядного оператора исключающее ИЛИ имеется одно интересное свойство, которое оказывается полезным в самых разных ситуациях. Так, если выполнить сначала поразрядную операцию исключающее *ИЛИ* одного значения `X` с другим значением `Y`, а затем такую же операцию над результатом предыдущей операции и значением `Y`, то вновь получится первоначальное значение `X`. Это означает, что в приведенном ниже фрагменте кода

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

значение переменной `R2` оказывается в итоге таким же, как и значение переменной `X`. Следовательно, в результате двух последовательно выполняемых поразрядных опера-

ций исключающее ИЛИ, в которых используется одно и то же значение, получается первоначальное значение. Этим свойством данной операции можно воспользоваться для написания простой программы шифрования, в которой некоторое целое значение служит в качестве ключа для кодирования и декодирования сообщения с помощью операции исключающее ИЛИ над символами этого сообщения. В первый раз операция исключающее ИЛИ выполняется для кодирования открытого текста в зашифрованный, а второй раз — для декодирования зашифрованного текста в открытый. Разумеется, такое шифрование не представляет никакой практической ценности, поскольку оно может быть легко разгадано. Тем не менее оно служит интересным примером для демонстрации результатов применения поразрядных операторов исключающее ИЛИ, как в приведенной ниже программе.

```
// Продемонстрировать применение поразрядного оператора исключающее ИЛИ.

using System;

class Encode {
  static void Main() {
    char ch1 = 'H';
    char ch2 = 'i';
    char ch3 = '!';
    int key = 88;

    Console.WriteLine("Исходное сообщение: " + ch1 + ch2 + ch3);

    // Зашифровать сообщение.
    ch1 = (char) (ch1 ^ key);
    ch2 = (char) (ch2 ^ key);
    ch3 = (char) (ch3 ^ key);

    Console.WriteLine("Зашифрованное сообщение: " + ch1 + ch2 + ch3);

    // Расшифровать сообщение.
    ch1 = (char) (ch1 ^ key);
    ch2 = (char) (ch2 ^ key);
    ch3 = (char) (ch3 ^ key);

    Console.WriteLine("Расшифрованное сообщение: " + ch1 + ch2 + ch3);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Исходное сообщение: Hi!
Зашифрованное сообщение: ❑1y
Расшифрованное сообщение: Hi!
```

Как видите, в результате выполнения двух последовательностей поразрядных операций исключающее ИЛИ получается расшифрованное сообщение. (Еще раз напомним, что такое шифрование не имеет никакой практической ценности, поскольку оно, в сущности, ненадежно.)

Поразрядный унарный оператор НЕ (или оператор дополнения до 1) изменяет на обратное состояние всех двоичных разрядов операнда. Так, если некоторое целое значение A имеет комбинацию двоичных разрядов 1001 0110, то в результате поразрядной операции `~A` получается значение с комбинацией двоичных разрядов 0110 1001.

В следующем примере программы демонстрируется применение поразрядного оператора НЕ с выводом некоторого числа и его дополнения до 1 в двоичном коде.

```
// Продемонстрировать применение поразрядного унарного оператора НЕ.

using System;

class NotDemo {
  static void Main() {
    sbyte b = -34;

    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      if((b & t) == 0) Console.Write("0 ");
    }
    Console.WriteLine();

    // обратить все биты
    b = (sbyte) ~b;
    for(int t=128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      if((b & t) == 0) Console.Write("0 ");
    }
  }
}
```

Результат выполнения этой программы приведен ниже.

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

Операторы сдвига

В C# имеется возможность сдвигать двоичные разряды, составляющие целое значение, влево или вправо на заданную величину. Для этой цели в C# определены два приведенных ниже оператора сдвига двоичных разрядов.

<<	Сдвиг влево
>>	Сдвиг вправо

Ниже приведена общая форма для этих операторов:

```
значение << число_битов
значение >> число_битов
```

где *число_битов* — это число двоичных разрядов, на которое сдвигается указанное *значение*.

При сдвиге влево все двоичные разряды в указываемом значении сдвигаются на одну позицию влево, а младший разряд сбрасывается в нуль. При сдвиге вправо все двоичные разряды в указываемом значении сдвигаются на одну позицию вправо. Если вправо сдвигается целое значение без знака, то старший разряд сбрасывается в нуль. А если вправо сдвигается целое значение со знаком, то разряд знака сохраняется. Напомним, что для представления отрицательных чисел старший разряд целого числа устанавливается в 1. Так, если сдвигаемое значение является отрицательным, то при

каждом сдвиге вправо старший разряд числа устанавливается в 1. А если сдвигаемое значение является положительным, то при каждом сдвиге вправо старший разряд числа сбрасывается в нуль.

При сдвиге влево и вправо крайние двоичные разряды теряются. Восстановить потерянные при сдвиге двоичные разряды нельзя, поскольку сдвиг в данном случае не является циклическим.

Ниже приведен пример программы, наглядно демонстрирующий действие сдвига влево и вправо. В данном примере сначала задается первоначальное целое значение, равное 1. Это означает, что младший разряд этого значения установлен. Затем это целое значение сдвигается восемь раз подряд влево. После каждого сдвига выводятся восемь младших двоичных разрядов данного значения. Далее процесс повторяется, но на этот раз 1 устанавливается на позиции восьмого разряда, а по существу, задается целое значение 128, которое затем сдвигается восемь раз подряд вправо.

```
// Продемонстрировать применение операторов сдвига.

using System;

class ShiftDemo {
  static void Main() {
    int val = 1;

    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        if((val & t) == 0) Console.Write("0 ");
      }
      Console.WriteLine();
      val = val << 1; // сдвиг влево
    }
    Console.WriteLine();

    val = 128;
    for(int i = 0; i < 8; i++) {
      for(int t=128; t > 0; t = t/2) {
          if((val & t) != 0) Console.Write("1 ");
          if((val & t) == 0) Console.Write("0 ");
      }
      Console.WriteLine();
      val = val >> 1; // сдвиг вправо
    }
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
```

```
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Двоичные разряды соответствуют форме представления чисел в степени 2, и поэтому операторы сдвига могут быть использованы для умножения или деления целых значений на 2. Так, при сдвиге вправо целое значение удваивается, а при сдвиге влево — уменьшается наполовину. Разумеется, все это справедливо лишь в том случае, если крайние разряды не теряются при сдвиге в ту или иную сторону. Ниже приведен соответствующий пример.

```
// Применить операторы сдвига для умножения и деления на 2.

using System;

class MultDiv {
  static void Main() {
    int n;

    n = 10;

    Console.WriteLine("Значение переменной n: " + n);

    // Умножить на 2.
    n = n << 1;
    Console.WriteLine("Значение переменной n после " +
                      "операции n = n * 2: " + n);

    // Умножить на 4.
    n = n << 2;
    Console.WriteLine("Значение переменной n после " +
                      "операции n = n * 4: " + n);

    // Разделить на 2.
    n = n >> 1;
    Console.WriteLine("Значение переменной n после " +
                      "операции n = n / 2: " + n);

    // Разделить на 4.
    n = n >> 2;
    Console.WriteLine("Значение переменной n после " +
                      "операции n = n / 4: " + n);
    Console.WriteLine();

    // Установить переменную n в исходное состояние.
    n = 10;
    Console.WriteLine("Значение переменной n: " + n);
```

```
    // Умножить на 2 тридцать раз.
    n = n << 30; // данные теряются
    Console.WriteLine("Значение переменной n после " +
                      "сдвига на 30 позиций влево: " + n);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Значение переменной n: 10
Значение переменной n после операции n = n * 2: 20
Значение переменной n после операции n = n * 4: 80
Значение переменной n после операции n = n / 2: 40
Значение переменной n после операции n = n / 4: 10

Значение переменной n: 10
Значение переменной n после сдвига на 30 позиций влево: -2147483648
```

Обратите внимание на последнюю строку приведенного выше результата. Когда целое значение 10 сдвигается влево тридцать раз подряд, информация теряется, поскольку двоичные разряды сдвигаются за пределы представления чисел для типа `int`. В данном случае получается совершенно "непригодное" значение, которое оказывается к тому же отрицательным, поскольку в результате сдвига в старшем разряде, используемом в качестве знакового, оказывается 1, а следовательно, данное числовое значение должно интерпретироваться как отрицательное. Этот пример наглядно показывает, что применять операторы сдвига для умножения или деления на 2 следует очень аккуратно. (Подробнее о типах данных со знаком и без знака см. в главе 3.)

Поразрядные составные операторы присваивания

Все двоичные поразрядные операторы могут быть использованы в составных операциях присваивания. Например, в двух приведенных ниже операторах переменной x присваивается результат выполнения операции исключающее ИЛИ над первоначальным значением переменной x и числовым значением 127.

```
x = x ^ 127;
x ^= 127;
```

Оператор ?

Оператор ? относится к числу самых примечательных в C#. Он представляет собой условный оператор и часто используется вместо определенных видов конструкций `if-then-else`. Оператор ? иногда еще называют *тернарным*, поскольку для него требуются три операнда. Ниже приведена общая форма этого оператора.

```
Выражение1 ? Выражение2 : Выражение3;
```

Здесь *Выражение1* должно относиться к типу `bool`, а *Выражение2* и *Выражение3* — к одному и тому же типу. Обратите внимание на применение двоеточия и его местоположение в операторе ?.

Значение выражения ? определяется следующим образом. Сначала вычисляется *Выражение1*. Если оно истинно, то вычисляется *Выражение2*, а полученный результат определяет значение всего выражения ? в целом. Если же *Выражение1* оказывается

ложным, то вычисляется *Выражение3*, и его значение становится общим для всего выражения ?. Рассмотрим следующий пример, в котором переменной absval присваивается значение переменной val.

```
absval = val < 0 ? -val : val; // получить абсолютное значение переменной val
```

В данном примере переменной absval присваивается значение переменной val, если оно больше или равно нулю. Если же значение переменной val отрицательно, то переменной absval присваивается результат отрицания этого значения, что в итоге дает положительное значение.

Ниже приведен еще один пример применения оператора ?. В данной программе одно число делится на другое, но при этом исключается деление на нуль.

```
// Исключить деление на нуль, используя оператор?.

using System;

class NoZeroDiv {
  static void Main() {
    int result;

    for(int i = -5; i < 6; i++) {
      result = i != 0 ? 100 / i : 0;
      if(i != 0)
        Console.WriteLine("100 / " + i + " равно " + result);
    }
  }
}
```

Выполнение этой программы дает следующий результат.

```
100 / -5 равно -20
100 / -4 равно -25
100 / -3 равно -33
100 / -2 равно -50
100 / -1 равно -100
100 / 1 равно 100
100 / 2 равно 50
100 / 3 равно 33
100 / 4 равно 25
100 / 5 равно 20
```

Обратите особое внимание на следующую строку из приведенной выше программы.

```
result = i != 0 ? 100 / i : 0;
```

В этой строке переменной result присваивается результат деления числа 100 на значение переменной i. Но это деление осуществляется лишь в том случае, если значение переменной i не равно нулю. Когда же оно равно нулю, переменной result присваивается значение, обнуляющее результат.

Присваивать переменной результат выполнения оператора ? совсем не обязательно. Например, значение, которое дает оператор ?, можно использовать в качестве аргумента при вызове метода. А если все выражения в операторе ? относятся к типу bool, то такой оператор может заменить собой условное выражение в цикле или операторе

if. В приведенном ниже примере программы выводятся результаты деления числа 100 только на четные, ненулевые значения.

```
// Разделить только на четные, ненулевые значения.

using System;

class NoZeroDiv2 {
  static void Main() {

    for(int i = -5; i < 6; i++)
      if(i != 0 ? (i%2 == 0) : false)
        Console.WriteLine("100 / " + i + " равно " + 100 / i);
  }
}
```

Обратите внимание на оператор if в приведенной выше программе. Если значение переменной i равно нулю, то оператор if дает ложный результат. А если значение переменной i не равно нулю, то оператор if дает истинный результат, когда значение переменной i оказывается четным, и ложный результат, если оно нечетное. Благодаря этому допускается деление только на четные и ненулевые значения. Несмотря на то что данный пример служит лишь для целей демонстрации, подобные конструкции иногда оказываются весьма полезными.

Использование пробелов и круглых скобок

В выражении на C# допускается наличие символов табуляции и пробелов, благодаря которым оно становится более удобным для чтения. Например, оба приведенных ниже выражения, по существу, одинаковы, но второе читается легче.

```
x=10/y*(127+x);
x = 10 / y * (127 + x);
```

Скобки могут служить для группирования подвыражений, по существу, повышая порядок предшествования заключенных в них операций, как в алгебре. Применение лишних или дополнительных скобок не приводит к ошибкам и не замедляет вычисление выражения. Поэтому скобки рекомендуется использовать, чтобы сделать более ясным и понятным порядок вычисления как для самого автора программы, так и для тех, кто будет разбираться в ней впоследствии. Например, какое из двух приведенных ниже выражение легче читается?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Предшествование операторов

В табл. 4.2 приведен порядок предшествования всех операторов в C#: от самого высокого до самого низкого. В таблицу включен ряд операторов, рассматриваемых далее в этой книге.

Таблица 4.2. Предшествование операторов в C#

Наивысший порядок									
()	[]	.	++ (постфиксный)	-- (постфиксный)	checked	new	sizeof	typeof	unchecked
!	~	(приведение типов)	+ (унарный)	- (унарный)	++ (префиксный)	-- префиксный)			
*	/	%							
+	-								
<<	>>								
<	>	<=	>=	is					
==	!=								
&									
^									
|									
&&									
||									
??									
?:									
=	op=	=>							
Наинизший порядок									

5 ГЛАВА

Управляющие операторы

В этой главе речь пойдет об операторах, управляющих ходом выполнения программы на C#. Управляющие операторы разделяются на три категории: операторы *выбора*, к числу которых относятся операторы `if` и `switch`, *итерационные* операторы, в том числе операторы цикла `for`, `while`, `do-while` и `foreach`, а также операторы *перехода*: `break`, `continue`, `goto`, `return` и `throw`. За исключением оператора `throw`, который является неотъемлемой частью встроенного в C# механизма обработки исключительных ситуаций, рассматриваемого в главе 13, все остальные управляющие операторы представлены в этой главе.

Оператор `if`

Оператор `if` уже был представлен в главе 2, а здесь он рассматривается более подробно. Ниже приведена полная форма этого оператора:

```
if(условие) оператор;
else оператор;
```

где *условие* — это некоторое условное выражение, а *оператор* — адресат операторов `if` и `else`. Оператор `else` не является обязательным. Адресатом обоих операторов, `if` и `else`, могут также служить блоки операторов. Ниже приведена общая форма оператора `if`, в котором используются блоки операторов.

```
if(условие)
{
  последовательность операторов
}
```

```
else
{
  последовательность операторов
}
```

Если условное выражение оказывается истинным, то выполняется адресат оператора `if`. В противном случае выполняется адресат оператора `else`, если таковой существует. Но одновременно не может выполняться и то и другое. Условное выражение, управляющее оператором `if`, должно давать результат типа `bool`.

Ниже приведен пример простой программы, в которой операторы `if` и `else` используются для того, чтобы сообщить, является ли число положительным или отрицательным.

```
// Определить, является ли числовое значение положительным или
отрицательным.

using System;

class PosNeg {
  static void Main() {
    int i;

    for(i=-5; i <= 5; i++) {
     Console.Write("Проверка " + i + ": ");

    if(i < 0) Console.WriteLine("отрицательное число");
    else Console.WriteLine("положительное число");
    }
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Проверка -5: отрицательное число
Проверка -4: отрицательное число
Проверка -3: отрицательное число
Проверка -2: отрицательное число
Проверка -1: отрицательное число
Проверка 0: положительное число
Проверка 1: положительное число
Проверка 2: положительное число
Проверка 3: положительное число
Проверка 4: положительное число
Проверка 5: положительное число
```

Если в данном примере значение переменной `i` оказывается меньше нуля, то выполнятся адресат оператора `if`. В противном случае выполняется адресат оператора `else`, одновременно они не выполняются.

Вложенные операторы `if`

Вложенным называется такой оператор `if`, который является адресатом другого оператора `if` или же оператора `else`. Вложенные операторы `if` очень часто применяются в программировании. Что же касается их применения в C#, то не следует забывать, что любой оператор `else` всегда связан с ближайшим оператором `if`, т.е. с тем

оператором if, который находится в том же самом блоке, где и оператор else, но не с другим оператором else. Рассмотрим следующий пример.

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // этот оператор else связан с оператором if(k > 100)
}
else a = d; // этот оператор else связан с оператором if(i == 10)
```

Как следует из комментариев к приведенному выше фрагменту кода, последний оператор else не связан с оператором if(j < 20), поскольку они не находятся в одном и том же блоке, несмотря на то, что этот оператор является для него ближайшим оператором if без вспомогательного оператора else. Напротив, последний оператор else связан с оператором if(i == 10). А внутренний оператор else связан с оператором if(k > 100), поскольку этот последний является для него ближайшим оператором if в том же самом блоке.

В приведенном ниже примере программы демонстрируется применение вложенного оператора if. В представленной ранее программе определения положительных и отрицательных чисел о нуле сообщалось как о положительном числе. Но, как правило, нуль считается числом, не имеющим знака. Поэтому в следующей версии данной программы о нуле сообщается как о числе, которое не является ни положительным, ни отрицательным.

```
// Определить, является ли числовое значение
// положительным, отрицательным или нулевым.

using System;

class PosNegZero {
  static void Main() {
    int i;

    for(i=-5; i <= 5; i++) {
      Console.Write("Проверка " + i + ": ");
      if(i < 0) Console.WriteLine("отрицательное число");
      else if(i == 0) Console.WriteLine("число без знака");
      else Console.WriteLine("положительное число");
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Проверка -5: отрицательное число
Проверка -4: отрицательное число
Проверка -3: отрицательное число
Проверка -2: отрицательное число
Проверка -1: отрицательное число
Проверка 0: число без знака
Проверка 1: положительное число
Проверка 2: положительное число
Проверка 3: положительное число
Проверка 4: положительное число
Проверка 5: положительное число
```

Конструкция if-else-if

В программировании часто применяется *многоступенчатая конструкция* `if-else-if`, состоящая из вложенных операторов `if`. Ниже приведена ее общая форма.

```
if(условие)
  оператор;
else if (условие)
  оператор;
else if (условие)
  оператор;
.
.
.
else
  оператор;
```

Условные выражения в такой конструкции вычисляются сверху вниз. Как только обнаружится истинное условие, выполняется связанный с ним оператор, а все остальные операторы в многоступенчатой конструкции опускаются.

Если ни одно из условий не является истинным, то выполняется последний оператор `else`, который зачастую служит в качестве условия, устанавливаемого по умолчанию. Когда же последний оператор `else` отсутствует, а все остальные проверки по условию дают ложный результат, то никаких действий вообще не выполняется.

В приведенном ниже примере программы демонстрируется применение многоступенчатой конструкции `if-else-if`. В этой программе обнаруживается наименьший множитель заданного целого значения, состоящий из одной цифры.

```
// Определить наименьший множитель заданного
// целого значения, состоящий из одной цифры.

using System;

class Ladder {
  static void Main() {
    int num;

    for(num = 2; num < 12; num++) {
    if((num % 2) == 0)
      Console.WriteLine("Наименьший множитель числа " + num + " равен 2.");
    else if((num % 3) == 0)
      Console.WriteLine("Наименьший множитель числа " + num + " равен 3.");
    else if((num % 5) == 0)
      Console.WriteLine("Наименьший множитель числа " + num + " равен 5.");
    else if((num % 7) == 0)
      Console.WriteLine("Наименьший множитель числа " + num + " равен 7.");
    else
    Console.WriteLine(num + " не делится на 2, 3, 5 или 7.");
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Наименьший множитель числа 2 равен 2
Наименьший множитель числа 3 равен 3
```

```
Наименьший множитель числа 4 равен 2
Наименьший множитель числа 5 равен 5
Наименьший множитель числа 6 равен 2
Наименьший множитель числа 7 равен 7
Наименьший множитель числа 8 равен 2
Наименьший множитель числа 9 равен 3
Наименьший множитель числа 10 равен 2
11 не делится на 2, 3, 5 или 7.
```

Как видите, последний оператор `else` выполняется лишь в том случае, если не удается выполнить ни один из предыдущих операторов.

Оператор `switch`

Вторым оператором выбора в C# является оператор `switch`, который обеспечивает многонаправленное ветвление программы. Следовательно, этот оператор позволяет сделать выбор среди нескольких альтернативных вариантов дальнейшего выполнения программы. Несмотря на то что многонаправленная проверка может быть организована с помощью последовательного ряда вложенных операторов `if`, во многих случаях более эффективным оказывается применение оператора `switch`. Этот оператор действует следующим образом. Значение выражения последовательно сравнивается с константами выбора из заданного списка. Как только будет обнаружено совпадение с одним из условий выбора, выполняется последовательность операторов, связанных с этим условием. Ниже приведена общая форма оператора `switch`.

```
switch(выражение) {
  case константа1:
    последовательность операторов
    break;
case константа2:
    последовательность операторов
    break;
  case константа3:
    последовательность операторов
    break;
.
.
.
  default:
    последовательность операторов
    break;
}
```

Заданное *выражение* в операторе `switch` должно быть целочисленного типа (`char`, `byte`, `short` или `int`), перечислимого или же строкового. (О перечислениях и символьных строках типа `string` речь пойдет далее в этой книге.) А выражения других типов, например с плавающей точкой, в операторе `switch` не допускаются. Зачастую выражение, управляющее оператором `switch`, просто сводится к одной переменной. Кроме того, константы выбора должны иметь тип, совместимый с типом выражения. В одном операторе `switch` не допускается наличие двух одинаковых по значению констант выбора.

Последовательность операторов из ветви `default` выполняется в том случае, если ни одна из констант выбора не совпадает с заданным выражением. Ветвь `default` не является обязательной. Если же она отсутствует и выражение не совпадает ни с одним из условий выбора, то никаких действий вообще не выполняется. Если же происходит совпадение с одним из условий выбора, то выполняются операторы, связанные с этим условием, вплоть до оператора `break`.

Ниже приведен пример программы, в котором демонстрируется применение оператора `switch`.

```
// Продемонстрировать применение оператора switch.

using System;

class SwitchDemo {
  static void Main() {
    int i;

    for(i=0; i<10; i++)
      switch(i) {
        case 0:
          Console.WriteLine("i равно нулю");
          break;
        case 1:
          Console.WriteLine("i равно единице");
          break;
        case 2:
          Console.WriteLine("i равно двум");
          break;
        case 3:
          Console.WriteLine("i равно трем");
          break;
        case 4:
          Console.WriteLine("i равно четырем");
        break;
        default:
          Console.WriteLine("i равно или больше пяти");
        break;
      }
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
i равно нулю.
i равно единице.
i равно двум.
i равно трем.
i равно четырем.
i равно или больше пяти.
i равно или больше пяти.
i равно или больше пяти.
i равно или больше пяти.
i равно или больше пяти.
```

Как видите, на каждом шаге цикла выполняются операторы, связанные с совпадающей константой выбора, в обход всех остальных операторов. Когда же значение

переменной i становится равным или больше пяти, то оно не совпадает ни с одной из констант выбора, а следовательно, выполняются операторы из ветви default.

В приведенном выше примере оператором switch управляла переменная i типа int. Как пояснялось ранее, для управления оператором switch может быть использовано выражение любого целочисленного типа, включая и char. Ниже приведен пример применения выражения и констант выбора типа char в операторе switch.

```
// Использовать элементы типа char для управления оператором switch.

using System;

class SwitchDemo2 {
  static void Main() {
    char ch;

    for(ch='A'; ch<= 'E'; ch++)
      switch(ch) {
        case 'A':
          Console.WriteLine("ch содержит A");
        break;
        case 'B':
          Console.WriteLine("ch содержит B");
          break;
        case 'C':
          Console.WriteLine("ch содержит C");
          break;
        case 'D':
          Console.WriteLine("ch содержит D");
          break;
        case 'E':
          Console.WriteLine("ch содержит E");
          break;
      }
  }
}
```

Вот какой результат дает выполнение этой программы.

```
ch содержит A
ch содержит B
ch содержит C
ch содержит D
ch содержит E
```

Обратите в данном примере внимание на отсутствие ветви default в операторе switch. Напомним, что ветвь default не является обязательной. Когда она не нужна, ее можно просто опустить.

Переход последовательности операторов, связанных с одной ветвью case, в следующую ветвь case считается ошибкой, поскольку в C# должно непременно соблюдаться правило недопущения "провалов" в передаче управления ходом выполнения программы. Именно поэтому последовательность операторов в каждой ветви case оператора switch оканчивается оператором break. (Избежать подобных "провалов", можно также с помощью оператора безусловного перехода goto, рассматриваемого далее в этой главе, но для данной цели чаще применяется оператор break.) Когда

в последовательности операторов отдельной ветви case встречается оператор break, происходит выход не только из этой ветви, но из всего оператора switch, а выполнение программы возобновляется со следующего оператора, находящегося за пределами оператора switch. Последовательность операторов в ветви default также должна быть лишена "провалов", поэтому она завершается, как правило, оператором break.

Правило недопущения "провалов" относится к тем особенностям языка C#, которыми он отличается от C, C++ и Java. В этих языках программирования одна ветвь case может переходить (т.е. "проваливаться") в другую. Данное правило установлено в C# для ветвей case по двум причинам. Во-первых, оно дает компилятору возможность свободно изменять порядок следования последовательностей операторов из ветвей case для целей оптимизации. Такая реорганизация была бы невозможной, если бы одна ветвь case могла переходить в другую. И во-вторых, требование завершать каждую ветвь case явным образом исключает непроизвольные ошибки программирования, допускающие переход одной ветви case в другую.

Несмотря на то что правило недопущения "провалов" не допускает переход одной ветви case в другую, в двух или более ветвях case все же разрешается ссылаться с помощью меток на одну и ту же кодовую последовательность, как показано в следующем примере программы.

```
// Пример "проваливания" пустых ветвей case.

using System;

class EmptyCasesCanFall {
  static void Main() {
    int i;

    for(i=1; i < 5; i++)
      switch(i) {
        case 1:
        case 2:
        case 3: Console.WriteLine("i равно 1, 2 или 3");
        break;
        case 4: Console.WriteLine("i равно 4");
        break;
      }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
i равно 1, 2 или 3
i равно 1, 2 или 3
i равно 1, 2 или 3
i равно 4
```

Если значение переменной i в данном примере равно 1, 2 или 3, то выполняется первый оператор, содержащий вызов метода WriteLine(). Такое расположение нескольких меток ветвей case подряд не нарушает правило недопущения "провалов", поскольку во всех этих ветвях используется одна и та же последовательность операторов.

Расположение нескольких меток ветвей case подряд зачастую применяется в том случае, если у нескольких ветвей имеется общий код. Благодаря этому исключается излишнее дублирование кодовых последовательностей.

Вложенные операторы switch

Один оператор switch может быть частью последовательности операторов другого, внешнего оператора switch. И такой оператор switch называется *вложенным*. Константы выбора внутреннего и внешнего операторов switch могут содержать общие значения, не вызывая никаких конфликтов. Например, следующий фрагмент кода является вполне допустимым.

```
switch(ch1) {
  case 'A': Console.WriteLine("Эта ветвь А — часть " +
                              "внешнего оператора switch.");
    switch(ch2) {
      case 'A':
        Console.WriteLine("Эта ветвь А — часть " +
                          "внутреннего оператора switch");
        break;
      case 'B': // ...
    } // конец внутреннего оператора switch
    break;
  case 'B': // ...
```

Оператор цикла for

Оператор for уже был представлен в главе 2, а здесь он рассматривается более подробно. Вас должны приятно удивить эффективность и гибкость этого оператора. Прежде всего, обратимся к самым основным и традиционным формам оператора for.

Ниже приведена общая форма оператора for для повторного выполнения единственного оператора.

```
for(инициализация; условие; итерация) оператор;
```

А вот как выглядит его форма для повторного выполнения кодового блока:

```
for(инициализация; условие; итерация)
{
  последовательность операторов;
}
```

где *инициализация*, как правило, представлена оператором присваивания, задающим первоначальное значение переменной, которая выполняет роль счетчика и управляет циклом; *условие* — это логическое выражение, определяющее необходимость повторения цикла; а итерация — выражение, определяющее величину, на которую должно изменяться значение переменной, управляющей циклом, при каждом повторе цикла. Обратите внимание на то, что эти три основные части оператора цикла for должны быть разделены точкой с запятой. Выполнение цикла for будет продолжаться до тех пор, пока проверка условия дает истинный результат. Как только эта проверка даст ложный результат, цикл завершится, а выполнение программы будет продолжено с оператора, следующего после цикла for.

Цикл for может продолжаться как в положительном, так и в отрицательном направлении, изменяя значение переменной управления циклом на любую величину. В приведенном ниже примере программы выводятся числа, постепенно уменьшающиеся от 100 до –100 на величину 5.

```
// Выполнение цикла for в отрицательном направлении.

using System;

class DecrFor {
  static void Main() {
    int x;

    for(x = 100; x > -100; x -= 5)
      Console.WriteLine(x);
  }
}
```

В отношении циклов `for` следует особо подчеркнуть, что условное выражение всегда проверяется в самом начале цикла. Это означает, что код в цикле может вообще не выполняться, если проверяемое условие с самого начала оказывается ложным. Рассмотрим следующий пример.

```
for(count=10; count < 5; count++)
  x += count; // этот оператор не будет выполняться
```

Данный цикл вообще не будет выполняться, поскольку первоначальное значение переменной `count`, которая им управляет, сразу же оказывается больше 5. Это означает, что условное выражение `count < 5` оказывается ложным с самого начала, т.е. еще до выполнения первого шага цикла.

Оператор цикла `for` — наиболее полезный для повторного выполнения операций известное число раз. В следующем примере программы используются два цикла `for` для выявления простых чисел в пределах от 2 до 20. Если число оказывается непростым, то выводится наибольший его множитель.

```
// Выяснить, является ли число простым. Если оно
// непростое, вывести наибольший его множитель.

using System;

class FindPrimes {
  static void Main() {
    int num;
    int i;
    int factor;
    bool isprime;

    for(num = 2; num < 20; num++) {
      isprime = true;
      factor = 0;

      // Выяснить, делится ли значение переменной num нацело.
      for(i=2; i <= num/2; i++) {
        if((num % i) == 0) {

          // Значение переменной num делится нацело.
          // Следовательно, это непростое число.
          isprime = false;
          factor = i;
        }
```

```
      }

      if(isprime)
        Console.WriteLine(num + " — простое число.");
      else
        Console.WriteLine("Наибольший множитель числа " + num +
                          " равен " + factor);
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
2 — простое число
3 — простое число
Наибольший множитель числа 4 равен 2
5 — простое число
Наибольший множитель числа 6 равен 3
7 — простое число
Наибольший множитель числа 8 равен 4
Наибольший множитель числа 9 равен 3
Наибольший множитель числа 10 равен 5
11 — простое число
Наибольший множитель числа 12 равен 6
13 — простое число
Наибольший множитель числа 14 равен 7
Наибольший множитель числа 15 равен 5
Наибольший множитель числа 16 равен 8
17 — простое число
Наибольший множитель числа 18 равен 9
19 — простое число
```

Некоторые разновидности оператора цикла `for`

Оператор цикла `for` относится к самым универсальным операторам языка C#, поскольку он допускает самые разные варианты своего применения. Некоторые разновидности оператора цикла `for` рассматриваются ниже.

Применение нескольких переменных управления циклом

В операторе цикла `for` разрешается использовать две или более переменных для управления циклом. В этом случае операторы инициализации и инкремента каждой переменной разделяются запятой. Рассмотрим следующий пример программы.

```
// Использовать запятые в операторе цикла for.

using System;

class Comma {
  static void Main() {
    int i, j;

    for(i=0, j=10; i < j; i++, j--)
      Console.WriteLine("i и j: " + i + " " + j);
  }
}
```

Выполнение этой программы дает следующий результат.

```
i и j: 0 10
i и j: 1 9
i и j: 2 8
i и j: 3 7
i и j: 4 6
```

В данном примере запятыми разделяются два оператора инициализации и еще два итерационных выражения. Когда цикл начинается, инициализируются обе переменные, `i` и `j`. Всякий раз, когда цикл повторяется, переменная `i` инкрементируется, а переменная `j` декрементируется. Применение нескольких переменных управления циклом нередко оказывается удобным, упрощая некоторые алгоритмы. Теоретически в операторе цикла `for` может присутствовать любое количество операторов инициализации и итерации, но на практике цикл получается слишком громоздким, если применяется более двух подобных операторов.

Ниже приведен практический пример применения нескольких переменных управления циклом в операторе `for`. В этом примере программы используются две переменные управления одним циклом `for` для выявления наибольшего и наименьшего множителя целого числа (в данном случае — 100). Обратите особое внимание на условие окончания цикла. Оно опирается на обе переменные управления циклом.

```
// Использовать запятые в операторе цикла for для
// выявления наименьшего и наибольшего множителя числа.

using System;

class Comma {
  static void Main() {
    int i, j;
    int smallest, largest;
    int num;

    num = 100;

    smallest = largest = 1;

    for(i=2, j=num/2; (i <= num/2) & (j >= 2); i++, j--) {

      if((smallest == 1) & ((num % i) == 0))
        smallest = i;

      if((largest == 1) & ((num % j) == 0))
        largest = j;

    }

    Console.WriteLine("Наибольший множитель: " + largest);
    Console.WriteLine("Наименьший множитель: " + smallest);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Наибольший множитель: 50
Наименьший множитель: 2
```

Благодаря применению двух переменных управления циклом удается выявить наименьший и наибольший множители числа в одном цикле `for`. В частности, управляющая переменная `i` служит для выявления наименьшего множителя. Первоначально ее значение устанавливается равным 2 и затем инкрементируется до тех пор, пока не превысит половину значения переменной `num`. А управляющая переменная `j` служит для выявления наибольшего множителя. Ее значение первоначально устанавливается равным половине значения переменной `num` и затем декрементируется до тех пор, пока не станет меньше 2. Цикл продолжает выполняться до тех пор, пока обе переменные, `i` и `j`, не достигнут своих конечных значений. По завершении цикла оба множителя оказываются выявленными.

Условное выражение

Условным выражением, управляющим циклом `for`, может быть любое действительное выражение, дающее результат типа `bool`. В него не обязательно должна входить переменная управления циклом. В следующем примере программы управление циклом `for` осуществляется с помощью значения переменной `done`.

```
// Условием выполнения цикла может служить любое выражение типа bool.

using System;

class forDemo {
  static void Main() {
    int i, j;
    bool done = false;

    for(i=0, j=100; !done; i++, j--) {
      if(i*i >= j) done = true;
        Console.WriteLine("i, j: " + i + " " + j);
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
i, j: 0 100
i, j: 1 99
i, j: 2 98
i, j: 3 97
i, j: 4 96
i, j: 5 95
i, j: 6 94
i, j: 7 93
i, j: 8 92
i, j: 9 91
i, j: 10 90
```

В данном примере цикл `for` повторяется до тех пор, пока значение переменной `done` типа не окажется истинным (`true`). Истинное значение переменной `done` устанавливается в цикле, когда квадрат значения переменной `i` оказывается больше или равным значению переменной `j`.

Отсутствующие части цикла

Ряд интересных разновидностей цикла `for` получается в том случае, если оставить пустыми отдельные части определения цикла. В C# допускается оставлять пустыми любые или же все части инициализации, условия и итерации в операторе цикла `for`. В качестве примера рассмотрим такую программу.

```
// Отдельные части цикла for могут оставаться пустыми.

using System;

class Empty {
  static void Main() {
    int i;

    for(i = 0; i < 10; ) {
      Console.WriteLine("Проход №" + i);
      i++; // инкрементировать переменную управления циклом
    }
  }
}
```

В данном примере итерационное выражение в определении цикла `for` оказывается пустым, т.е. оно вообще отсутствует. Вместо этого переменная `i`, управляющая циклом, инкрементируется в теле самого цикла. Это означает, что всякий раз, когда цикл повторяется, значение переменной `i` проверяется на равенство числу 10, но никаких других действий при этом не происходит. А поскольку переменная `i` инкрементируется в теле цикла, то сам цикл выполняется обычным образом, выводя приведенный ниже результат.

```
Проход №0
Проход №1
Проход №2
Проход №3
Проход №4
Проход №5
Проход №6
Проход №7
Проход №8
Проход №9
```

В следующем примере программы из определения цикла `for` исключена инициализирующая часть.

```
// Исключить еще одну часть из определения цикла for.

using System;

class Empty2 {
  static void Main() {
    int i;

    i = 0; // исключить инициализацию из определения цикла
    for(; i < 10; ) {
      Console.WriteLine("Проход №" + i);
```

```
      i++; // инкрементировать переменную управления циклом
    }
  }
}
```

В данном примере переменная `i` инициализируется перед началом цикла, а не в самом цикле `for`. Как правило, переменная управления циклом инициализируется в цикле `for`. Выведение инициализирующей части за пределы цикла обычно делается лишь в том случае, если первоначальное значение данной переменной получается в результате сложного процесса, который нецелесообразно вводить в операторе цикла `for`.

Бесконечный цикл

Если оставить пустым выражение условия в операторе цикла `for`, то получится *бесконечный цикл*, т.е. такой цикл, который никогда не заканчивается. В качестве примера в следующем фрагменте кода показано, каким образом в C# обычно создается бесконечный цикл.

```
for(;;) // цикл, намеренно сделанный бесконечным
{
  //...
}
```

Этот цикл будет выполняться бесконечно. Несмотря на то что бесконечные циклы требуются для решения некоторых задач программирования, например при разработке командных процессоров операционных систем, большинство так называемых "бесконечных" циклов на самом деле представляет собой циклы со специальными требованиями к завершению. (Подробнее об этом — в разделе "Применение оператора `break` для выхода из цикла" далее в этой главе.)

Циклы без тела

В C# допускается оставлять пустым тело цикла `for` или любого другого цикла, поскольку *пустой оператор* с точки зрения синтаксиса этого языка считается действительным. Циклы без тела нередко оказываются полезными. Например, в следующей программе цикл без тела служит для получения суммы чисел от 1 до 5.

```
// Тело цикла может быть пустым.

using System;

class Empty3 {
  static void Main() {
   int i;
   int sum = 0;

   // получить сумму чисел от 1 до 5
   for(i = 1; i <= 5; sum += i++);

   Console.WriteLine("Сумма равна " + sum);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Сумма равна 15
```

Обратите внимание на то, что процесс суммирования выполняется полностью в операторе цикла `for`, и для этого тело цикла не требуется. В этом цикле особое внимание обращает на себя итерационное выражение.

```
sum += i++
```

Подобные операторы не должны вас смущать. Они часто встречаются в программах, профессионально написанных на C#, и становятся вполне понятными, если разобрать их по частям. Дословно приведенный выше оператор означает следующее: сложить со значением переменной `sum` результат суммирования значений переменных `sum` и `i`, а затем инкрементировать значение переменной `i`. Следовательно, данный оператор равнозначен следующей последовательности операторов.

```
sum = sum + i;
i++;
```

Объявление управляющих переменных в цикле `for`

Нередко переменная, управляющая циклом `for`, требуется только для выполнения самого цикла и нигде больше не используется. В таком случае управляющую переменную можно объявить в инициализирующей части оператора цикла `for`. Например, в приведенной ниже программе вычисляется сумма и факториал чисел от 1 до 5, а переменная `i`, управляющая циклом `for`, объявляется в этом цикле.

```
// Объявить переменную управления циклом в самом цикле for.

using System;

class ForVar {
  static void Main() {
    int sum = 0;
    int fact = 1;

    // вычислить факториал чисел от 1 до 5
    for(int i = 1; i <= 5; i++) {
      sum += i; // Переменная i действует в цикле.
      fact *= i;
    }

    // А здесь переменная i недоступна.

    Console.WriteLine("Сумма равна " + sum);
    Console.WriteLine("Факториал равен " + fact);
  }
}
```

Объявляя переменную в цикле `for`, не следует забывать о том, что область действия этой переменной ограничивается пределами оператора цикла `for`. Это означает, что за пределами цикла действие данной переменной прекращается. Так, в приведенном выше примере переменная `i` оказывается недоступной за пределами цикла `for`. Для того чтобы использовать переменную управления циклом в каком-нибудь другом месте программы, ее нельзя объявлять в цикле `for`.

Прежде чем переходить к чтению следующего материала, поэкспериментируйте с собственными разновидностями оператора цикла `for`. В ходе эксперимента вы непременно обнаружите замечательные свойства этого оператора цикла.

Оператор цикла `while`

Еще одним оператором цикла в C# является оператор `while`. Ниже приведена общая форма этого оператора:

```
while (условие) оператор;
```

где *оператор* — это единственный оператор или же блок операторов, а *условие* означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле *оператор* выполняется до тех пор, пока *условие* истинно. Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла.

Ниже приведен простой пример программы, в которой цикл `while` используется для вычисления порядка величины целого числа.

```
// Вычислить порядок величины целого числа.

using System;

class WhileDemo {
  static void Main() {
    int num;
    int mag;

    num = 435679;
    mag = 0;

    Console.WriteLine("Число: " + num);

    while(num > 0) {
      mag++;
      num = num / 10;
    };

    Console.WriteLine("Порядок величины: " + mag);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Число: 435679
Порядок величины: 6
```

Приведенный выше цикл `while` действует следующим образом. Сначала проверяется значение переменной `num`. Если оно больше нуля, то переменная `mag`, выполняющая роль счетчика порядка величины, инкрементируется, а значение переменной `num` делится на 10. Цикл повторяется до тех пор, пока значение переменной `num` остается больше нуля. Как только оно окажется равным нулю, цикл завершается, а в переменной `mag` остается порядок величины первоначального числового значения.

Как и в цикле `for`, в цикле `while` проверяется условное выражение, указываемое в самом начале цикла. Это означает, что код в теле цикла может вообще не выполняться, а также избавляет от необходимости выполнять отдельную проверку перед самим циклом. Данное свойство цикла `while` демонстрируется в следующем примере программы, где вычисляются целые степени числа 2 от 0 до 9.

```
// Вычислить целые степени числа 2.

using System;

class Power {
  static void Main() {
    int e;
    int result;

    for(int i=0; i < 10; i++) {
      result = 1;
      e = i;

      while(e > 0) {
        result *= 2;
        e--;
      }

      Console.WriteLine("2 в степени " + i + " равно " + result);
    }
  }
}
```

Результат выполнения этой программы приведен ниже.

```
2 в степени 0 равно 1
2 в степени 1 равно 2
2 в степени 2 равно 4
2 в степени 3 равно 8
2 в степени 4 равно 16
2 в степени 5 равно 32
2 в степени 6 равно 64
2 в степени 7 равно 128
2 в степени 8 равно 256
2 в степени 9 равно 512
```

Обратите внимание на то, что цикл `while` выполняется только в том случае, если значение переменной `e` больше нуля. А когда оно равно нулю, как это имеет место на первом шаге цикла `for`, цикл `while` пропускается.

Оператор цикла do-while

Третьим оператором цикла в C# является оператор `do-while`. В отличие от операторов цикла `for` и `while`, в которых условие проверялось в самом начале цикла, в операторе `do-while` условие выполнения цикла проверяется в самом его конце. Это означает, что цикл `do-while` всегда выполняется хотя бы один раз. Ниже приведена общая форма оператора цикла `do-while`.

```
do {
   операторы;
} while (условие);
```

При наличии лишь одного оператора фигурные скобки в данной форме записи необязательны. Тем не менее они зачастую используются для того, чтобы сделать кон-

струкцию do-while более удобочитаемой и не путать ее с конструкцией цикла while. Цикл do-while выполняется до тех пор, пока условное выражение истинно.

В приведенном ниже примере программы цикл do-while используется для представления отдельных цифр целого числа в обратном порядке.

```
// Отобразить цифры целого числа в обратном порядке.

using System;

class DoWhileDemo {
  static void Main() {
    int num;
    int nextdigit;

    num = 198;

    Console.WriteLine("Число: " + num);

    Console.Write("Число в обратном порядке: ");

    do {
      nextdigit = num % 10;
      Console.Write(nextdigit);
      num = num / 10;
    } while(num > 0);

    Console.WriteLine();
  }
}
```

Выполнение этой программы дает следующий результат.

```
Число: 198
Число в обратном порядке: 891
```

Приведенный выше цикл действует следующим образом. На каждом его шаге крайняя слева цифра получается в результате расчета остатка от деления целого числа (значения переменной num) на 10. Полученная в итоге цифра отображается. Далее значение переменной num делится на 10. А поскольку это целочисленное деление, то в его результате крайняя слева цифра отбрасывается. Этот процесс повторяется до тех пор, пока значение переменной num не достигнет нуля.

Оператор цикла `foreach`

Оператор цикла foreach служит для циклического обращения к элементам *коллекции*, которая представляет собой группу объектов. В C# определено несколько видов коллекций, к числу которых относится массив. Подробнее о цикле foreach речь пойдет в главе 7, где рассматриваются массивы.

Применение оператора `break` для выхода из цикла

С помощью оператора break можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку

условия цикла. Когда в теле цикла встречается оператор `break`, цикл завершается, а выполнение программы возобновляется с оператора, следующего после этого цикла. Рассмотрим простой пример программы.

```
// Применить оператор break для выхода из цикла.

using System;

class BreakDemo {
  static void Main() {

    // Использовать оператор break для выхода из этого цикла.
    for(int i=-10; i <= 10; i++) {
      if(i > 0) break; // завершить цикл, как только значение
                       // переменной i станет положительным
      Console.Write(i + " ");
    }
    Console.WriteLine("Готово!");
  }
}
```

Выполнение этой программы дает следующий результат.

```
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 Готово!
```

Как видите, цикл `for` организован для выполнения в пределах от –10 до 10, но, несмотря на это, оператор `break` прерывает его раньше, когда значение переменной `i` становится положительным.

Оператор `break` можно применять в любом цикле, предусмотренном в C#. В качестве примера ниже приведена версия предыдущей программы, переделанная с целью использовать цикл `do-while`.

```
// Применить оператор break для выхода из цикла do-while.

using System;

class BreakDemo2 {
  static void Main() {
    int i;

    i = -10;
    do {
    if(i > 0) break;
    Console.Write(i + " ");
    i++;
    } while(i <= 10);

    Console.WriteLine("Готово!");
  }
}
```

А теперь рассмотрим более практический пример применения оператора `break`. В приведенной ниже программе выявляется наименьший множитель числа.

```
// Выявить наименьший множитель числа.

using System;
```

```
class FindSmallestFactor {
  static void Main() {
    int factor = 1;
    int num = 1000;

    for(int i=2; i <= num/i; i++) {
      if((num%i) == 0) {
        factor = i;
        break; // прервать цикл, как только будет
               // выявлен наименьший множитель числа
      }
    }

    Console.WriteLine("Наименьший множитель равен " + factor);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Наименьший множитель равен 2
```

Оператор `break` прерывает выполнение цикла `for`, как только будет выявлен наименьший множитель числа. Благодаря такому применению оператора `break` исключается опробование любых других значений после выявления наименьшего множителя числа, а следовательно, и неэффективное выполнение кода.

Если оператор `break` применяется в целом ряде вложенных циклов, то он прерывает выполнение только самого внутреннего цикла. В качестве примера рассмотрим следующую программу.

```
// Применить оператор break во вложенных циклах.
using System;

class BreakNested {
  static void Main() {

    for(int i=0; i<3; i++) {
      Console.WriteLine("Подсчет во внешнем цикле: " + i);
      Console.Write("    Подсчет во внутреннем цикле: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // прервать цикл, если t равно 10
        Console.Write(t + " ");
        t++;
      }
    Console.WriteLine();
    }
    Console.WriteLine("Циклы завершены.");
  }
}
```

Выполнение этой программы дает следующий результат.

```
Подсчет во внешнем цикле: 0
    Подсчет во внутреннем цикле: 0 1 2 3 4 5 6 7 8 9
Подсчет во внешнем цикле: 1
```

```
    Подсчет во внутреннем цикле: 0 1 2 3 4 5 6 7 8 9
Подсчет во внешнем цикле: 2
    Подсчет во внутреннем цикле: 0 1 2 3 4 5 6 7 8 9
Циклы завершены
```

Как видите, оператор `break` из внутреннего цикла вызывает прерывание только этого цикла, а на выполнение внешнего цикла он не оказывает никакого влияния.

В отношении оператора `break` необходимо также иметь в виду следующее. Во-первых, в теле цикле может присутствовать несколько операторов `break`, но применять их следует очень аккуратно, поскольку чрезмерное количество операторов `break` обычно приводит к нарушению нормальной структуры кода. И во-вторых, оператор `break`, выполняющий выход из оператора `switch`, оказывает воздействие только на этот оператор, но не на объемлющие его циклы.

Применение оператора `continue`

С помощью оператора `continue` можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом. Оператор `continue` осуществляет принудительный переход к следующему шагу цикла, пропуская любой код, оставшийся невыполненным. Таким образом, оператор `continue` служит своего рода дополнением оператора `break`. В приведенном ниже примере программы оператор `continue` используется в качестве вспомогательного средства для вывода четных чисел в пределах от 0 до 100.

```
// Применить оператор continue.

using System;

class ContDemo {
  static void Main() {

    // вывести четные числа от 0 до 100.
    for(int i = 0; i <= 100; i++) {
      if((i%2) != 0) continue; // перейти к следующему шагу итерации
      Console.WriteLine(i);
    }
  }
}
```

В данном примере выводятся только четные числа, поскольку при обнаружении нечетного числа шаг итерации цикла завершается преждевременно в обход вызова метода `WriteLine()`.

В циклах `while` и `do-while` оператор `continue` вызывает передачу управления непосредственно условному выражению, после чего продолжается процесс выполнения цикла. А в цикле `for` сначала вычисляется итерационное выражение, затем условное выражение, после чего цикл продолжается.

Оператор `continue` редко находит удачное применение, в частности, потому, что в C# предоставляется богатый набор операторов цикла, удовлетворяющих большую часть прикладных потребностей. Но в тех особых случаях, когда требуется преждевременное прерывание шага итерации цикла, оператор `continue` предоставляет структурированный способ осуществления такого прерывания.

Оператор return

Оператор return организует возврат из метода. Его можно также использовать для возврата значения. Более подробно он рассматривается в главе 6.

Оператор goto

Имеющийся в C# оператор goto представляет собой оператор безусловного перехода. Когда в программе встречается оператор goto, ее выполнение переходит непосредственно к тому месту, на которое указывает этот оператор. Он уже давно "вышел из употребления" в программировании, поскольку способствует созданию "макаронного" кода. Тем не менее оператор goto все еще находит применение — иногда даже эффективное. В этой книге не делается никаких далеко идущих выводов относительно правомочности использования оператора goto для управления программой. Следует, однако, подчеркнуть, что этому оператору трудно найти полезное применение, и поэтому он не особенно нужен для полноты языка программирования. Хотя в некоторых случаях он оказывается удобным и дает определенные преимущества, если используется благоразумно. В силу этих причин оператор goto упоминается только в данном разделе книги. Главный недостаток оператора goto с точки зрения программирования заключается в том, что он вносит в программу беспорядок и делает ее практически неудобочитаемой. Но иногда применение оператора goto может, скорее, прояснить, чем запутать ход выполнения программы.

Для выполнения оператора goto требуется *метка* — действительный в C# идентификатор с двоеточием. Метка должна находиться в том же методе, где и оператор goto, а также в пределах той же самой области действия. В приведенном ниже примере программы цикл суммирования чисел от 1 до 100 организован с помощью оператора goto и соответствующей метки.

```
x = 1;
loop1:
  x++;
  if(x < 100) goto loop1;
```

Кроме того, оператор goto может быть использован для безусловного перехода к ветви case или default в операторе switch. Формально ветви case или default выполняют в операторе switch роль меток. Поэтому они могут служить адресатами оператора goto. Тем не менее оператор goto должен выполняться в пределах оператора switch. Это означает, что его нельзя использовать как внешнее средство для безусловного перехода в оператор switch. В приведенном ниже примере программы демонстрируется применение оператора goto в операторе switch.

```
// Применить оператор goto в операторе switch.

using System;

class SwitchGoto {
  static void Main() {

    for(int i=1; i < 5; i++) {
      switch(i) {
        case 1:
```

```
          Console.WriteLine("В ветви case 1");
          goto case 3;
        case 2:
          Console.WriteLine("В ветви case 2");
          goto case 1;
        case 3:
          Console.WriteLine("В ветви case 3");
          goto default;
        default:
          Console.WriteLine("В ветви default");
          break;
      }

      Console.WriteLine();
    }

// goto case 1; // Ошибка! Безусловный переход к оператору switch недопустим.
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
В ветви case 1
В ветви case 3
В ветви default

В ветви case 2
В ветви case 1
В ветви case 3
В ветви default

В ветви case 3
В ветви default

В ветви default
```

Обратите внимание на то, как оператор `goto` используется в операторе `switch` для перехода к другим его ветвям `case` или к ветви `default`. Обратите также внимание на то, что ветви `case` не оканчиваются оператором `break`. Благодаря тому что оператор `goto` препятствует последовательному переходу от одной ветви `case` к другой, упоминавшееся ранее правило недопущения "провалов" не нарушается, а следовательно, необходимость в применении оператора `break` в данном случае отпадает. Но как пояснялось выше, оператор `goto` нельзя использовать как внешнее средство для безусловного перехода к оператору `switch`. Так, если удалить символы комментария в начале следующей строки:

```
// goto case 1; // Ошибка! Безусловный переход к оператору switch недопустим.
```

приведенная выше программа не будет скомпилирована. Откровенно говоря, применение оператора `goto` в операторе `switch`, в общем, не рекомендуется как стиль программирования, хотя в ряде особых случаев это может принести определенную пользу.

Ниже приведен один из полезных примеров применения оператора `goto` для выхода из глубоко вложенной части программы.

```
// Продемонстрировать практическое применение оператора goto.

using System;

class Use_goto {
  static void Main() {
    int i=0, j=0, k=0;

    for(i=0; i < 10; i++) {
      for(j=0; j < 10; j++ ) {
        for(k=0; k < 10; k++) {
          Console.WriteLine("i, j, k: " + i + " " + j +
                            " " + k);
          if(k == 3) goto stop;
        }
      }
    }

  stop:
       Console.WriteLine("Остановлено! i, j, k: " + i +
                         ", " + j + " " + k);
  }
}
```

Выполнение этой программы дает следующий результат.

```
i, j, k: 0 0 0
i, j, k: 0 0 1
i, j, k: 0 0 2
i, j, k: 0 0 3
Остановлено! i, j, k: 0, 0 3
```

Если бы не оператор `goto`, то в приведенной выше программе пришлось бы прибегнуть к трем операторам `if` и `break`, чтобы выйти из глубоко вложенной части этой программы. В данном случае оператор `goto` действительно упрощает код. И хотя приведенный выше пример служит лишь для демонстрации применения оператора `goto`, вполне возможны ситуации, в которых этот оператор может на самом деле оказаться полезным.

И последнее замечание: как следует из приведенного выше примера, из кодового блока можно выйти непосредственно, но войти в него так же непосредственно нельзя.

6 ГЛАВА

Введение в классы, объекты и методы

Эта глава служит введением в классы. Класс составляет основу языка C#, поскольку он определяет характер объекта. Кроме того, класс служит основанием для объектно-ориентированного программирования (ООП). В пределах класса определяются данные и код. А поскольку классы и объекты относятся к основополагающим элементам C#, то для их рассмотрения требуется не одна глава книги. В данной главе рассмотрение классов и объектов начинается с их главных особенностей.

Основные положения о классах

Классы использовались в примерах программ с самого начала этой книги. Разумеется, это были лишь самые простые классы, что не позволяло выгодно воспользоваться большинством их возможностей. На самом же деле классы намного более эффективны, чем это следует из приведенных ранее примеров их ограниченного применения.

Начнем рассмотрение классов с основных положений. *Класс* представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В C# используется спецификация класса для построения *объектов*, которые являются *экземплярами* класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

Общая форма определения класса

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными. Если самые простые классы могут содержать только код или только данные, то большинство настоящих классов содержит и то и другое.

Вообще говоря, данные содержатся в *членах данных*, определяемых классом, а код — в *функциях-членах*. Следует сразу же подчеркнуть, что в C# предусмотрено несколько разновидностей членов данных и функций-членов. Например, к членам данных, называемым также *полями*, относятся переменные экземпляра и статические переменные, а к функциям-членам — методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства. Ограничимся пока что рассмотрением самых основных компонентов класса: переменных экземпляра и методов. А далее в этой главе будут представлены конструкторы и деструкторы. Об остальных разновидностях членов класса речь пойдет в последующих главах.

Класс создается с помощью ключевого слова `class`. Ниже приведена общая форма определения простого класса, содержащая только переменные экземпляра и методы.

```
class имя_класса {
   // Объявление переменных экземпляра.
   доступ тип переменная1;
   доступ тип переменная2;
   //...
   доступ тип переменнаяN;

   // Объявление методов.
   доступ возвращаемый_тип метод1(параметры) {
      // тело метода
   }
   доступ возвращаемый_тип метод2(параметры) {
      // тело метода
   }

      //...
   доступ возвращаемый_тип методN(параметры) {
      // тело метода
   }
}
```

Обратите внимание на то, что перед каждым объявлением переменной и метода указывается *доступ*. Это спецификатор доступа, например `public`, определяющий порядок доступа к данному члену класса. Как упоминалось в главе 2, члены класса могут быть как закрытыми (`private`) в пределах класса, так открытыми (`public`), т.е. более доступными. Спецификатор доступа определяет *тип* разрешенного доступа. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса. В примерах программ, приведенных в этой главе, все члены, за исключением метода `Main()`, обозначаются как открытые (`public`). Это означает, что их можно использовать во всех остальных фрагментах кода — даже в тех, что определены за пределами класса. Мы еще вернемся к обсуждению спецификаторов доступа в главе 8.

ПРИМЕЧАНИЕ

Помимо спецификатора доступа, в объявлении члена класса могут также присутствовать один или несколько модификаторов. О модификаторах речь пойдет далее в этой главе.

Несмотря на отсутствие соответствующего правила в синтаксисе C#, правильно сконструированный класс должен определять одну и только одну логическую сущность. Например, класс, в котором хранятся Ф.И.О. и номера телефонов, обычно не содержит сведения о фондовом рынке, среднем уровне осадков, циклах солнечных пятен или другую информацию, не связанную с перечисляемыми фамилиями. Таким образом, в правильно сконструированном классе должна быть сгруппирована логически связанная информация. Если же в один и тот же класс помещается логически несвязанная информация, то структурированность кода быстро нарушается.

Классы, использовавшиеся в приведенных ранее примерах программ, содержали только один метод: `Main()`. Но в представленной выше общей форме определения класса метод `Main()` не указывается. Этот метод требуется указывать в классе лишь в том случае, если программа начинается с данного класса.

Определение класса

Для того чтобы продемонстрировать классы на конкретных примерах, разработаем постепенно класс, инкапсулирующий информацию о зданиях, в том числе о домах, складских помещениях, учреждениях и т.д. В этом классе (назовем его `Building`) будут храниться три элемента информации о зданиях: количество этажей, общая площадь и количество жильцов.

Ниже приведен первый вариант класса `Building`. В нем определены три переменные экземпляра: `Floors`, `Area` и `Occupants`. Как видите, в классе `Building` вообще отсутствуют методы. Это означает, что в настоящий момент этот класс состоит только из данных. (Впоследствии в него будут также введены методы.)

```
class Building {
  public int Floors;     // количество этажей
  public int Area;       // общая площадь здания
  public int Occupants; // количество жильцов
}
```

Переменные экземпляра, определенные в классе `Building`, демонстрируют общий порядок объявления переменных экземпляра. Ниже приведена общая форма для объявления переменных экземпляра:

доступ тип имя_переменной;

где *доступ* обозначает вид доступа; *тип* — конкретный тип переменной, а *имя_переменной* — имя, присваиваемое переменной. Следовательно, за исключением спецификатора доступа, переменная экземпляра объявляется таким же образом, как и локальная переменная. Все переменные объявлены в классе `Building` с предваряющим их модификатором доступа `public`. Как пояснялось выше, благодаря этому они становятся доступными за пределами класса `Building`.

Определение `class` обозначает создание нового типа данных. В данном случае новый тип данных называется `Building`. С помощью этого имени могут быть объявлены

объекты типа Building. Не следует, однако, забывать, что объявление class лишь описывает тип, но не создает конкретный объект. Следовательно, в приведенном выше фрагменте кода объекты типа Building не создаются.

Для того чтобы создать конкретный объект типа Building, придется воспользоваться следующим оператором.

```
Building house = new Building(); // создать объект типа Building
```

После выполнения этого оператора объект house станет экземпляром класса Building, т.е. обретет "физическую" реальность. Не обращайте пока что внимание на отдельные составляющие данного оператора.

Всякий раз, когда получается экземпляр класса, создается также объект, содержащий собственную копию каждой переменной экземпляра, определенной в данном классе. Таким образом, каждый объект типа Building будет содержать свои копии переменных экземпляра Floors, Area и Occupants. Для доступа к этим переменным служит оператор доступа к члену класса, который принято называть *оператором-точкой*. Оператор-точка связывает имя объекта с именем члена класса. Ниже приведена общая форма оператора-точки.

```
объект.член
```

В этой форме *объект* указывается слева, а *член* — справа. Например, присваивание значения 2 переменной Floors объекта house осуществляется с помощью следующего оператора.

```
house.Floors = 2;
```

В целом, оператор-точка служит для доступа к переменным экземпляра и методам. Ниже приведен полноценный пример программы, в которой используется класс Building.

```
// Программа, в которой используется класс Building.

using System;

class Building {
  public int Floors;    // количество этажей
  public int Area;      // общая площадь здания
  public int Occupants; // количество жильцов
}

// В этом классе объявляется объект типа Building.
class BuildingDemo {
  static void Main() {
    Building house = new Building(); // создать объект типа Building
    int areaPP; // площадь на одного человека

    // Присвоить значения полям в объекте house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Вычислить площадь на одного человека.
    areaPP = house.Area / house.Occupants;
```

```
    Console.WriteLine("Дом имеет:\n  " +
                      house.Floors + " этажа\n  " +
                      house.Occupants + " жильца\n  " +
                      house.Area +
                      " кв. футов общей площади, из них\n  " +
                      areaPP + " приходится на одного человека");
  }
}
```

Эта программа состоит из двух классов: `Building` и `BuildingDemo`. В классе `BuildingDemo` сначала создается экземпляр `house` класса `Building` с помощью метода `Main()`, а затем в коде метода `Main()` осуществляется доступ к переменным экземпляра `house` для присваивания им значений и последующего использования этих значений. Следует особо подчеркнуть, что `Building` и `BuildingDemo` — это два совершенно отдельных класса. Единственная взаимосвязь между ними состоит в том, что в одном из них создается экземпляр другого. Но, несмотря на то, что это раздельные классы, у кода из класса `BuildingDemo` имеется доступ к членам класса `Building`, поскольку они объявлены как открытые (`public`). Если бы при их объявлении не был указан спецификатор доступа `public`, то доступ к ним ограничивался бы пределами `Building`, а следовательно, их нельзя было бы использовать в классе `BuildingDemo`.

Допустим, что исходный текст приведенной выше программы сохранен в файле `UseBuilding.cs`. В результате ее компиляции создается файл `UseBuilding.exe`. При этом оба класса, `Building` и `BuildingDemo`, автоматически включаются в состав исполняемого файла. При выполнении данной программы выводится следующий результат.

```
Дом имеет:
  2 этажа
  4 жильца
  2500 кв. футов общей площади, из них
  625 приходится на одного человека
```

Но классам `Building` и `BuildingDemo` совсем не обязательно находиться в одном и том же исходном файле. Каждый из них можно поместить в отдельный файл, например `Building.cs` и `BuildingDemo.cs`, а компилятору C# достаточно сообщить, что оба файла должны быть скомпилированы вместе. Так, если разделить рассматриваемую здесь программу на два таких файла, для ее компилирования можно воспользоваться следующей командной строкой.

```
csc Building.cs BuildingDemo.cs
```

Если вы пользуетесь интегрированной средой разработки Visual Studio, то вам нужно ввести оба упомянутых выше файла в свой проект и затем скомпоновать их.

Прежде чем двигаться дальше, рассмотрим следующий основополагающий принцип: у каждого объекта имеются свои копии переменных экземпляра, определенных в его классе. Следовательно, содержимое переменных в одном объекте может отличаться от их содержимого в другом объекте. Между обоими объектами не существует никакой связи, за исключением того факта, что они являются объектами одного и того же типа. Так, если имеются два объекта типа `Building`, то у каждого из них своя копия переменных `Floors`, `Area` и `Occupants`, а их содержимое в обоих объектах может отличаться. Этот факт демонстрируется в следующей программе.

```
// В этой программе создаются два объекта типа Building.

using System;

  class Building {
    public int Floors;    // количество этажей
    public int Area;      // общая площадь здания
    public int Occupants; // количество жильцов
}

// В этом классе объявляются два объекта типа Building.
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();

    int areaPP; // площадь на одного человека

    // Присвоить значения полям в объекте house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Присвоить значения полям в объекте office.
    office.Occupants = 25;
    office.Area = 4200;
    office.Floors = 3;

// Вычислить площадь на одного человека в жилом доме.
    areaPP = house.Area / house.Occupants;

    Console.WriteLine("Дом имеет:\n  " +
                      house.Floors + " этажа\n  " +
                      house.Occupants + " жильца\n  " +
                      house.Area +
                      " кв. футов общей площади, из них\n  " +
                      areaPP + " приходится на одного человека");

    // Вычислить площадь на одного человека в учреждении.
    areaPP = office.Area / office.Occupants;
     Console.WriteLine("Учреждение имеет:\n  " +
                       office.Floors + " этажа\n  " +
                       office.Occupants + " работников\n  " +
                       office.Area +
                       " кв. футов общей площади, из них\n  " +
                       areaPP + " приходится на одного человека");
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Дом имеет:
  2 этажа
  4 жильца
  2500 кв. футов общей площади, из них
  625 приходится на одного человека
```

```
Учреждение имеет:
  3 этажа
  25 работников
  4200 кв. футов общей площади, из них
  168 приходится на одного человека
```

Как видите, данные из объекта house полностью отделены от данных, содержащихся в объекте office. Эта ситуация наглядно показана на рис. 6.1.

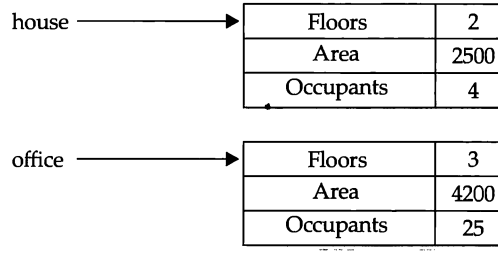


Рис. 6.1. Переменные экземпляра одного объекта полностью отделены от переменных экземпляра другого объекта

Создание объектов

В предыдущих примерах программ для объявления объекта типа Building использовалась следующая строка кода.

```
Building house = new Building();
```

Эта строка объявления выполняет три функции. Во-первых, объявляется переменная house, относящаяся к типу класса Building. Сама эта переменная не является объектом, а лишь переменной, которая может *ссылаться* на объект. Во-вторых, создается конкретная, физическая, копия объекта. Это делается с помощью оператора new. И наконец, переменной house присваивается ссылка на данный объект. Таким образом, после выполнения анализируемой строки объявленная переменная house ссылается на объект типа Building.

Оператор new динамически (т.е. во время выполнения) распределяет память для объекта и возвращает ссылку на него, которая затем сохраняется в переменной. Следовательно, в C# для объектов всех классов должна быть динамически распределена память.

Как и следовало ожидать, объявление переменной house можно отделить от создания объекта, на который она ссылается, следующим образом.

```
Building house; // объявить ссылку на объект
house = new Building(); // распределить память для объекта типа Building
```

В первой строке объявляется переменная house в виде ссылки на объект типа Building. Следовательно, house — это переменная, которая может ссылаться на объект, хотя сама она не является объектом. А во второй строке создается новый объект типа Building, и ссылка на него присваивается переменной house. В итоге переменная house оказывается связанной с данным объектом.

То обстоятельство, что объекты классов доступны по ссылке, объясняет, почему классы называются *ссылочными типами*. Главное отличие типов значений от ссылочных типов заключается в том, что именно содержит переменная каждого из этих типов. Так, переменная типа значения содержит конкретное значение. Например, во фрагменте кода

```
int x;
x = 10;
```

переменная x содержит значение 10, поскольку она относится к типу int, который является типом значения. Но в строке

```
Building house = new Building();
```

переменная house содержит не сам объект, а лишь ссылку на него.

Переменные ссылочного типа и присваивание

В операции присваивания переменные ссылочного типа действуют иначе, чем переменные типа значения, например типа int. Когда одна переменная типа значения присваивается другой, ситуация оказывается довольно простой. Переменная, находящаяся в левой части оператора присваивания, получает копию значения переменной, находящейся в правой части этого оператора. Когда же одна переменная ссылки на объект присваивается другой, то ситуация несколько усложняется, поскольку такое присваивание приводит к тому, что переменная, находящаяся в левой части оператора присваивания, ссылается на тот же самый объект, на который ссылается переменная, находящаяся в правой части этого оператора. Сам же объект не копируется. В силу этого отличия присваивание переменных ссылочного типа может привести к несколько неожиданным результатам. В качестве примера рассмотрим следующий фрагмент кода.

```
Building house1 = new Building();
Building house2 = house1;
```

На первый взгляд, переменные house1 и house2 ссылаются на совершенно разные объекты, но на самом деле это не так. Переменные house1 и house2, напротив, ссылаются на один и тот же объект. Когда переменная house1 присваивается переменой house2, то в конечном итоге переменная house2 просто ссылается на тот же самый объект, что и переменная house1. Следовательно, этим объектом можно оперировать с помощью переменной house1 или house2. Например, после очередного присваивания

```
house1.Area = 2600;
```

оба метода WriteLine()

```
Console.WriteLine(house1.Area);
Console.WriteLine(house2.Area);
```

выводят одно и то же значение: 2600.

Несмотря на то что обе переменные, house1 и house2, ссылаются на один и тот же объект, они никак иначе не связаны друг с другом. Например, в результате следующей последовательности операций присваивания просто изменяется объект, на который ссылается переменная house2.

```
Building house1 = new Building();
Building house2 = house1;
Building house3 = new Building();
house2 = house3; // теперь обе переменные, house2 и house3,
                 // ссылаются на один и тот же объект.
```

После выполнения этой последовательности операций присваивания переменная `house2` ссылается на тот же самый объект, что и переменная `house3`. А ссылка на объект в переменной `house1` не меняется.

Методы

Как пояснялось выше, переменные экземпляра и методы являются двумя основными составляющими классов. До сих пор класс `Building`, рассматриваемый здесь в качестве примера, содержал только данные, но не методы. Хотя классы, содержащие только данные, вполне допустимы, у большинства классов должны быть также методы. *Методы* представляют собой подпрограммы, которые манипулируют данными, определенными в классе, а во многих случаях они предоставляют доступ к этим данным. Как правило, другие части программы взаимодействуют с классом посредством его методов.

Метод состоит из одного или нескольких операторов. В грамотно написанном коде C# каждый метод выполняет только одну функцию. У каждого метода имеется свое имя, по которому он вызывается. В общем, методу в качестве имени можно присвоить любой действительный идентификатор. Следует, однако, иметь в виду, что идентификатор `Main()` зарезервирован для метода, с которого начинается выполнение программы. Кроме того, в качестве имен методов нельзя использовать ключевые слова C#.

В этой книге методы именуются в соответствии с условными обозначениями, принятыми в литературе по C#. В частности, после имени метода следуют круглые скобки. Так, если методу присвоено имя `GetVal`, то в тексте книги он упоминается в следующем виде: `GetVal()`. Такая форма записи помогает отличать имена методов от имен переменных при чтении книги.

Ниже приведена общая форма определения метода:

```
доступ возвращаемый_тип имя(список_параметров) {
      // тело метода
}
```

где *доступ* — это модификатор доступа, определяющий те части программы, из которых может вызываться метод. Как пояснялось выше, указывать модификатор доступа необязательно. Но если он отсутствует, то метод оказывается закрытым (`private`) в пределах того класса, в котором он объявляется. Мы будем пока что объявлять методы открытыми (`public`), чтобы вызывать их из любой другой части кода в программе. Затем *возвращаемый_тип* обозначает тип данных, возвращаемых методом. Этот тип должен быть действительным, в том числе и типом создаваемого класса. Если метод не возвращает значение, то в качестве возвращаемого для него следует указать тип `void`. Далее *имя* обозначает конкретное имя, присваиваемое методу. В качестве имени метода может служить любой допустимый идентификатор, не приводящий к конфликтам в текущей области объявлений. И наконец, *список_параметров* — это последовательность пар, состоящих из типа и идентификатора и разделенных запятыми. Параметры представляют собой переменные, получающие значение *аргументов*, передаваемых методу при его вызове. Если у метода отсутствуют параметры, то список параметров оказывается пустым.

Добавление метода в класс Building

Как пояснялось выше, методы класса, как правило, манипулируют данными класса и предоставляют доступ к ним. С учетом этого напомним, что в приведенных выше примерах в методе Main() вычислялась площадь на одного человека путем деления общей площади здания на количество жильцов. И хотя такой способ формально считается правильным, на самом деле он оказывается далеко не самым лучшим для организации подобного вычисления. Площадь на одного человека лучше всего вычислять в самом классе Building, просто потому, что так легче понять сам характер вычисления. Ведь площадь на одного человека зависит от значений в полях Area и Occupants, инкапсулированных в классе Building. Следовательно, данное вычисление может быть вполне произведено в самом классе Building. Кроме того, вводя вычисление площади на одного человека в класс Building, мы тем самым избавляем все программы, пользующиеся классом Building, от необходимости выполнять это вычисление самостоятельно. Благодаря этому исключается ненужное дублирование кода. И наконец, добавление в класс Building метода, вычисляющего площадь на одного человека, способствует улучшению его объектно-ориентированной структуры, поскольку величины, непосредственно связанные со зданием, инкапсулируются в классе Building.

Для того чтобы добавить метод в класс Building, достаточно указать его в области объявлений в данном классе. В качестве примера ниже приведен переработанный вариант класса Building, содержащий метод AreaPerPerson(), который выводит площадь, рассчитанную на одного человека в конкретном здании.

```
// Добавить метод в класс Building.

using System;

class Building {
  public int Floors;    // количество этажей
  public int Area;      // общая площадь здания
  public int Occupants; // количество жильцов

  // Вывести площадь на одного человека.
  public void AreaPerPerson() {
    Console.WriteLine("  " + Area / Occupants +
                      " приходится на одного человека");
  }
}

// Использовать метод AreaPerPerson().
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();

    // Присвоить значения полям в объекте house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

    // Присвоить значения полям в объекте office.
    office.Occupants = 25;
```

```
    office.Area = 4200;
    office.Floors = 3;

    Console.WriteLine("Дом имеет:\n  " +
                      house.Floors + " этажа\n  " +
                      house.Occupants + " жильца\n  " +
                      house.Area +
                      " кв. футов общей площади, из них");
    house.AreaPerPerson();

    Console.WriteLine();

    Console.WriteLine("Учреждение имеет:\n  " +
                      office.Floors + " этажа\n  " +
                      office.Occupants + " работников\n  " +
                      office.Area +
                      " кв. футов общей площади, из них");
    office.AreaPerPerson();
  }
}
```

Эта программа дает такой же результат, как и прежде.

```
Дом имеет:
  2 этажа
  4 жильца
  2500 кв. футов общей площади, из них
  625 приходится на одного человека

Учреждение имеет:
  3 этажа
  25 работников
  4200 кв. футов общей площади, из них
  168 приходится на одного человека
```

Рассмотрим основные элементы этой программы, начиная с метода `AreaPerPerson()`. Первая его строка выглядит следующим образом.

```
public void AreaPerPerson() {
```

В этой строке объявляется метод, именуемый `AreaPerPerson` и не имеющий параметров. Для него указывается тип `public`, а это означает, что его можно вызывать из любой другой части программы. Метод `AreaPerPerson()` возвращает пустое значение типа `void`, т.е. он практически ничего не возвращает вызывающей части программы. Анализируемая строка завершается фигурной скобкой, открывающей тело данного метода.

Тело метода `AreaPerPerson()` состоит всего лишь из одного оператора.

```
Console.WriteLine(" " + Area / Occupants +
                  " приходится на одного человека");
```

Этот оператор осуществляет вывод величины площади на одного человека, которая получается путем деления общей площади здания (переменной `Area`) на количество жильцов (переменную `Occupants`). А поскольку у каждого объекта типа `Building` имеется своя копия переменных `Area` и `Occupants`, то при вызове метода `AreaPerPerson()` в вычислении используются копии этих переменных, принадлежащие вызывающему объекту.

Метод AreaPerPerson() завершается закрывающейся фигурной скобкой. Когда встречается эта скобка, управление передается обратно вызывающей части программы.

Далее проанализируем внимательно следующую строку кода из метода Main().

```
house.AreaPerPerson();
```

В этой строке вызывается метод AreaPerPerson() для объекта house. Это означает, что метод AreaPerPerson() вызывается относительно объекта, на который ссылается переменная house, и для этой цели служит оператор-точка. Когда метод AreaPerPerson() вызывается, ему передается управление программой. А по его завершении управление передается обратно вызывающей части программы, выполнение которой возобновляется со строки кода, следующей после вызова данного метода.

В данном случае в результате вызова house.AreaPerPerson() выводится площадь на одного человека в здании, определенном в объекте house. Аналогично, в результате вызова office.AreaPerPerson() выводится площадь на одного человека в здании, определенном в объекте office. Таким образом, при каждом вызове метода AreaPerPerson() выводится площадь на одного человека для указанного объекта.

В методе AreaPerPerson() особого внимания заслуживает следующее обстоятельство: обращение к переменным экземпляра Area и Occupants осуществляется непосредственно, т.е. без помощи оператора-точки. Если в методе используется переменная экземпляра, определенная в его классе, то делается это непосредственно, без указания явной ссылки на объект и без помощи оператора-точки. Понять это нетрудно, если хорошенько подумать. Ведь метод всегда вызывается относительно некоторого объекта его класса. Как только вызов произойдет, объект становится известным. Поэтому объект не нужно указывать в методе еще раз. В данном случае это означает, что переменные экземпляра Area и Occupants в методе AreaPerPerson() неявно ссылаются на копии этих же переменных в том объекте, который вызывает метод AreaPerPerson().

ПРИМЕЧАНИЕ

Попутно следует заметить, что значение переменной Occupants в методе AreaPerPerson() не должно быть равно нулю (это касается всех примеров, приведенных в данной главе). Если бы значение переменной Occupants оказалось равным нулю, то произошла бы ошибка из-за деления на нуль. В главе 13, где рассматриваются исключительные ситуации, будет показано, каким образом в C# отслеживаются и обрабатываются ошибки, которые могут возникнуть во время выполнения программы.

Возврат из метода

В целом, возврат из метода может произойти при двух условиях. Во-первых, когда встречается фигурная скобка, закрывающая тело метода, как показывает пример метода AreaPerPerson() из приведенной выше программы. И во-вторых, когда выполняется оператор return. Имеются две формы оператора return: одна — для методов типа void, т.е. тех методов, которые не возвращают значения, а другая — для методов, возвращающих конкретные значения. Первая форма рассматривается в этом разделе, а в следующем разделе будет пояснено, каким образом значения возвращаются из методов.

Для немедленного завершения метода типа void достаточно воспользоваться следующей формой оператора return.

```
return;
```

Когда выполняется этот оператор, управление возвращается вызывающей части программы, а оставшийся в методе код пропускается. В качестве примера рассмотрим следующий метод.

```
public void MyMeth() {
  int i;

  for(i=0; i<10; i++) {
    if(i == 5) return; // прервать на шаге 5
    Console.WriteLine();
  }
}
```

В данном примере выполняется лишь 5 полноценных шагов цикла `for`, поскольку при значении 5 переменной `i` происходит возврат из метода.

В методе допускается наличие нескольких операторов `return`, особенно если имеются два или более вариантов возврата из него. Например:

```
public void MyMeth() {
  // ...
  if(done) return;
  // ...
  if(error) return;
}
```

В данном примере возврат из метода происходит в двух случаях: если метод завершает свою работу или происходит ошибка. Но пользоваться таким приемом программирования следует очень аккуратно. Ведь из-за слишком большого числа точек возврата из метода может нарушиться структура кода.

Итак, напомним еще раз: возврат из метода типа `void` может произойти при двух условиях: по достижении закрывающей фигурной скобки или при выполнении оператора `return`.

Возврат значения

Методы с возвратом типа `void` нередко применяются в программировании, тем не менее, большинство методов возвращает конкретное значение. В действительности способность возвращать значение является одним из самых полезных свойств метода. Возврат значения уже демонстрировался в главе 3 на примере метода `Math.Sqrt()`, использовавшегося для получения квадратного корня.

Возвращаемые значения используются в программировании с самыми разными целями. В одних случаях, как в примере метода `Math.Sqrt()`, возвращаемое значение содержит результат некоторого вычисления, в других — оно может просто указывать на успешное или неудачное завершение метода, а в третьих — содержать код состояния. Но независимо от преследуемой цели использование возвращаемых значений является неотъемлемой частью программирования на C#.

Для возврата значения из метода в вызывающую часть программы служит следующая форма оператора `return`:

```
return значение;
```

где *значение* — это конкретное возвращаемое значение.

Используя возвращаемое значение, можно усовершенствовать рассматривавшийся ранее метод `AreaPerPerson()`. Вместо того чтобы выводить величину площади на одного человека, лучше возвратить ее из этого метода. Среди прочих преимуществ такого подхода следует особо отметить возможность использовать возвращаемое значение для выполнения других вычислений. Приведенный ниже пример представляет собой улучшенный вариант рассматривавшейся ранее программы с усовершенствованным методом `AreaPerPerson()`, возвращающим величину площади на одного человека вместо того, чтобы выводить ее.

```
// Возвратить значение из метода AreaPerPerson().

using System;

class Building {
  public int Floors;    // количество этажей
  public int Area;      // общая площадь здания
  public int Occupants; // количество жильцов

  // Возвратить величину площади на одного человека.
  public int AreaPerPerson() {
    return Area / Occupants;
  }
}
// Использовать значение, возвращаемое методом AreaPerPerson().
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();
    int areaPP; // площадь на одного человека


    // Присвоить значения полям в объекте house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;


    // Присвоить значения полям в объекте office.
    office.Occupants = 25;
    office.Area = 4200;
    office.Floors = 3;


    // Получить площадь на одного человека в жилом доме.
    areaPP = house.AreaPerPerson();

Console.WriteLine("Дом имеет:\n  " +
                  house.Floors + " этажа\n  " +
                  house.Occupants + " жильца\n  " +
                  house.Area +
                  " кв. футов общей площади, из них\n  " +
                  areaPP + " приходится на одного человека");
```

```
Console.WriteLine();

    // Получить площадь на одного человека в учреждении.
    areaPP = office.AreaPerPerson();

Console.WriteLine("Учреждение имеет:\n  " +
                  office.Floors + " этажа\n  " +
                  office.Occupants + " работников\n  " +
                  office.Area +
                  " кв. футов общей площади, из них\n  " +
                  areaPP + " приходится на одного человека");
  }
}
```

Эта программа дает такой же результат, как и прежде.

В данной программе обратите внимание на следующее: когда метод `AreaPerPerson()` вызывается, он указывается в правой части оператора присваивания. А в левой части этого оператора указывается переменная, которой передается значение, возвращаемое методом `AreaPerPerson()`. Следовательно, после выполнения оператора

```
areaPP = house.AreaPerPerson();
```

в переменной `areaPP` сохраняется величина площади на одного человека в жилом доме (объект `house`).

Обратите также внимание на то, что теперь метод `AreaPerPerson()` имеет возвращаемый тип `int`. Это означает, что он будет возвращать целое значение вызывающей части программы. Тип, возвращаемый методом, имеет очень большое значение, поскольку тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, указанным в методе. Так, если метод должен возвращать данные типа `double`, то в нем следует непременно указать возвращаемый тип `double`.

Несмотря на то что приведенная выше программа верна, она, тем не менее, написана не совсем эффективно. В частности, в ней можно вполне обойтись без переменной `areaPP`, указав вызов метода `AreaPerPerson()` непосредственно в операторе, содержащем вызов метода `WriteLine()`, как показано ниже.

```
Console.WriteLine("Дом имеет:\n  " +
                  house.Floors + " этажа\n  " +
                  house.Occupants + "  жильца\n  " +
                  house.Area +
                  " кв. футов общей площади, из них\n  " +
                  house.AreaPerPerson() +
                  " приходится на одного человека");
```

В данном случае при выполнении оператора, содержащего вызов метода `WriteLine()`, автоматически вызывается метод `house.AreaPerPerson()`, а возвращаемое им значение передается методу `WriteLine()`. Кроме того, вызов метода `AreaPerPerson()` можно использовать всякий раз, когда требуется получить величину площади на одного человека для конкретного объекта типа `Building`. Например, в приведенном ниже операторе сравниваются величины площади на одного человека для двух зданий.

```
if(b1.AreaPerPerson() > b2.AreaPerPerson())
  Console.WriteLine("В здании b1 больше места для каждого человека");
```

Использование параметров

При вызове метода ему можно передать одно или несколько значений. Значение, передаваемое методу, называется *аргументом*. А переменная, получающая аргумент, называется *формальным параметром*, или просто *параметром*. Параметры объявляются в скобках после имени метода. Синтаксис объявления параметров такой же, как и у переменных. А областью действия параметров является тело метода. За исключением особых случаев передачи аргументов методу, параметры действуют так же, как и любые другие переменные.

Ниже приведен пример программы, в котором демонстрируется применение параметра. В классе `ChkNum` используется метод `IsPrime()`, который возвращает значение `true`, если ему передается значение, являющееся простым числом. В противном случае он возвращает значение `false`. Следовательно, возвращаемым для метода `IsPrime()` является тип `bool`.

```
// Простой пример применения параметра.

using System;

class ChkNum {
  // Возвратить значение true, если значение
  // параметра x окажется простым числом.
  public bool IsPrime(int x) {
    if(x <= 1) return false;

    for(int i=2; i <= x/i; i++)
      if((x %i) == 0) return false;

    return true;
  }
}

class ParmDemo {
  static void Main() {
    ChkNum ob = new ChkNum();

    for(int i=2; i < 10; i++)
      if(ob.IsPrime(i)) Console.WriteLine(i + " простое число.");
      else Console.WriteLine(i + " непростое число.");
  }
}
```

Вот какой результат дает выполнение этой программы.

```
2 простое число.
3 простое число.
4 непростое число.
5 простое число.
6 непростое число.
7 простое число.
8 непростое число.
9 непростое число.
```

В данной программе метод `IsPrime()` вызывается восемь раз, и каждый раз ему передается другое значение. Проанализируем этот процесс более подробно. Прежде

всего обратите внимание на то, как вызывается метод `IsPrime()`. Его аргумент указывается в скобках. Когда метод `IsPrime()` вызывается в первый раз, ему передается значение 2. Следовательно, когда метод `IsPrime()` начинает выполняться, его параметр `x` принимает значение 2. При втором вызове этого метода его параметр `x` принимает значение 3, при третьем вызове — значение 4 и т.д. Таким образом, значение, передаваемое методу `IsPrime()` в качестве аргумента при его вызове, представляет собой значение, которое принимает его параметр `x`.

У метода может быть не только один, но и несколько параметров. Каждый его параметр объявляется, отделяясь от другого запятой. В качестве примера ниже приведен класс `ChkNum`, который расширен дополнительным методом `LeastComFactor()`, возвращающим наименьший общий множитель двух его аргументов. Иными словами, этот метод возвращает наименьшее число, на которое оба его аргумента делятся нацело.

```
// Добавить метод, принимающий два аргумента.

using System;

class ChkNum {
  // Возвратить значение true, если значение
  // параметра x окажется простым числом.
  public bool IsPrime(int x) {
    if(x <= 1) return false;

    for(int i=2; i <= x/i; i++)
      if((x %i) == 0) return false;

    return true;
  }

  // Возвратить наименьший общий множитель.
  public int LeastComFactor(int a, int b) {
    int max;

    if(IsPrime(a) || IsPrime(b)) return 1;

    max = a < b ? a : b;

    for(int i=2; i <= max/2; i++)
      if(((a%i) == 0) && ((b%i) == 0)) return i;
    return 1;
  }
}

class ParmDemo {
  static void Main() {
    ChkNum ob = new ChkNum();
    int a, b;

    for(int i=2; i < 10; i++)
      if(ob.IsPrime(i)) Console.WriteLine(i + " простое число.");
      else Console.WriteLine(i + " непростое число.");
```

```
    a = 7;
    b = 8;
    Console.WriteLine("Наименьший общий множитель чисел " +
                      a + " и " + b + " равен " +
                      ob.LeastComFactor(a, b));

    a = 100;
    b = 8;
    Console.WriteLine("Наименьший общий множитель чисел " +
                      a + " и " + b + " равен " +
                      ob.LeastComFactor(a, b));

    a = 100;
    b = 75;
    Console.WriteLine("Наименьший общий множитель чисел " +
                      a + " и " + b + " равен " +
                      ob.LeastComFactor(a, b));
  }
}
```

Обратите внимание на следующее: когда вызывается метод `LeastComFactor()`, его аргументы также разделяются запятыми. Ниже приведен результат выполнения данной программы.

```
2 простое число.
3 простое число.
4 непростое число.
5 простое число.
6 непростое число.
7 простое число.
8 непростое число.
9 непростое число.
Наименьший общий множитель чисел 7 и 8 равен 1
Наименьший общий множитель чисел 100 и 8 равен 2
Наименьший общий множитель чисел 100 и 75 равен 5
```

Если в методе используется несколько параметров, то для каждого из них указывается свой тип, отличающийся от других. Например, приведенный ниже код является вполне допустимым.

```
int MyMeth(int a, double b, float c) {
  // ...
```

Добавление параметризированного метода в класс `Building`

С помощью параметризированного метода можно дополнить класс `Building` новым средством, позволяющим вычислять максимальное количество жильцов в здании, исходя из определенной величины минимальной площади на одного человека. Этим новым средством является приведенный ниже метод `MaxOccupant()`.

```
// Возвратить максимальное количество человек, занимающих здание,
// исходя из заданной минимальной площади на одного человека.
public int MaxOccupant(int minArea) {
  return Area / minArea;
}
```

Когда вызывается метод MaxOccupant(), его параметр minArea принимает величину необходимой минимальной площади на одного человека. На эту величину делится общая площадь здания при выполнении данного метода, после чего он возвращает результат.

Ниже приведен весь класс Building, включая и метод MaxOccupant().

```
/*
  Добавить параметризированный метод, вычисляющий
  максимальное количество человек, которые могут
  занимать здание, исходя из заданной минимальной
  площади на одного человека.
*/

using System;

class Building {
  public int Floors;    // количество этажей
  public int Area;      // общая площадь здания
  public int Occupants; // количество жильцов

  // Возвратить площадь на одного человека.
    public int AreaPerPerson() {
    return Area / Occupants;
  }

  // Возвратить максимальное количество человек, занимающих здание,
  // исходя из заданной минимальной площади на одного человека.
  public int MaxOccupant(int minArea) {
    return Area / minArea;
  }
}

// Использовать метод MaxOccupant().
class BuildingDemo {
  static void Main() {
    Building house = new Building();
    Building office = new Building();

// Присвоить значения полям в объекте house.
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;

// Присвоить значения полям в объекте office.
    office.Occupants = 25;
    office.Area = 4200;
    office.Floors = 3;

    Console.WriteLine("Максимальное количество человек в доме, \n" +
                      "если на каждого должно приходиться " +
                      300 + " кв. футов: " +
                      house.MaxOccupant(300));

    Console.WriteLine("Максимальное количество человек " +
```

```
                    "в учреждении, \n" +
                    "если на каждого должно приходиться " +
                    300 + " кв. футов: " +
                    office.MaxOccupant(300));
  }
}
```

Выполнение этой программы дает следующий результат.

```
Максимальное количество человек в доме,
если на каждого должно приходиться 300 кв. футов: 8
Максимальное количество человек в учреждении,
если на каждого должно приходиться 300 кв. футов: 14
```

Исключение недоступного кода

При создании методов следует исключить ситуацию, при которой часть кода не может быть выполнена ни при каких обстоятельствах. Такой код называется *недоступным* и считается в C# неправильным. Если создать метод, содержащий недоступный код, компилятор выдаст предупреждающее сообщение соответствующего содержания. Рассмотрим следующий пример кода.

```
public void MyMeth() {
  char a, b;

  // ...

  if(a==b) {
    Console.WriteLine("равно");
    return;
  } else {
    Console.WriteLine("не равно");
    return;
  }
  Console.WriteLine("это недоступный код");
}
```

В данном примере возврат из метода `MyMeth()` всегда происходит до выполнения последнего оператора, содержащего вызов метода `WriteLine()`. Если попытаться скомпилировать этот код, то будет выдано предупреждающее сообщение. Вообще говоря, недоступный код считается ошибкой программирования, и поэтому предупреждения о таком коде следует воспринимать всерьез.

Конструкторы

В приведенных выше примерах программ переменные экземпляра каждого объекта типа `Building` приходилось инициализировать вручную, используя, в частности, следующую последовательность операторов.

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

Такой прием обычно не применяется в профессионально написанном коде C#. Кроме того, он чреват ошибками (вы можете просто забыть инициализировать одно из полей). Впрочем, существует лучший способ решить подобную задачу: воспользоваться конструктором.

Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно. Ниже приведена общая форма конструктора.

```
доступ имя_класса(список_параметров) {
  // тело конструктора
}
```

Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. Кроме того, *доступ* обычно представляет собой модификатор доступа типа `public`, поскольку конструкторы зачастую вызываются в классе. А *список_параметров* может быть как пустым, так и состоящим из одного или более указываемых параметров.

У всех классов имеются конструкторы, независимо от того, определите вы их или нет, поскольку в C# автоматически предоставляется конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа `bool` — значение `false`, а для ссылочных типов — пустое значение. Но как только вы определите свой собственный конструктор, то конструктор по умолчанию больше не используется.

Ниже приведен простой пример применения конструктора.

```
// Простой конструктор.

using System;

class MyClass {
  public int x;

  public MyClass() {
    x = 10;
  }
}

class ConsDemo {
  static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

В данном примере конструктор класса `MyClass` имеет следующий вид.

```
public MyClass() {
  x = 10;
}
```

Обратите внимание на то, что этот конструктор обозначается как `public`. Дело в том, что он должен вызываться из кода, определенного за пределами его класса. В этом конструкторе переменной экземпляра класса `MyClass` присваивается значение 10. Он вызывается в операторе `new` при создании объекта. Например, в следующей строке:

```
MyClass t1 = new MyClass();
```

конструктор `MyClass()` вызывается для объекта `t1`, присваивая переменной его экземпляра `t1.x` значение 10. То же самое происходит и для объекта `t2`. После конструирования переменная `t2.x` будет содержать то же самое значение 10. Таким образом, выполнение приведенного выше кода приведет к следующему результату.

```
10 10
```

Параметризированные конструкторы

В предыдущем примере использовался конструктор без параметров. В некоторых случаях этого оказывается достаточно, но зачастую конструктор должен принимать один или несколько параметров. В конструктор параметры вводятся таким же образом, как и в метод. Для этого достаточно объявить их в скобках после имени конструктора. Ниже приведен пример применения параметризированного конструктора `MyClass`.

```
// Параметризированный конструктор.

using System;

class MyClass {
  public int x;

  public MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  static void Main() {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

При выполнении этого кода получается следующий результат.

```
10 88
```

В данном варианте конструктора `MyClass()` определен параметр `i`, с помощью которого инициализируется переменная экземпляра `x`. Поэтому при выполнении следующей строки кода:

```
MyClass t1 = new MyClass(10);
```

параметру `i` передается значение, которое затем присваивается переменной `x`.

Добавление конструктора в класс Building

Класс Building можно усовершенствовать, добавив в него конструктор, автоматически инициализирующий поля Floors, Area и Occupants при создании объекта. Обратите особое внимание на то, как создаются объекты класса Building.

```
// Добавить конструктор в класс Building.

using System;

class Building {
  public int Floors;     // количество этажей
  public int Area;       // общая площадь здания
  public int Occupants; // количество жильцов

  // Параметризированный конструктор для класса Building.
  public Building(int f, int a, int o) {
    Floors = f;
    Area = a;
    Occupants = o;
  }

  // Возвратить площадь на одного человека.
    public int AreaPerPerson() {
    return Area / Occupants;
  }

  // Возвратить максимальное количество человек, занимающих здание,
  // исходя из заданной минимальной площади на одного человека.
  public int MaxOccupant(int minArea) {
    return Area / minArea;
  }
}

// Использовать параметризированный конструктор класса Building.
class BuildingDemo {
  static void Main() {
    Building house = new Building(2, 2500, 4);
    Building office = new Building(3, 4200, 25);

    Console.WriteLine("Максимальное количество человек в доме, \n" +
                      "если на каждого должно приходиться " +
                      300 + " кв. футов: " +
                      house.MaxOccupant(300));

    Console.WriteLine("Максимальное количество человек " +
                      "в учреждении, \n" +
                      "если на каждого должно приходиться " +
                      300 + " кв. футов: " +
                      office.MaxOccupant(300));
  }
}
```

Результат выполнения этой программы оказывается таким же, как и в предыдущей ее версии.

Оба объекта, house и office, были инициализированы конструктором Building() при их создании в соответствии с параметрами, указанными в этом конструкторе. Например, в строке

```
Building house = new Building(2, 2500, 4);
```

конструктору Building() передаются значения 2, 2500 и 4 при создании нового объекта. Следовательно, в копиях переменных экземпляра Floors, Area и Occupants объекта house будут храниться значения 2, 2500 и 4 соответственно.

Еще раз об операторе new

Теперь, когда вы ближе ознакомились с классами и их конструкторами, вернемся к оператору new, чтобы рассмотреть его более подробно. В отношении классов общая форма оператора new такова:

```
new имя_класса (список_аргументов)
```

где *имя_класса* обозначает имя класса, реализуемого в виде экземпляра его объекта. А *имя_класса* с последующими скобками обозначает конструктор этого класса. Если в классе не определен его собственный конструктор, то в операторе new будет использован конструктор, предоставляемый в C# по умолчанию. Следовательно, оператор new может быть использован для создания объекта, относящегося к классу любого типа.

Оперативная память не бесконечна, и поэтому вполне возможно, что оператору new не удастся распределить память для объекта из-за нехватки имеющейся оперативной памяти. В этом случае возникает исключительная ситуация во время выполнения (подробнее об обработке исключительных ситуаций речь пойдет в главе 13). В примерах программ, приведенных в этой книге, ситуация, связанная с исчерпанием оперативной памяти, не учитывается, но при написании реальных программ такую возможность, вероятно, придется принимать во внимание.

Применение оператора new вместе с типами значений

В связи с изложенным выше возникает резонный вопрос: почему оператор new нецелесообразно применять к переменным таких типов значений, как int или float? В C# переменная типа значения содержит свое собственное значение. Память для хранения этого значения выделяется автоматически во время прогона программы. Следовательно, распределять память явным образом с помощью оператора new нет никакой необходимости. С другой стороны, в переменной ссылочного типа хранится ссылка на объект, и поэтому память для хранения этого объекта должна распределяться динамически во время выполнения программы.

Благодаря тому что основные типы данных, например int или char, не преобразуются в ссылочные типы, существенно повышается производительность программы. Ведь при использовании ссылочного типа существует уровень косвенности, повышающий издержки на доступ к каждому объекту. Такой уровень косвенности исключается при использовании типа значения.

Но ради интереса следует все же отметить, что оператор new разрешается использовать вместе с типами значений, как показывает следующий пример.

```
int i = new int();
```

При этом для типа int вызывается конструктор, инициализирующий по умолчанию переменную i нулевым значением. В качестве примера рассмотрим такую программу.

```
// Использовать оператор new вместе с типом значения.

using System;

class newValue {
  static void Main() {
    int i = new int(); // инициализировать переменную i нулевым значением

    Console.WriteLine("Значение переменной i равно: " + i);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Значение переменной i равно: 0
```

Как показывает результат выполнения данной программы, переменная i инициализируется нулевым значением. Напомним, что если не применить оператор new, то переменная i окажется неинициализированной. Это может привести к ошибке при попытке воспользоваться ею в операторе, содержащем вызов метода WriteLine(), если предварительно не задать ее значение явным образом.

В общем, обращение к оператору new для любого типа значения приводит к вызову конструктора, используемого по умолчанию для данного типа. Но в этом случае память динамически не распределяется. Откровенно говоря, в программировании обычно не принято пользоваться оператором new вместе с типами значений.

"Сборка мусора" и применение деструкторов

Как было показано выше, при использовании оператора new свободная память для создаваемых объектов динамически распределяется из доступной буферной области оперативной памяти. Разумеется, оперативная память не бесконечна, и поэтому свободно доступная память рано или поздно исчерпывается. Это может привести к неудачному выполнению оператора new из-за нехватки свободной памяти для создания требуемого объекта. Именно по этой причине одной из главных функций любой схемы динамического распределения памяти является освобождение свободной памяти от неиспользуемых объектов, чтобы сделать ее доступной для последующего перераспределения. Во многих языках программирования освобождение распределенной ранее памяти осуществляется вручную. Например, в C++ для этой цели служит оператор delete. Но в C# применяется другой, более надежный подход: *"сборка мусора"*.

Система "сборки мусора" в C# освобождает память от лишних объектов автоматически, действуя незаметно и без всякого вмешательства со стороны программиста. "Сборка мусора" происходит следующим образом. Если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге освобождается и накапливается. Эта утилизированная память может быть затем распределена для других объектов.

"Сборка мусора" происходит лишь время от времени по ходу выполнения программы. Она не состоится только потому, что существует один или более объектов, которые больше не используются. Следовательно, нельзя заранее знать или предположить, когда именно произойдет "сборка мусора".

Деструкторы

В языке C# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется *деструктором* и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта. Например, деструктор может быть использован для гарантированного освобождения системного ресурса, задействованного освобождаемым объектом. Следует, однако, сразу же подчеркнуть, что деструкторы — весьма специфические средства, применяемые только в редких, особых случаях. И, как правило, они не нужны. Но здесь они рассматриваются вкратце ради полноты представления о возможностях языка C#.

Ниже приведена общая форма деструктора:

```
~имя_класса() {
  // код деструктора
}
```

где *имя_класса* означает имя конкретного класса. Следовательно, деструктор объявляется аналогично конструктору, за исключением того, что перед его именем указывается знак "тильда" (~). Обратите внимание на то, что у деструктора отсутствуют возвращаемый тип и передаваемые ему аргументы.

Для того чтобы добавить деструктор в класс, достаточно включить его в класс в качестве члена. Он вызывается всякий раз, когда предполагается утилизировать объект его класса. В деструкторе можно указать те действия, которые следует выполнить перед тем, как уничтожать объект.

Следует, однако, иметь в виду, что деструктор вызывается непосредственно перед "сборкой мусора". Он не вызывается, например, в тот момент, когда переменная, содержащая ссылку на объект, оказывается за пределами области действия этого объекта. (В этом отношении деструкторы в C# отличаются от деструкторов в C++, где они вызываются в тот момент, когда объект оказывается за пределами области своего действия.) Это означает, что заранее нельзя знать, когда именно следует вызывать деструктор. Кроме того, программа может завершиться до того, как произойдет "сборка мусора", а следовательно, деструктор может быть вообще не вызван.

Ниже приведен пример программы, демонстрирующий применение деструктора. В этой программе создается и уничтожается большое число объектов. В какой-то момент по ходу данного процесса активизируется "сборка мусора" и вызываются деструкторы для уничтожения ненужных объектов.

```
// Продемонстрировать применение деструктора.

using System;

class Destruct {
  public int x;

  public Destruct(int i) {
```

```
    x = i;
  }

  // Вызывается при утилизации объекта.
  ~Destruct() {
    Console.WriteLine("Уничтожить " + x);
  }

  // Создает объект и тут же уничтожает его.
  public void Generator(int i) {
  Destruct o = new Destruct(i);
  }
}

class DestructDemo {
  static void Main() {
    int count;

    Destruct ob = new Destruct(0);

    /* А теперь создать большое число объектов.
       В какой-то момент произойдет "сборка мусора".
       Примечание: для того чтобы активизировать
       "сборку мусора", возможно, придется увеличить
       число создаваемых объектов. */

    for(count=1; count < 100000; count++)
      ob.Generator(count);

    Console.WriteLine("Готово!");
  }
}
```

Эта программа работает следующим образом. Конструктор инициализирует переменную x известным значением. В данном примере переменная x служит в качестве идентификатора объекта. А деструктор выводит значение переменной x, когда объект утилизируется. Особый интерес вызывает метод `Generator()`, который создает и тут же уничтожает объект типа `Destruct`. Сначала в классе `DestructDemo` создается исходный объект `ob` типа `Destruct`, а затем осуществляется поочередное создание и уничтожение 100 тыс. объектов. В разные моменты этого процесса происходит "сборка мусора". Насколько часто она происходит — зависит от нескольких факторов, в том числе от первоначального объема свободной памяти, типа используемой операционной системы и т.д. Тем не менее в какой-то момент начинают появляться сообщения, формируемые деструктором. Если же они не появятся до окончания программы, т.е. до того момента, когда будет выдано сообщение "Готово!", попробуйте увеличить число создаваемых объектов, повысив предельное количество подсчитываемых шагов в цикле `for`.

И еще одно важное замечание: метод `WriteLine()` вызывается в деструкторе `~Destruct()` исключительно ради наглядности данного примера его использования. Как правило, деструктор должен воздействовать только на переменные экземпляра, определенные в его классе.

В силу того что порядок вызова деструкторов не определен точно, их не следует применять для выполнения действий, которые должны происходить в определенный

момент выполнения программы. В то же время имеется возможность запрашивать "сборку мусора", как будет показано в части II этой книги при рассмотрении библиотеки классов C#. Тем не менее инициализация "сборки мусора" вручную в большинстве случаев не рекомендуется, поскольку это может привести к снижению эффективности программы. Кроме того, у системы "сборки мусора" имеются свои особенности — даже если запросить "сборку мусора" явным образом, все равно нельзя заранее знать, когда именно будет утилизирован конкретный объект.

Ключевое слово this

Прежде чем завершать эту главу, необходимо представить ключевое слово this. Когда метод вызывается, ему автоматически передается ссылка на вызывающий объект, т.е. тот объект, для которого вызывается данный метод. Эта ссылка обозначается ключевым словом this. Следовательно, ключевое слово this обозначает именно тот объект, по ссылке на который действует вызываемый метод. Для того чтобы стало яснее назначение ключевого слова this, рассмотрим сначала пример программы, в которой создается класс Rect, инкапсулирующий ширину и высоту прямоугольника и включающий в себя метод Area(), возвращающий площадь прямоугольника.

```
using System;

class Rect {
  public int Width;
  public int Height;

  public Rect(int w, int h) {
    Width = w;
    Height = h;
  }

  public int Area() {
    return Width * Height;
  }
}

class UseRect {
  static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine("Площадь прямоугольника r1: " + r1.Area());

    Console.WriteLine("Площадь прямоугольника r2: " + r2.Area());
  }
}
```

Как вам должно уже быть известно, другие члены класса могут быть доступны непосредственно без дополнительного уточнения имени объекта или класса. Поэтому оператор

```
return Width * Height;
```

в методе Area() означает, что копии переменных Width и Height, связанные с вызывающим объектом, будут перемножены, а метод возвратит их произведение. Но тот же самый оператор можно написать следующим образом.

```
return this.Width * this.Height;
```

В этом операторе ключевое слово this обозначает объект, для которого вызван метод Area(). Следовательно, в выражении this.Width делается ссылка на копию переменной Width данного объекта, а в выражении this.Height — ссылка на копию переменной Height этого же объекта. Так, если бы метод Area() был вызван для объекта x, то ключевое слово this в приведенном выше операторе обозначало бы ссылку на объект x. Написание оператора без ключевого слова this представляет собой не более чем сокращенную форму записи.

Ключевое слово this можно также использовать в конструкторе. В этом случае оно обозначает объект, который конструируется. Например, следующие операторы в методе Rect()

```
Width = w;
Height = h;
```

можно было бы написать таким образом.

```
this.Width = w;
this.Height = h;
```

Разумеется, такой способ записи не дает в данном случае никаких преимуществ.

Ради примера ниже приведен весь класс Rect, написанный с использованием ссылки this.

```
using System;

class Rect {
  public int Width;
  public int Height;

  public Rect(int w, int h) {
    this.Width = w;
    this.Height = h;
  }

  public int Area() {
    return this.Width * this.Height;
  }
}

class UseRect {
  static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine("Площадь прямоугольника r1: " + r1.Area());

    Console.WriteLine("Площадь прямоугольника r2: " + r2.Area());
  }
}
```

В действительности ключевое слово `this` не используется приведенным выше способом в программировании на C#, поскольку это практически ничего не дает, да и стандартная форма записи намного проще и понятнее. Тем не менее ключевому слову `this` можно найти не одно полезное применение. Например, в синтаксисе C# допускается называть параметр или локальную переменную тем же именем, что и у переменной экземпляра. В этом случае имя локальной переменной *скрывает* переменную экземпляра. Для доступа к скрытой переменной экземпляра и служит ключевое слово `this`. Например, приведенный ниже код является правильным с точки зрения синтаксиса C# способом написания конструктора `Rect()`.

```
public Rect(int Width, int Height) {
  this.Width = Width;
  this.Height = Height;
}
```

В этом варианте написания конструктора `Rect()` имена параметров совпадают с именами переменных экземпляра, а следовательно, скрывают их. Но для "обнаружения" скрытых переменных служит ключевое слово `this`.

7 ГЛАВА

Массивы и строки

В этой главе речь вновь пойдет о типах данных в C#. В ней рассматриваются массивы и тип `string`, а также оператор цикла `foreach`.

Массивы

Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем. В C# массивы могут быть как одномерными, так и многомерными, хотя чаще всего применяются одномерные массивы. Массивы служат самым разным целям, поскольку они предоставляют удобные средства для объединения связанных вместе переменных. Например, в массиве можно хранить максимальные суточные температуры, зарегистрированные в течение месяца, перечень биржевых курсов или же названия книг по программированию из домашней библиотеки.

Главное преимущество массива — в организации данных таким образом, чтобы ими было проще манипулировать. Так, если имеется массив, содержащий дивиденды, выплачиваемые по определенной группе акций, то, организовав циклическое обращение к элементам этого массива, можно без особого труда рассчитать средний доход от этих акций. Кроме того, массивы позволяют организовать данные таким образом, чтобы легко отсортировать их.

Массивами в C# можно пользоваться практически так же, как и в других языках программирования. Тем не менее у них имеется одна особенность: они реализованы в виде объектов. Именно поэтому их рассмотрение было отложено до тех пор, пока в этой книге не были представлены

объекты. Реализация массивов в виде объектов дает ряд существенных преимуществ, и далеко не самым последним среди них является возможность утилизировать неиспользуемые массивы средствам "сборки мусора".

Одномерные массивы

Одномерный массив представляет собой список связанных переменных. Такие списки часто применяются в программировании. Например, в одномерном массиве можно хранить учетные номера активных пользователей сети или текущие средние уровни достижений бейсбольной команды.

Для того чтобы воспользоваться массивом в программе, требуется двухэтапная процедура, поскольку в C# массивы реализованы в виде объектов. Во-первых, необходимо объявить переменную, которая может обращаться к массиву. И во-вторых, нужно создать экземпляр массива, используя оператор `new`. Так, для объявления одномерного массива обычно применяется следующая общая форма:

```
тип[] имя_массива = new тип[размер];
```

где *тип* объявляет конкретный тип элемента массива. Тип элемента определяет тип данных каждого элемента, составляющего массив. Обратите внимание на квадратные скобки, которые сопровождают *тип*. Они указывают на то, что объявляется одномерный массив. А *размер* определяет число элементов массива.

ПРИМЕЧАНИЕ

Если у вас имеется некоторый опыт программирования на C или C++, обратите особое внимание на то, как объявляются массивы в C#. В частности, квадратные скобки следуют после названия типа, а не имени массива.

Обратимся к конкретному примеру. В приведенной ниже строке кода создается массив типа `int`, который составляется из десяти элементов и связывается с переменной ссылки на массив, именуемой `sample`.

```
int[] sample = new int[10];
```

В переменной `sample` хранится ссылка на область памяти, выделяемой для массива оператором `new`. Эта область памяти должна быть достаточно большой, чтобы в ней могли храниться десять элементов массива типа `int`.

Как и при создании экземпляра класса, приведенное выше объявление массива можно разделить на два отдельных оператора. Например:

```
int[] sample;
sample = new int[10];
```

В данном случае переменная `sample` не ссылается на какой-то определенный физический объект, когда она создается в первом операторе. И лишь после выполнения второго оператора эта переменная ссылается на массив.

Доступ к отдельному элементу массива осуществляется по индексу: *Индекс* обозначает положение элемента в массиве. В языке C# индекс первого элемента всех массивов оказывается нулевым. В частности, массив `sample` состоит из 10 элементов с индексами от 0 до 9. Для индексирования массива достаточно указать номер требуемого эле-

мента в квадратных скобках. Так, первый элемент массива sample обозначается как sample[0], а последний его элемент — как sample[9]. Ниже приведен пример программы, в которой заполняются все 10 элементов массива sample.

```
// Продемонстрировать одномерный массив.

using System;

class ArrayDemo {
  static void Main() {
    int[] sample = new int[10];
    int i;

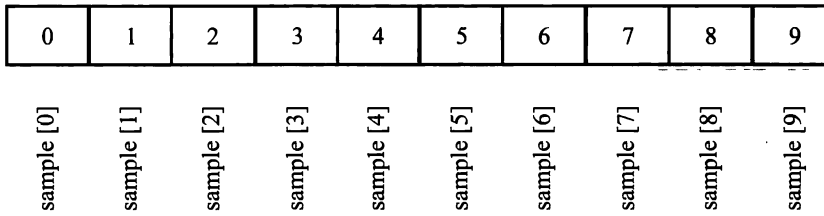
    for(i = 0; i < 10; i = i+1)
      sample[i] = i;

    for(i = 0; i < 10; i = i+1)
      Console.WriteLine("sample[" + i + "]: " + sample[i]);
  }
}
```

При выполнении этой программы получается следующий результат.

```
sample[0]: 0
sample[1]: 1
sample[2]: 2
sample[3]: 3
sample[4]: 4
sample[5]: 5
sample[6]: 6
sample[7]: 7
sample[8]: 8
sample[9]: 9
```

Схематически массив sample можно представить таким образом.



Массивы часто применяются в программировании потому, что они дают возможность легко обращаться с большим числом взаимосвязанных переменных. Например, в приведенной ниже программе выявляется среднее арифметическое ряда значений, хранящихся в массиве nums, который циклически опрашивается с помощью оператора цикла for.

```
// Вычислить среднее арифметическое ряда значений.

using System;

class Average {
  static void Main() {
```

```
    int[] nums = new int[10];
    int avg = 0;

    nums[0] = 99;
    nums[1] = 10;
    nums[2] = 100;
    nums[3] = 18;
    nums[4] = 78;
    nums[5] = 23;
    nums[6] = 63;
    nums[7] = 9;
    nums[8] = 87;
    nums[9] = 49;

    for(int i=0; i < 10; i++)
      avg = avg + nums[i];

    avg = avg / 10;

    Console.WriteLine("Среднее: " + avg);
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Среднее: 53
```

Инициализация массива

В приведенной выше программе первоначальные значения были заданы для элементов массива `nums` вручную в десяти отдельных операторах присваивания. Конечно, такая инициализация массива совершенно правильна, но то же самое можно сделать намного проще. Ведь массивы могут инициализироваться, когда они создаются. Ниже приведена общая форма инициализации одномерного массива:

```
тип[] имя_массива = {val1, val2, val3, ..., valN};
```

где *val1–valN* обозначают первоначальные значения, которые присваиваются по очереди, слева направо и по порядку индексирования. Для хранения инициализаторов массива в C# автоматически распределяется достаточный объем памяти. А необходимость пользоваться оператором `new` явным образом отпадает сама собой. В качестве примера ниже приведен улучшенный вариант программы, вычисляющей среднее арифметическое.

```
// Вычислить среднее арифметическое ряда значений.

using System;

class Average {
  static void Main() {
    int[] nums = { 99, 10, 100, 18, 78, 23,
                   63, 9, 87, 49 };
    int avg = 0;

    for(int i=0; i < 10; i++)
```

```
      avg = avg + nums[i];
    avg = avg / 10;

    Console.WriteLine("Среднее: " + avg);
  }
}
```

Любопытно, что при инициализации массива можно также воспользоваться оператором new, хотя особой надобности в этом нет. Например, приведенный ниже фрагмент кода считается верным, но избыточным для инициализации массива nums в упомянутой выше программе.

```
int[] nums = new int[] { 99, 10, 100, 18, 78, 23,
                         63, 9, 87, 49 };
```

Несмотря на свою избыточность, форма инициализации массива с оператором new оказывается полезной в том случае, если новый массив присваивается уже существующей переменной ссылки на массив. Например:

```
int[] nums;
nums = new int[] { 99, 10, 100, 18, 78, 23,
                   63, 9, 87, 49 };
```

В данном случае переменная nums объявляется в первом операторе и инициализируется во втором.

И последнее замечание: при инициализации массива его размер можно указывать явным образом, но этот размер должен совпадать с числом инициализаторов. В качестве примера ниже приведен еще один способ инициализации массива nums.

```
int[] nums = new int[10] { 99, 10, 100, 18, 78, 23,
                           63, 9, 87, 49 };
```

В этом объявлении размер массива nums задается равным 10 явно.

Соблюдение границ массива

Границы массива в C# строго соблюдаются. Если границы массива не достигаются или же превышаются, то возникает ошибка при выполнении. Для того чтобы убедиться в этом, попробуйте выполнить приведенную ниже программу, в которой намеренно превышаются границы массива.

```
// Продемонстрировать превышение границ массива.

using System;

class ArrayErr {
  static void Main() {
    int[] sample = new int[10];
    int i;

    // Воссоздать превышение границ массива.
    for(i = 0; i < 100; i = i+1)
      sample[i] = i;
  }
}
```

Как только значение переменной `i` достигнет 10, возникнет исключительная ситуация типа `IndexOutOfRangeException`, связанная с выходом за пределы индексирования массива, и программа преждевременно завершится. (Подробнее об исключительных ситуациях и их обработке речь пойдет в главе 13.)

Многомерные массивы

В программировании чаще всего применяются одномерные массивы, хотя и многомерные не так уж и редки. *Многомерным* называется такой массив, который отличается двумя или более измерениями, причем доступ к каждому элементу такого массива осуществляется с помощью определенной комбинации двух или более индексов.

Двумерные массивы

Простейшей формой многомерного массива является двумерный массив. Местоположение любого элемента в двумерном массиве обозначается двумя индексами. Такой массив можно представить в виде таблицы, на строки которой указывает один индекс, а на столбцы — другой.

В следующей строке кода объявляется двумерный массив `integer` размерами 10×20.

```
int[,] table = new int[10, 20];
```

Обратите особое внимание на объявление этого массива. Как видите, оба его размера разделяются запятой. В первой части этого объявления синтаксическое обозначение

```
[,]
```

означает, что создается переменная ссылки на двумерный массив. Если же память распределяется для массива с помощью оператора `new`, то используется следующее синтаксическое обозначение.

```
int[10, 20]
```

В данном объявлении создается массив размерами 10×20, но и в этом случае его размеры разделяются запятой.

Для доступа к элементу двумерного массива следует указать оба индекса, разделив их запятой. Например, в следующей строке кода элементу массива `table` с координатами местоположения (3,5) присваивается значение 10.

```
table[3, 5] = 10;
```

Ниже приведен более наглядный пример в виде небольшой программы, в которой двумерный массив сначала заполняется числами от 1 до 12, а затем выводится его содержимое.

```
// Продемонстрировать двумерный массив.

using System;

class TwoD {
  static void Main() {
    int t, i;
```

```
    int[,] table = new int[3, 4];

    for(t=0; t < 3; ++t) {
      for(i=0; i < 4; ++i) {
        table[t,i] = (t*4)+i+1;
        Console.Write(table[t,i] + " ");
      }
      Console.WriteLine();
    }
  }
}
```

В данном примере элемент массива `table[0,0]` будет иметь значение 1, элемент массива `table[0,1]` — значение 2, элемент массива `table[0,2]` — значение 3 и т.д. А значение элемента массива `table[2,3]` окажется равным 12. На рис. 7.1 показано схематически расположение элементов этого массива и их значений.

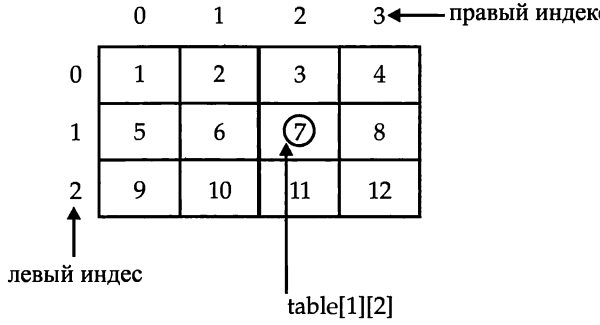


Рис. 7.1. Схематическое представление массива `table`, созданного в программе `TwoD`

СОВЕТ

Если вам приходилось раньше программировать на C, C++ или Java, то будьте особенно внимательны, объявляя или организуя доступ к многомерным массивам в C#. В этих языках программирования размеры массива и индексы указываются в отдельных квадратных скобках, тогда как в C# они разделяются запятой.

Массивы трех и более измерений

В C# допускаются массивы трех и более измерений. Ниже приведена общая форма объявления многомерного массива.

```
тип[,...,] имя_массива = new тип[размер1, размер2, ... размерN];
```

Например, в приведенном ниже объявлении создается трехмерный целочисленный массив размерами 4×10×3.

```
int[,,] multidim = new int[4, 10, 3];
```

А в следующем операторе элементу массива `multidim` с координатами местоположения (2,4,1) присваивается значение 100.

```
multidim[2, 4, 1] = 100;
```

Ниже приведен пример программы, в которой сначала организуется трехмерный массив, содержащий матрицу значений 3×3×3, а затем значения элементов этого массива суммируются по одной из диагоналей матрицы.

```
// Суммировать значения по одной из диагоналей матрицы 3x3x3.

using System;

class ThreeDMatrix {
  static void Main() {
    int[,,] m = new int[3, 3, 3];
    int sum = 0;
    int n = 1;

    for(int x=0; x < 3; x++)
      for(int y=0; y < 3; y++)
        for(int z=0; z < 3; z++)
          m[x, y, z] = n++;

    sum = m[0, 0, 0] + m[1, 1, 1] + m[2, 2, 2];

    Console.WriteLine("Сумма значений по первой диагонали: " + sum);
  }
}
```

Вот какой результат дает выполнение этой программы.

```
Сумма значений по первой диагонали: 42
```

Инициализация многомерных массивов

Для инициализации многомерного массива достаточно заключить в фигурные скобки список инициализаторов каждого его размера. Ниже в качестве примера приведена общая форма инициализации двумерного массива:

```
тип[,] имя_массива = {
  {val, val, val, ..., val},
  {val, val, val, ..., val},
.
.
.
  {val, val, val, ..., val}
};
```

где *val* обозначает инициализирующее значение, а каждый внутренний блок — отдельный ряд. Первое значение в каждом ряду сохраняется на первой позиции в массиве, второе значение — на второй позиции и т.д. Обратите внимание на то, что блоки инициализаторов разделяются запятыми, а после завершающей эти блоки закрывающей фигурной скобки ставится точка с запятой.

В качестве примера ниже приведена программа, в которой двумерный массив `sqrs` инициализируется числами от 1 до 10 и квадратами этих чисел.

```
// Инициализировать двумерный массив.

using System;
```

```
class Squares {
  static void Main() {
    int[,] sqrs = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };

    int i, j;

    for(i=0; i < 10; i++) {
      for(j=0; j < 2; j++)
        Console.Write(sqrs[i,j] + " ");
      Console.WriteLine();
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Ступенчатые массивы

В приведенных выше примерах применения двумерного массива, по существу, создавался так называемый *прямоугольный массив*. Двумерный массив можно представить в виде таблицы, в которой длина каждой строки остается неизменной по всему массиву. Но в C# можно также создавать специальный тип двумерного массива, называемый *ступенчатым массивом*. Ступенчатый массив представляет собой *массив массивов*, в котором длина каждого массива может быть разной. Следовательно, ступенчатый массив может быть использован для составления таблицы из строк разной длины.

Ступенчатые массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

```
тип[][] имя_массива = new тип[размер][];
```

где *размер* обозначает число строк в массиве. Память для самих строк распределяется индивидуально, и поэтому длина строк может быть разной. Например, в приведенном ниже фрагменте кода объявляется ступенчатый массив `jagged`. Память сначала распределяется для его первого измерения автоматически, а затем для второго измерения вручную.

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```

После выполнения этого фрагмента кода массив `jagged` выглядит так, как показано ниже.



Теперь нетрудно понять, почему такие массивы называются ступенчатыми! После создания ступенчатого массива доступ к его элементам осуществляется по индексу, указываемому в отдельных квадратных скобках. Например, в следующей строке кода элементу массива `jagged`, находящемуся на позиции с координатами (2,1), присваивается значение 10.

```
jagged[2][1] = 10;
```

Обратите внимание на синтаксические отличия в доступе к элементу ступенчатого и прямоугольного массива.

В приведенном ниже примере программы демонстрируется создание двумерного ступенчатого массива.

```
// Продемонстрировать применение ступенчатых массивов.

using System;

class Jagged {
  static void Main() {
    int[][] jagged = new int[3][];
    jagged[0] = new int[4];
    jagged[1] = new int[3];
    jagged[2] = new int[5];

    int i;

    // Сохранить значения в первом массиве.
    for(i=0; i < 4; i++)
      jagged[0][i] = i;

    // Сохранить значения во втором массиве.
    for(i=0; i < 3; i++)
      jagged[1][i] = i;
```

```
    // Сохранить значения в третьем массиве.
    for(i=0; i < 5; i++)
      jagged[2][i] = i;

    // Вывести значения из первого массива.
    for(i=0; i < 4; i++)
      Console.Write(jagged[0][i] + " ");

    Console.WriteLine();

    // Вывести значения из второго массива.
    for(i=0; i < 3; i++)
      Console.Write(jagged[1][i] + " ");

    Console.WriteLine();

    // Вывести значения из третьего массива.
    for(i=0; i < 5; i++)
      Console.Write(jagged[2][i] + " ");

    Console.WriteLine();
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
0 1 2 3
0 1 2
0 1 2 3 4
```

Ступенчатые массивы находят полезное применение не во всех, а лишь в некоторых случаях. Так, если требуется очень длинный двумерный массив, который заполняется не полностью, т.е. такой массив, в котором используются не все, а лишь отдельные его элементы, то для этой цели идеально подходит ступенчатый массив.

И последнее замечание: ступенчатые массивы представляют собой массивы массивов, и поэтому они не обязательно должны состоять из одномерных массивов. Например, в приведенной ниже строке кода создается массив двумерных массивов.

```
int[][,] jagged = new int[3][,];
```

В следующей строке кода элементу массива `jagged[0]` присваивается ссылка на массив размерами 4×2.

```
jagged[0] = new int[4, 2];
```

А в приведенной ниже строке кода элементу массива `jagged[0][1,0]` присваивается значение переменной `i`.

```
jagged[0][1,0] = i;
```

Присваивание ссылок на массивы

Присваивание значения одной переменной ссылки на массив другой переменной, по существу, означает, что обе переменные ссылаются на один и тот же массив,

и в этом отношении массивы ничем не отличаются от любых других объектов. Такое присваивание не приводит ни к созданию копии массива, ни к копированию содержимого одного массива в другой. В качестве примера рассмотрим следующую программу.

```
// Присваивание ссылок на массивы.

using System;

class AssignARef {
  static void Main() {
    int i;

    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < 10; i++) nums1[i] = i;

    for(i=0; i < 10; i++) nums2[i] = -i;
      Console.Write("Содержимое массива nums1: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();

    Console.Write("Содержимое массива nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    nums2 = nums1; // теперь nums2 ссылается на nums1

    Console.Write("Содержимое массива nums2\n" + "после присваивания: ");
    for(i=0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    // Далее оперировать массивом nums1 посредством
    // переменной ссылки на массив nums2.
    nums2[3] = 99;

    Console.Write("Содержимое массива nums1 после изменения\n" +
                  "посредством переменной nums2: ");
    for(i=0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Содержимое массива nums1: 0 1 2 3 4 5 6 7 8 9
Содержимое массива nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Содержимое массива nums2
после присваивания: 0 1 2 3 4 5 6 7 8 9
Содержимое массива nums1 после изменения
посредством переменной nums2: 0 1 2 99 4 5 6 7 8 9
```

Как видите, после присваивания переменной nums2 значения переменной nums1 обе переменные ссылки на массив ссылаются на один и тот же объект.

Применение свойства Length

Реализация в C# массивов в виде объектов дает целый ряд преимуществ. Одно из них заключается в том, что с каждым массивом связано свойство Length, содержащее число элементов, из которых может состоять массив. Следовательно, у каждого массива имеется специальное свойство, позволяющее определить его длину. Ниже приведен пример программы, в которой демонстрируется это свойство.

```
// Использовать свойство Length массива.

using System;

class LengthDemo {
  static void Main() {
    int[] nums = new int[10];

    Console.WriteLine("Длина массива nums равна " + nums.Length);

    // Использовать свойство Length для инициализации массива nums.
    for(int i=0; i < nums.Length; i++)
      nums[i] = i * i;

    // А теперь воспользоваться свойством Length
    // для вывода содержимого массива nums.
    Console.Write("Содержимое массива nums: ");
    for(int i=0; i < nums.Length; i++)
      Console.Write(nums[i] + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Длина массива nums равна 10
Содержимое массива nums: 0 1 4 9 16 25 36 49 64 81
```

Обратите внимание на то, как в классе LengthDemo свойство nums.Length используется в циклах for для управления числом повторяющихся шагов цикла. У каждого массива имеется своя длина, поэтому вместо отслеживания размера массива вручную можно использовать информацию о его длине. Следует, однако, иметь в виду, что значение свойства Length никак не отражает число элементов, которые в нем используются на самом деле. Свойство Length содержит лишь число элементов, из которых может состоять массив.

Когда запрашивается длина многомерного массива, то возвращается общее число элементов, из которых может состоять массив, как в приведенном ниже примере кода.

```
// Использовать свойство Length трехмерного массива.

using System;
```

```
class LengthDemo3D {
  static void Main() {
    int[,,] nums = new int[10, 5, 6];

    Console.WriteLine("Длина массива nums равна " + nums.Length);
  }
}
```

При выполнении этого кода получается следующий результат.

```
Длина массива nums равна 300
```

Как подтверждает приведенный выше результат, свойство `Length` содержит число элементов, из которых может состоять массив (в данном случае — 300 (10×5×6) элементов). Тем не менее свойство `Length` нельзя использовать для определения длины массива в отдельном его измерении.

Благодаря наличию у массивов свойства `Length` операции с массивами во многих алгоритмах становятся более простыми, а значит, и более надежными. В качестве примера свойство `Length` используется в приведенной ниже программе с целью поменять местами содержимое элементов массива, скопировав их в обратном порядке в другой массив.

```
// Поменять местами содержимое элементов массива.

using System;

class RevCopy {
  static void Main() {
    int i,j;
    int[] nums1 = new int[10];
    int[] nums2 = new int[10];

    for(i=0; i < nums1.Length; i++) nums1[i] = i;

    Console.Write("Исходное содержимое массива: ");
    for(i=0; i < nums2.Length; i++)
      Console.Write(nums1[i] + " ");

    Console.WriteLine();

    // Скопировать элементы массива nums1 в массив nums2 в обратном порядке.
    if(nums2.Length >= nums1.Length) // проверить, достаточно ли
                                     // длины массива nums2
      for(i=0, j=nums1.Length-1; i < nums1.Length; i++, j--)
        nums2[j] = nums1[i];

    Console.Write("Содержимое массива в обратном порядке: ");
    for(i=0; i < nums2.Length; i++)
      Console.Write(nums2[i] + " ");

    Console.WriteLine();
  }
}
```

Выполнение этой программы дает следующий результат.

```
Исходное содержимое массива: 0 1 2 3 4 5 6 7 8 9
Содержимое массива в обратном порядке: 9 8 7 6 5 4 3 2 1 0
```

В данном примере свойство `Length` помогает выполнить две важные функции. Во-первых, оно позволяет убедиться в том, что длины целевого массива достаточно для хранения содержимого исходного массива. И во-вторых, оно предоставляет условие для завершения цикла `for`, в котором выполняется копирование исходного массива в обратном порядке. Конечно, в этом простом примере размеры массивов нетрудно выяснить и без свойства `Length`, но аналогичный подход может быть применен в целом ряде других, более сложных ситуаций.

Применение свойства `Length` при обращении со ступенчатыми массивами

Особый случай представляет применение свойства `Length` при обращении со ступенчатыми массивами. В этом случае с помощью данного свойства можно получить длину каждого массива, составляющего ступенчатый массив. В качестве примера рассмотрим следующую программу, в которой имитируется работа центрального процессора (ЦП) в сети, состоящей из четырех узлов.

```
// Продемонстрировать применение свойства Length
// при обращении со ступенчатыми массивами.

using System;

class Jagged {
  static void Main() {
    int[][] network_nodes = new int[4][];
    network_nodes[0] = new int[3];
    network_nodes[1] = new int[7];
    network_nodes[2] = new int[2];
    network_nodes[3] = new int[5];

    int i, j;

    // Сфабриковать данные об использовании ЦП.
    for(i=0; i < network_nodes.Length; i++)
      for(j=0; j < network_nodes[i].Length; j++)
        network_nodes[i][j] = i * j + 70;

    Console.WriteLine("Общее количество узлов сети: " +
                      network_nodes.Length + "\n");

    for(i=0; i < network_nodes.Length; i++) {
      for(j=0; j < network_nodes[i].Length; j++) {
        Console.Write("Использование в узле сети " + i +
                      " ЦП " + j + ": ");
        Console.Write(network_nodes[i][j] + "% ");
        Console.WriteLine();
      }
      Console.WriteLine();
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
Общее количество узлов сети: 4

Использование в узле 0 ЦП 0: 70%
Использование в узле 0 ЦП 1: 70%
Использование в узле 0 ЦП 2: 70%

Использование в узле 1 ЦП 0: 70%
Использование в узле 1 ЦП 1: 71%
Использование в узле 1 ЦП 2: 72%
Использование в узле 1 ЦП 3: 73%
Использование в узле 1 ЦП 4: 74%
Использование в узле 1 ЦП 5: 75%
Использование в узле 1 ЦП 6: 76%

Использование в узле 2 ЦП 0: 70%
Использование в узле 2 ЦП 1: 72%

Использование в узле 3 ЦП 0: 70%
Использование в узле 3 ЦП 1: 73%
Использование в узле 3 ЦП 2: 76%
Использование в узле 3 ЦП 3: 79%
Использование в узле 3 ЦП 4: 82%
```

Обратите особое внимание на то, как свойство `Length` используется в ступенчатом массиве `network_nodes`. Напомним, что двумерный ступенчатый массив представляет собой массив массивов. Следовательно, когда используется выражение

```
network_nodes.Length
```

то в нем определяется число массивов, хранящихся в массиве `network_nodes` (в данном случае — четыре массива). А для получения длины любого отдельного массива, составляющего ступенчатый массив, служит следующее выражение.

```
network_nodes[0].Length
```

В данном случае это длина первого массива.

Неявно типизированные массивы

Как пояснялось в главе 3, в версии C# 3.0 появилась возможность объявлять неявно типизированные переменные с помощью ключевого слова `var`. Это переменные, тип которых определяется компилятором, исходя из типа инициализирующего выражения. Следовательно, все неявно типизированные переменные должны быть непременно инициализированы. Используя тот же самый механизм, можно создать и неявно типизированный массив. Как правило, неявно типизированные массивы предназначены для применения в определенного рода вызовах, включающих в себя элементы языка LINQ, о котором речь пойдет в главе 19. А в большинстве остальных случаев используется "обычное" объявление массивов. Неявно типизированные массивы рассматриваются здесь лишь ради полноты представления о возможностях языка C#.

Неявно типизированный массив объявляется с помощью ключевого слова `var`, но *без* последующих квадратных скобок `[]`. Кроме того, неявно типизированный мас-

сив должен быть непременно инициализирован, поскольку по типу инициализаторов определяется тип элементов данного массива. Все инициализаторы должны быть одного и того же согласованного типа. Ниже приведен пример объявления неявно типизированного массива.

```
var vals = new[] { 1, 2, 3, 4, 5 };
```

В данном примере создается массив типа `int`, состоящий из пяти элементов. Ссылка на этот массив присваивается переменной `vals`. Следовательно, тип этой переменной соответствует типу `int` массива, состоящего из пяти элементов. Обратите внимание на то, что в левой части приведенного выше выражения отсутствуют квадратные скобки `[]`. А в правой части этого выражения, где происходит инициализация массива, квадратные скобки присутствуют. В данном контексте они обязательны.

Рассмотрим еще один пример, в котором создается двумерный массив типа `double`.

```
var vals = new[,] { {1.1, 2.2}, {3.3, 4.4},{ 5.5, 6.6} };
```

В данном случае получается массив `vals` размерами 2×3.

Объявлять можно также неявно типизированные ступенчатые массивы. В качестве примера рассмотрим следующую программу.

```
// Продемонстрировать неявно типизированный ступенчатый массив.

using System;

class Jagged {
  static void Main() {

    var jagged = new[] {
      new[] { 1, 2, 3, 4 },
      new[] { 9, 8, 7 },
      new[] { 11, 12, 13, 14, 15 }
    };

    for(int j = 0; j < jagged.Length; j++) {
      for(int i=0; i < jagged[j].Length; i++)
        Console.Write(jagged[j][i] + " ");

      Console.WriteLine();
    }
  }
}
```

Выполнение этой программы дает следующий результат.

```
1 2 3 4
9 8 7
11 12 13 14 15
```

Обратите особое внимание на объявление массива `jagged`.

```
var jagged = new[] {
  new[] { 1, 2, 3, 4 },
  new[] { 9, 8, 7 },
  new[] { 11, 12, 13, 14, 15 }
};
```

Как видите, оператор new[] используется в этом объявлении двояким образом. Во-первых, этот оператор создает массив массивов. И во-вторых, он создает каждый массив в отдельности, исходя из количества инициализаторов и их типа. Как и следовало ожидать, все инициализаторы отдельных массивов должны быть одного и того же типа. Таким образом, к объявлению любого неявно типизированного ступенчатого массива применяется тот же самый общий подход, что и к объявлению обычных ступенчатых массивов.

Как упоминалось выше, неявно типизированные массивы чаще всего применяются в LINQ-ориентированных запросах. А в остальных случаях следует использовать явно типизированные массивы.

Оператор цикла foreach

Как упоминалось в главе 5, в языке C# определен оператор цикла foreach, но его рассмотрение было отложено до более подходящего момента. Теперь этот момент настал.

Оператор foreach служит для циклического обращения к элементам *коллекции*, представляющей собой группу объектов. В C# определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла foreach.

```
foreach (тип имя_переменной_цикла in коллекция) оператор;
```

Здесь *тип имя_переменной_цикла* обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла foreach. А *коллекция* обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, *тип* переменной цикла должен соответствовать типу элемента массива. Кроме того, *тип* может обозначаться ключевым словом var. В этом случае компилятор определяет тип переменной цикла, исходя из типа элемента массива. Это может оказаться полезным для работы с определенного рода запросами, как будет показано далее в данной книге. Но, как правило, тип указывается явным образом.

Оператор цикла foreach действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными. Следовательно, оператор foreach циклически опрашивает массив по отдельным его элементам от начала и до конца.

Следует, однако, иметь в виду, что переменная цикла в операторе foreach служит только для чтения. Это означает, что, присваивая этой переменной новое значение, нельзя изменить содержимое массива.

Ниже приведен простой пример применения оператора цикла foreach. В этом примере сначала создается целочисленный массив и задается ряд его первоначальных значений, а затем эти значения выводятся, а по ходу дела вычисляется их сумма.

```
// Использовать оператор цикла foreach.

using System;

class ForeachDemo {
  static void Main() {
```

```
    int sum = 0;
    int[] nums = new int[10];

    // Задать первоначальные значения элементов массива nums.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // Использовать цикл foreach для вывода значений
    // элементов массива и подсчета их суммы.
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента равно: " + x);
      sum += x;
    }

    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

Выполнение приведенного выше кода дает следующий результат.

```
Значение элемента равно: 0
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Значение элемента равно: 5
Значение элемента равно: 6
Значение элемента равно: 7
Значение элемента равно: 8
Значение элемента равно: 9
Сумма равна: 45
```

Как видите, оператор `foreach` циклически опрашивает массив по порядку индексирования от самого первого до самого последнего его элемента.

Несмотря на то что цикл `foreach` повторяется до тех пор, пока не будут опрошены все элементы массива, его можно завершить преждевременно, воспользовавшись оператором `break`. Ниже приведен пример программы, в которой суммируются только пять первых элементов массива `nums`.

```
// Использовать оператор break для преждевременного завершения цикла
foreach.

using System;

class ForeachDemo {
  static void Main() {
    int sum = 0;
    int[] nums = new int[10];

    // Задать первоначальные значения элементов массива nums.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // Использовать цикл foreach для вывода значений
    // элементов массива и подсчета их суммы.
```

```
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента равно: " + x);
      sum += x;
      if(x == 4) break; // прервать цикл, как только индекс массива достигнет 4
    }
    Console.WriteLine("Сумма первых 5 элементов: " + sum);
  }
}
```

Вот какой результат дает выполнение этой программы.

```
Значение элемента равно: 0
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
Сумма первых 5 элементов: 10
```

Совершенно очевидно, что цикл `foreach` завершается после выбора и вывода значения пятого элемента массива.

Оператор цикла `foreach` можно также использовать для циклического обращения к элементам многомерного массива. В этом случае элементы многомерного массива возвращаются по порядку следования строк от первой до последней, как демонстрирует приведенный ниже пример программы.

```
// Использовать оператор цикла foreach для обращения к двумерному массиву.

using System;

class ForeachDemo2 {
  static void Main() {
    int sum = 0;
    int[,] nums = new int[3,5];

    // Задать первоначальные значения элементов массива nums.
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i,j] = (i+1)*(j+1);

    // Использовать цикл foreach для вывода значений
    // элементов массива и подсчета их суммы.
    foreach(int x in nums) {
      Console.WriteLine("Значение элемента равно: " + x);
      sum += x;
    }
    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Значение элемента равно: 1
Значение элемента равно: 2
Значение элемента равно: 3
Значение элемента равно: 4
```

```
Значение элемента равно: 5
Значение элемента равно: 2
Значение элемента равно: 4
Значение элемента равно: 6
Значение элемента равно: 8
Значение элемента равно: 10
Значение элемента равно: 3
Значение элемента равно: 6
Значение элемента равно: 9
Значение элемента равно: 12
Значение элемента равно: 15
Сумма равна: 90
```

Оператор `foreach` допускает циклическое обращение к массиву только в определенном порядке: от начала и до конца массива, поэтому его применение кажется, на первый взгляд, ограниченным. Но на самом деле это не так. В большом числе алгоритмов, самым распространенным из которых является алгоритм поиска, требуется именно такой механизм. В качестве примера ниже приведена программа, в которой цикл `foreach` используется для поиска в массиве определенного значения. Как только это значение будет найдено, цикл прервется.

```
// Поиск в массиве с помощью оператора цикла foreach.

using System;

class Search {
  static void Main() {
    int[] nums = new int[10];
    int val;
    bool found = false;

    // Задать первоначальные значения элементов массива nums.
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    val = 5;

    // Использовать цикл foreach для поиска заданного
    // значения в массиве nums.
    foreach(int x in nums) {
      if(x == val) {
      found = true;
      break;
      }
    }

    if(found)
      Console.WriteLine("Значение найдено!");
  }
}
```

При выполнении этой программы получается следующий результат.

```
Значение найдено!
```

Оператор цикла `foreach` отлично подходит для такого применения, поскольку при поиске в массиве приходится анализировать каждый его элемент. К другим примерам применения оператора цикла `foreach` относится вычисление среднего, поиск минимального или максимального значения среди ряда заданных значений, обнаружение дубликатов и т.д. Как будет показано далее в этой книге, оператор цикла `foreach` оказывается особенно полезным для работы с разными типами коллекций.

Строки

С точки зрения регулярного программирования строковый тип данных `string` относится к числу самых важных в C#. Этот тип определяет и поддерживает символьные строки. В целом ряде других языков программирования строка представляет собой массив символов. А в C# строки являются объектами. Следовательно, тип `string` относится к числу ссылочных. И хотя `string` является встроенным в C# типом данных, его рассмотрение пришлось отложить до тех пор, пока не были представлены классы и объекты.

На самом деле класс типа `string` уже не раз применялся в примерах программ, начиная с главы 2, но это обстоятельство выясняется только теперь, когда очередь дошла до строк. При создании строкового литерала в действительности формируется строковый объект. Например, в следующей строке кода:

```
Console.WriteLine("В C# строки являются объектами.");
```

текстовая строка `"В C# строки являются объектами."` автоматически преобразуется в строковый объект средствами C#. Следовательно, применение класса типа `string` происходило в предыдущих примерах программ неявным образом. А в этом разделе будет показано, как обращаться со строками явным образом.

Построение строк

Самый простой способ построить символьную строку — воспользоваться строковым литералом. Например, в следующей строке кода переменной ссылки на строку `str` присваивается ссылка на строковый литерал.

```
string str = "Строки в C# весьма эффективны.";
```

В данном случае переменная `str` инициализируется последовательностью символов `"Строки в C# весьма эффективны."`.

Объект типа `string` можно также создать из массива типа `char`. Например:

```
char[] charray = {'t', 'e', 's', 't'};
string str = new string(charray);
```

Как только объект типа `string` будет создан, его можно использовать везде, где только требуется строка текста, заключенного в кавычки. Как показано в приведенном ниже примере программы, объект типа `string` может служить в качестве аргумента при вызове метода `WriteLine()`.

```
// Создать и вывести символьную строку.

using System;
```

```
class StringDemo {
  static void Main() {

    char[] charray = {'Э', 'т', 'о', ' ', 'с', 'т', 'р', 'о', 'к', 'а',
'.'};
    string str1 = new string(charray);
    string str2 = "Еще одна строка.";

    Console.WriteLine(str1);
    Console.WriteLine(str2);
  }
}
```

Результат выполнения этой программы приведен ниже.

```
Это строка.
Еще одна строка.
```

Обращение со строками

Класс типа `string` содержит ряд методов для обращения со строками. Некоторые из этих методов перечислены в табл. 7.1. Обратите внимание на то, что некоторые методы принимают параметр типа `StringComparison`. Это перечислимый тип, определяющий различные значения, которые определяют порядок сравнения символьных строк. (О перечислениях речь пойдет в главе 12, но для применения типа `StringComparison` к символьным строкам знать о перечислениях необязательно.) Нетрудно догадаться, что символьные строки можно сравнивать разными способами. Например, их можно сравнивать на основании двоичных значений символов, из которых они состоят. Такое сравнение называется *порядковым*. Строки можно также сравнивать с учетом различных особенностей культурной среды, например, в лексикографическом порядке. Это так называемое сравненение *с учетом культурной среды*. (Учитывать культурную среду особенно важно в локализуемых приложениях.) Кроме того, строки можно сравнивать *с учетом* или *без учета* регистра. Несмотря на то что существуют перегружаемые варианты методов `Compare()`, `Equals()`, `IndexOf()` и `LastIndexOf()`, обеспечивающие используемый по умолчанию подход к сравнению символьных строк, в настоящее время считается более приемлемым явно указывать способ требуемого сравнения, чтобы избежать неоднозначности, а также упростить локализацию приложений. Именно поэтому здесь рассматривают разные способы сравнения символьных строк.

Как правило и за рядом исключений, для сравнения символьных строк с учетом культурной среды (т.е. языковых и региональных стандартов) применяется способ `StringComparison.CurrentCulture`. Если же требуется сравнить строки только на основании значений их символов, то лучше воспользоваться способом `StringComparison.Ordinal`, а для сравнения строк без учета регистра — одним из двух способов: `StringComparison.CurrentCultureIgnoreCase` или `StringComparison.OrdinalIgnoreCase`. Кроме того, можно указать сравнение строк без учета культурной среды (подробнее об этом — в главе 22).

Обратите внимание на то, что метод `Compare()` объявляется в табл. 7.1 как `static`. Подробнее о модификаторе `static` речь пойдет в главе 8, а до тех пор вкратце поясним, что он обозначает следующее: метод `Compare()` вызывается по имени своего

класса, а не по его экземпляру. Следовательно, для вызова метода `Compare()` служит следующая общая форма:

```
результат = string.Compare(str1, str2, способ);
```

где *способ* обозначает конкретный подход к сравнению символьных строк.

ПРИМЕЧАНИЕ

Дополнительные сведения о способах сравнения и поиска символьных строк, включая и особое значение выбора подходящего способа, приведены в главе 22, где подробно рассматривается обработка строк.

Обратите также внимание на методы `ToUpper()` и `ToLower()`, преобразующие содержимое строки в символы верхнего и нижнего регистра соответственно. Их формы, представленные в табл. 7.1, содержат параметр `CultureInfo`, относящийся к классу, в котором описываются атрибуты культурной среды, применяемые для сравнения. В примерах, приведенных в этой книге, используются текущие настройки культурной среды (т.е. текущие языковые и региональные стандарты). Эти настройки указываются при передаче методу аргумента `CultureInfo.CurrentCulture`. Класс `CultureInfo` относится к пространству имен `System.Globalization`. Любопытно, имеются варианты рассматриваемых здесь методов, в которых текущая культурная среда используется по умолчанию, но во избежание неоднозначности в примерах из этой книги аргумент `CultureInfo.CurrentCulture` указывается явно.

Объекты типа `string` содержат также свойство `Length`, где хранится длина строки.

Таблица 7.1. Некоторые общеупотребительные методы обращения со строками

Метод	Описание
`static int Compare(string strA, string strB, StringComparison comparisonType)`	Возвращает отрицательное значение, если строка *strA* меньше строки *strB*; положительное значение, если строка *strA* больше строки *strB*; и нуль, если сравниваемые строки равны. Способ сравнения определяется аргументом *comparisonType*
`bool Equals(string value, StringComparison comparisonType)`	Возвращает логическое значение `true`, если вызывающая строка имеет такое же значение, как и у аргумента *value*. Способ сравнения определяется аргументом *comparisonType*
`int IndexOf(char value)`	Осуществляет поиск в вызывающей строке первого вхождения символа, определяемого аргументом *value*. Применяется порядковый способ поиска. Возвращает индекс первого совпадения с искомым символом или -1, если он не обнаружен
`int IndexOf(string value, StringComparison comparisonType)`	Осуществляет поиск в вызывающей строке первого вхождения подстроки, определяемой аргументом *value*. Возвращает индекс первого совпадения с искомой подстрокой или -1, если она не обнаружена. Способ поиска определяется аргументом *comparisonType*

Окончание табл. 7.1

Метод	Описание
`int LastIndexOf(char value)`	Осуществляет поиск в вызывающей строке последнего вхождения символа, определяемого аргументом `value`. Применяется порядковый способ поиска. Возвращает индекс последнего совпадения с искомым символом или -1, если он не обнаружен
`int LastIndexOf(string value, StringComparison comparisonType)`	Осуществляет поиск в вызывающей строке последнего вхождения подстроки, определяемой аргументом `value`. Возвращает индекс последнего совпадения с искомой подстрокой или -1, если она не обнаружена. Способ поиска определяется аргументом `comparisonType`
`string ToLower(CultureInfo. CurrentCulture culture)`	Возвращает вариант вызывающей строки в нижнем регистре. Способ преобразования определяется аргументом `culture`
`string ToUpper(CultureInfo. CurrentCulture culture)`	Возвращает вариант вызывающей строки в верхнем регистре. Способ преобразования определяется аргументом `culture`

Отдельный символ выбирается из строки с помощью индекса, как в приведенном ниже фрагменте кода.

```
string str = "тест";
Console.WriteLine(str[0]);
```

В этом фрагменте кода выводится символ `"т"`, который является первым в строке `"тест"`. Как и в массивах, индексирование строк начинается с нуля. Следует, однако, иметь в виду, что с помощью индекса нельзя присвоить новое значение символу в строке. Индекс может служить только для выборки символа из строки.

Для проверки двух строк на равенство служит оператор ==. Как правило, если оператор == применяется к ссылкам на объект, то он определяет, являются ли они ссылками на один и тот же объект. Совсем иначе обстоит дело с объектами типа `string`. Когда оператор == применяется к ссылкам на две строки, он сравнивает содержимое этих строк. Это же относится и к оператору `!=`. В обоих случаях выполняется порядковое сравнение. Для проверки двух строк на равенство с учетом культурной среды служит метод `Equals()`, где непременно нужно указать способ сравнения в виде аргумента `StringComparison.CurrentCulture`. Следует также иметь в виду, что метод `Compare()` служит для сравнения строк с целью определить отношение порядка, например для сортировки. Если же требуется проверить символьные строки на равенство, то для этой цели лучше воспользоваться методом `Equals()` или строковыми операторами.

В приведенном ниже примере программы демонстрируется несколько операций со строками.

```
// Некоторые операции со строками.

using System;
using System.Globalization;
```

```
class StrOps {
  static void Main() {
    string str1 = "Программировать в .NET лучше всего на C#.";
    string str2 = "Программировать в .NET лучше всего на C#.";
    string str3 = "Строки в C# весьма эффективны.";
    string strUp, strLow;
    int result, idx;

    Console.WriteLine("str1: " + str1);
    Console.WriteLine("Длина строки str1: " + str1.Length);

    // Создать варианты строки str1, набранные
    // прописными и строчными буквами.
    strLow = str1.ToLower(CultureInfo.CurrentCulture);
    strUp = str1.ToUpper(CultureInfo.CurrentCulture);
    Console.WriteLine("Вариант строки str1, " +
                      "набранный строчными буквами:\n    " + strLow);
    Console.WriteLine("Вариант строки str1, " +
                      "набранный прописными буквами:\n    " + strUp);

    Console.WriteLine();

    // Вывести строку str1 посимвольно.
    Console.WriteLine("Вывод строки str1 посимвольно.");
    for(int i=0; i < str1.Length; i++)
      Console.Write(str1[i]);

    Console.WriteLine("\n");
    // Сравнить строки способом порядкового сравнения.
    if(str1 == str2)
      Console.WriteLine("str1 == str2");
    else
      Console.WriteLine("str1 != str2");
    if(str1 == str3)
      Console.WriteLine("str1 == str3");
    else
      Console.WriteLine("str1 != str3");
    // Сравнить строки с учетом культурной среды.
    result = string.Compare(str3, str1, StringComparison.CurrentCulture);
    if(result == 0)
      Console.WriteLine("Строки str1 и str3 равны");
    else if(result < 0)
      Console.WriteLine("Строка str1 меньше строки str3");
    else
      Console.WriteLine("Строка str1 больше строки str3");

    Console.WriteLine();

    // Присвоить новую строку переменной str2.
    str2 = "Один Два Три Один";

    // Поиск подстроки.
    idx = str2.IndexOf("Один", StringComparison.Ordinal);
    Console.WriteLine("Индекс первого вхождения подстроки <Один>: " + idx);
```

```
    idx = str2.LastIndexOf("Один", StringComparison.Ordinal);
    Console.WriteLine("Индекс последнего вхождения подстроки <Один>: " +
idx);
  }
}
```

При выполнении этой программы получается следующий результат.

```
str1: Программировать в .NET лучше всего на C#.
Длина строки str1: 41
Вариант строки str1, набранный строчными буквами:
    программировать в .net лучше всего на c#.
Вариант строки str1, набранный прописными буквами:
    ПРОГРАММИРОВАТЬ В .NET ЛУЧШЕ ВСЕГО НА C#.

Вывод строки str1 посимвольно.
Программировать в .NET лучше всего на C#.

str1 == str2
str1 != str3
Строка str1 больше строки str3
Индекс первого вхождения подстроки <Один>: 0
Индекс последнего вхождения подстроки <Один>: 13
```

Прежде чем читать дальше, обратите внимание на то, что метод `Compare()` вызывается следующим образом.

```
result = string.Compare(str1, str3, StringComparison.CurrentCulture);
```

Как пояснялось ранее, метод `Compare()` объявляется как `static`, и поэтому он вызывается по имени, а не по экземпляру своего класса.

С помощью оператора `+` можно сцепить (т.е. объединить вместе) две строки. Например, в следующем фрагменте кода:

```
string str1 = "Один";
string str2 = "Два";
string str3 = "Три";
string str4 = str1 + str2 + str3;
```

переменная `str4` инициализируется строкой `"ОдинДваТри"`.

И еще одно замечание: ключевое слово `string` является *псевдонимом* класса `System.String`, определенного в библиотеке классов для среды .NET Framework, т.е. оно устанавливает прямое соответствие с этим классом. Следовательно, поля и методы, определяемые типом `string`, относятся непосредственно к классу `System.String`, в который входят и многие другие компоненты. Подробнее о классе `System.String` речь пойдет в части II этой книги.

Массивы строк

Аналогично данным любого другого типа, строки могут быть организованы в массивы. Ниже приведен соответствующий пример.

```
// Продемонстрировать массивы строк.

using System;
```

```
class StringArrays {
  static void Main() {
    string[] str = { "Это", "очень", "простой", "тест." };

    Console.WriteLine("Исходный массив: ");
    for(int i=0; i < str.Length; i++)
      Console.Write(str[i] + " ");
    Console.WriteLine("\n");

    // Изменить строку.
    str[1] = "тоже";
    str[3] = "до предела тест!";

    Console.WriteLine("Видоизмененный массив: ");
    for(int i=0; i < str.Length; i++)
      Console.Write(str[i] + " ");
  }
}
```

Вот какой результат дает выполнение приведенного выше кода.

```
Исходный массив:
Это очень простой тест.

Видоизмененный массив:
Это тоже простой до предела тест!
```

Рассмотрим более интересный пример. В приведенной ниже программе целое число выводится словами. Например, число 19 выводится словами "один девять".

```
// Вывести отдельные цифры целого числа словами.

using System;

class ConvertDigitsToWords {
  static void Main() {
    int num;
    int nextdigit;
    int numdigits;
    int[] n = new int[20];

    string[] digits = { "нуль", "один", "два",
                        "три", "четыре", "пять",
                        "шесть", "семь", "восемь",
                         "девять" };

    num = 1908;

    Console.WriteLine("Число: " + num);

    Console.Write("Число словами: ");

    nextdigit = 0;
    numdigits = 0;

    // Получить отдельные цифры и сохранить их в массиве n.
```

```
    // Эти цифры сохраняются в обратном порядке.
    do {
      nextdigit = num % 10;
      n[numdigits] = nextdigit;
      numdigits++;
      num = num / 10;
    } while(num > 0);
    numdigits--;

    // Вывести полученные слова.
    for( ; numdigits >= 0; numdigits--)
      Console.Write(digits[n[numdigits]] + " ");

    Console.WriteLine();
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Число: 1908
Число словами: один девять нуль восемь
```

В данной программе использован массив строк `digits` для хранения словесных обозначений цифр от 0 до 9. По ходу выполнения программы целое число преобразуется в слова. Для этого сначала получаются отдельные цифры числа, а затем они сохраняются в обратном порядке следования в массиве `n` типа `int`. После этого выполняется циклический опрос массива `n` в обратном порядке. При этом каждое целое значение из массива `n` служит в качестве индекса, указывающего на слова, соответствующие полученным цифрам числа и выводимые как строки.

Постоянство строк

Как ни странно, содержимое объекта типа `string` не подлежит изменению. Это означает, что однажды созданную последовательность символов изменить нельзя. Но данное ограничение способствует более эффективной реализации символьных строк. Поэтому этот, на первый взгляд, очевидный недостаток на самом деле превращается в преимущество. Так, если требуется строка в качестве разновидности уже имеющейся строки, то для этой цели следует создать новую строку, содержащую все необходимые изменения. А поскольку неиспользуемые строковые объекты автоматически собираются в "мусор", то о дальнейшей судьбе ненужных строк можно даже не беспокоиться.

Следует, однако, подчеркнуть, что переменные ссылки на строки (т.е. объекты типа `string`) подлежат изменению, а следовательно, они могут ссылаться на другой объект. Но содержимое самого объекта типа `string` не меняется после его создания.

Для того чтобы стало понятнее, почему неизменяемые строки не являются помехой, воспользуемся еще одним методом обращения со строками: `Substring()`. Этот метод возвращает новую строку, содержащую часть вызывающей строки. В итоге создается новый строковый объект, содержащий выбранную подстроку, тогда как исходная строка не меняется, а следовательно, соблюдается принцип постоянства строк. Ниже приведена рассматриваемая здесь форма метода `Substring()`:

```
string Substring(int индекс_начала, int длина)
```

где *индекс_начала* обозначает начальный индекс исходной строки, а *длина* — длину выбираемой подстроки.

Ниже приведена программа, в которой принцип постоянства строк демонстрируется на примере использования метода `Substring()`.

```
// Применить метод Substring().

using System;

class SubStr {
  static void Main() {
    string orgstr = "В C# упрощается обращение со строками.";

    // сформировать подстроку
    string substr = orgstr.Substring(5, 20);

    Console.WriteLine("orgstr: " + orgstr);
    Console.WriteLine("substr: " + substr);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
orgstr: В C# упрощается обращение со строками.
substr: упрощается обращение
```

Как видите, исходная строка из переменной `orgstr` не меняется, а выбранная из нее подстрока содержится в переменной `substr`.

И последнее замечание: несмотря на то, что постоянство строк обычно не является ни ограничением, ни помехой для программирования на C#, иногда оказывается полезно иметь возможность видоизменять строки. Для этой цели в C# имеется класс `StringBuilder`, который определен в пространстве имен `System.Text`. Этот класс позволяет создавать строковые объекты, которые можно изменять. Но, как правило, в программировании на C# используется тип `string`, а не класс `StringBuilder`.

Применение строк в операторах `switch`

Объекты типа `string` могут использоваться для управления оператором `switch`. Это единственный нецелочисленный тип данных, который допускается применять в операторе `switch`. Благодаря такому применению строк в некоторых сложных ситуациях удается найти более простой выход из положения, чем может показаться на первый взгляд. Например, в приведенной ниже программе выводятся отдельные цифры, соответствующие словам "один", "два" и "три".

```
// Продемонстрировать управление оператором switch посредством строк.

using System;

class StringSwitch {
  static void Main() {
    string[] strs = { "один", "два", "три", "два", "один" };

    foreach(string s in strs) {
      switch(s) {
        case "один":
          Console.Write(1);
```

```
            break;
          case "два":
            Console.Write(2);
            break;
          case "три":
            Console.Write(3);
            break;
        }
      }

      Console.WriteLine();
    }
}
```

При выполнении этой программы получается следующий результат.

```
12321
```

ГЛАВА 8

Подробнее о методах и классах

В данной главе возобновляется рассмотрение классов и методов. Оно начинается с пояснения механизма управления доступом к членам класса. А затем обсуждаются такие вопросы, как передача и возврат объектов, перегрузка методов, различные формы метода `Main()`, рекурсия и применение ключевого слова `static`.

Управление доступом к членам класса

Поддержка свойства инкапсуляции в классе дает два главных преимущества. Во-первых, класс связывает данные с кодом. Это преимущество использовалось в предыдущих примерах программ, начиная с главы 6. И во-вторых, класс предоставляет средства для управления доступом к его членам. Именно эта, вторая преимущественная особенность и будет рассмотрена ниже.

В языке C#, по существу, имеются два типа членов класса: открытые и закрытые, хотя в действительности дело обстоит немного сложнее. Доступ к открытому члену свободно осуществляется из кода, определенного за пределами класса. Именно этот тип члена класса использовался в рассматривавшихся до сих пор примерах программ. А закрытый член класса доступен только методам, определенным в самом классе. С помощью закрытых членов и организуется управление доступом.

Ограничение доступа к членам класса является основополагающим этапом объектно-ориентированного программирования, поскольку позволяет исключить неверное использование объекта. Разрешая доступ к закрытым

данным только с помощью строго определенного ряда методов, можно предупредить присваивание неверных значений этим данным, выполняя, например, проверку диапазона представления чисел. Для закрытого члена класса нельзя задать значение непосредственно в коде за пределами класса. Но в то же время можно полностью управлять тем, как и когда данные используются в объекте. Следовательно, правильно реализованный класс образует некий "черный ящик", которым можно пользоваться, но внутренний механизм его действия закрыт для вмешательства извне.

Модификаторы доступа

Управление доступом в языке C# организуется с помощью четырех *модификаторов доступа*: `public`, `private`, `protected` и `internal`. В этой главе основное внимание уделяется модификаторам доступа `public` и `private`. Модификатор `protected` применяется только в тех случаях, которые связаны с наследованием, и поэтому речь о нем пойдет в главе 11. А модификатор `internal` служит в основном для *сборки*, которая в широком смысле означает в C# разворачиваемую программу или библиотеку, и поэтому данный модификатор подробнее рассматривается в главе 16.

Когда член класса обозначается спецификатором `public`, он становится доступным из любого другого кода в программе, включая и методы, определенные в других классах. Когда же член класса обозначается спецификатором `private`, он может быть доступен только другим членам этого класса. Следовательно, методы из других классов не имеют доступа к закрытому члену (`private`) данного класса. Как пояснялось в главе 6, если ни один из спецификаторов доступа не указан, член класса считается закрытым для своего класса по умолчанию. Поэтому при создании закрытых членов класса спецификатор `private` указывать для них необязательно.

Спецификатор доступа указывается перед остальной частью описания типа отдельного члена. Это означает, что именно с него должен начинаться оператор объявления члена класса. Ниже приведены соответствующие примеры.

```
public string errMsg;
private double bal;
private bool isError(byte status) { // ...
```

Для того чтобы стали более понятными отличия между модификаторами `public` и `private`, рассмотрим следующий пример программы.

```
// Отличия между видами доступа public и private к членам класса.

using System;

class MyClass {
  private int alpha; // закрытый доступ, указываемый явно
  int beta;          // закрытый доступ по умолчанию
  public int gamma;  // открытый доступ

  // Методы, которым доступны члены alpha и beta данного класса.
  // Член класса может иметь доступ к закрытому члену этого же класса.

  public void SetAlpha(int a) {
    alpha = a;
  }
```

```
  public int GetAlpha() {
    return alpha;
  }

  public void SetBeta(int a) {
    beta = a;
  }

  public int GetBeta() {
    return beta;
  }
}

class AccessDemo {
  static void Main() {
    MyClass ob = new MyClass();

    // Доступ к членам alpha и beta данного класса
    // разрешен только посредством его методов.
    ob.SetAlpha(-99);
    ob.SetBeta(19);
    Console.WriteLine("ob.alpha равно " + ob.GetAlpha());
    Console.WriteLine("ob.beta равно " + ob.GetBeta());

    // Следующие виды доступа к членам alpha и beta
    // данного класса не разрешаются.
// ob.alpha = 10; // Ошибка! alpha - закрытый член!
// ob.beta = 9;   // Ошибка! beta - закрытый член!

    // Член gamma данного класса доступен непосредственно,
    // поскольку он является открытым.
    ob.gamma = 99;
    }
}
```

Как видите, в классе `MyClass` член `alpha` указан явно как `private`, член `beta` становится `private` по умолчанию, а член `gamma` указан как `public`. Таким образом, члены `alpha` и `beta` недоступны непосредственно из кода за пределами данного класса, поскольку они являются закрытыми. В частности, ими нельзя пользоваться непосредственно в классе `AccessDemo`. Они доступны только с помощью таких открытых (`public`) методов, как `SetAlpha()` и `GetAlpha()`. Так, если удалить символы комментария в начале следующей строки кода:

```
// ob.alpha = 10; // Ошибка! alpha - закрытый член!
```

то приведенная выше программа не будет скомпилирована из-за нарушения правил доступа. Но несмотря на то, что член `alpha` недоступен непосредственно за пределами класса `MyClass`, свободный доступ к нему организуется с помощью методов, определенных в классе `MyClass`, как наглядно показывают методы `SetAlpha()` и `GetAlpha()`. Это же относится и к члену `beta`.

Из всего сказанного выше можно сделать следующий важный вывод: закрытый член может свободно использоваться другими членами этого же класса, но недоступен для кода за пределами своего класса.

Организация закрытого и открытого доступа

Правильная организация закрытого и открытого доступа — залог успеха в объектно-ориентированном программировании. И хотя для этого не существует твердо установленных правил, ниже перечислен ряд общих принципов, которые могут служить в качестве руководства к действию.

- Члены, используемые только в классе, должны быть закрытыми.
- Данные экземпляра, не выходящие за определенные пределы значений, должны быть закрытыми, а при организации доступа к ним с помощью открытых методов следует выполнять проверку диапазона представления чисел.
- Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым.
- Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование.
- Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми.
- Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми.

Разумеется, существует немало ситуаций, на которые приведенные выше принципы не распространяются, а в особых случаях один или несколько этих принципов могут вообще нарушаться. Но в целом, следуя этим правилам, вы сможете создавать объекты, устойчивые к попыткам неправильного их использования.

Практический пример организации управления доступом

Для чтобы стали понятнее особенности внутреннего механизма управления доступом, обратимся к конкретному примеру. Одним из самых характерных примеров объектно-ориентированного программирования служит класс, реализующий *стек* — структуру данных, воплощающую магазинный список, действующий по принципу "первым пришел — последним обслужен". Свое название он получил по аналогии со стопкой тарелок, стоящих на столе. Первая тарелка в стопке является в то же время последней использовавшейся тарелкой.

Стек служит классическим примером объектно-ориентированного программирования потому, что он сочетает в себе средства хранения информации с методами доступа к ней. Для реализации такого сочетания отлично подходит класс, в котором члены, обеспечивающие хранение информации в стеке, должны быть закрытыми, а методы доступа к ним — открытыми. Благодаря инкапсуляции базовых средств хранения информации соблюдается определенный порядок доступа к отдельным элементам стека из кода, в котором он используется.

Для стека определены две основные операции: *поместить* данные в стек и *извлечь* их оттуда. Первая операция помещает значение на вершину стека, а вторая — извлекает значение из вершины стека. Следовательно, операция извлечения является безвозвратной: как только значение извлекается из стека, оно удаляется и уже недоступно в стеке.

В рассматриваемом здесь примере создается класс Stack, реализующий функции стека. В качестве базовых средств для хранения данных в стеке служит закрытый массив. А операции размещения и извлечения данных из стека доступны с помощью открытых методов класса Stack. Таким образом, открытые методы действуют по упомянутому выше принципу "последним пришел — первым обслужен". Как следует из приведенного ниже кода, в классе Stack сохраняются символы, но тот же самый механизм может быть использован и для хранения данных любого другого типа.

```
// Класс для хранения символов в стеке.

using System;

class Stack {
  // Эти члены класса являются закрытыми.
  char[] stck; // массив, содержащий стек
  int tos;     // индекс вершины стека

  // Построить пустой класс Stack для реализации стека заданного размера.
  public Stack(int size) {
    stck = new char[size]; // распределить память для стека
    tos = 0;
  }

  // Поместить символы в стек.
  public void Push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" - Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлечь символ из стека.
  public char Pop() {
    if(tos==0) {
      Console.WriteLine(" - Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Возвратить значение true, если стек заполнен.
  public bool IsFull() {
    return tos==stck.Length;
  }

  // Возвратить значение true, если стек пуст.
  public bool IsEmpty() {
    return tos==0;
  }
```

```
  // Возвратить общую емкость стека.
  public int Capacity() {
    return stck.Length;
  }

  // Возвратить количество объектов, находящихся в данный момент в стеке.
  public int GetNum() {
    return tos;
  }
}
```

Рассмотрим класс `Stack` более подробно. В начале этого класса объявляются две следующие переменные экземпляра.

```
// Эти члены класса являются закрытыми.
  char[] stck; // массив, содержащий стек
  int tos;     // индекс вершины стека
```

Массив `stck` предоставляет базовые средства для хранения данных в стеке (в данном случае — символов). Обратите внимание на то, что память для этого массива не распределяется. Это делается в конструкторе класса `Stack`. А член `tos` данного класса содержит индекс вершины стека.

Оба члена, `tos` и `stck`, являются закрытыми, и благодаря этому соблюдается принцип "последним пришел — первым обслужен". Если же разрешить открытый доступ к члену `stck`, то элементы стека окажутся доступными не по порядку. Кроме того, член `tos` содержит индекс вершины стека, где находится первый обслуживаемый в стеке элемент, и поэтому манипулирование членом `tos` в коде, находящемся за пределами класса `Stack`, следует исключить, чтобы не допустить разрушение самого стека. Но в то же время члены `stck` и `tos` доступны пользователю класса `Stack` косвенным образом с помощью различных отрытых методов, описываемых ниже.

Рассмотрим далее конструктор класса `Stack`.

```
// Построить пустой класс Stack для реализации стека заданного размера.
  public Stack(int size) {
    stck = new char[size]; // распределить память для стека
    tos = 0;
  }
```

Этому конструктору передается требуемый размер стека. Он распределяет память для базового массива и устанавливает значение переменной `tos` в нуль. Следовательно, нулевое значение переменной `tos` указывает на то, что стек пуст.

Открытый метод `Push()` помещает конкретный элемент в стек, как показано ниже.

```
// Поместить символы в стек.
public void Push(char ch) {
  if(tos==stck.Length) {
    Console.WriteLine(" - Стек заполнен.");
    return;
  }

  stck[tos] = ch;
  tos++;
}
```

Элемент, помещаемый в стек, передается данному методу в качестве параметра ch. Перед тем как поместить элемент в стек, выполняется проверка на наличие свободного места в базовом массиве, а именно: не превышает ли значение переменной tos длину массива stck. Если свободное место в массиве stck есть, то элемент сохраняется в нем по индексу, хранящемуся в переменной tos, после чего значение этой переменной инкрементируется. Таким образом, в переменной tos всегда хранится индекс следующего свободного элемента массива stck.

Для извлечения элемента из стека вызывается открытый метод Pop(), приведенный ниже.

```
// Извлечь символ из стека.
public char Pop() {
  if(tos==0) {
    Console.WriteLine(" - Стек пуст.");
    return (char) 0;
  }

  tos--;
  return stck[tos];
}
```

В этом методе сначала проверяется значение переменной tos. Если оно равно нулю, значит, стек пуст. В противном случае значение переменной tos декрементируется, и затем из стека возвращается элемент по указанному индексу.

Несмотря на то что для реализации стека достаточно методов Push() и Pop(), полезными могут оказаться и другие методы. Поэтому в классе Stack определены еще четыре метода: IsFull(), IsEmpty(), Capacity() и GetNum(). Эти методы предоставляют всю необходимую информацию о состоянии стека и приведены ниже.

```
// Возвратить значение true, если стек заполнен.
public bool IsFull() {
  return tos==stck.Length;
}

// Возвратить значение true, если стек пуст.
public bool IsEmpty() {
  return tos==0;
}

// Возвратить общую емкость стека.
public int Capacity() {
  return stck.Length;
}

// Возвратить количество объектов, находящихся в данный момент в стеке.
public int GetNum() {
  return tos;
}
```

Метод IsFull() возвращает логическое значение true, если стек заполнен, а иначе — логическое значение false. Метод IsEmpty() возвращает логическое значение true, если стек пуст, а иначе — логическое значение false. Для получения общей емкости стека (т.е. общего числа элементов, которые могут в нем храниться) достаточно

вызвать метод Capacity(), а для получения количества элементов, хранящихся в настоящий момент в стеке, — метод GetNum(). Польза этих методов состоит в том, что для получения информации, которую они предоставляют, требуется доступ к закрытой переменной tos. Кроме того, они служат наглядными примерами организации безопасного доступа к закрытым членам класса с помощью открытых методов.

Конкретное применение класса Stack для реализации стека демонстрируется в приведенной ниже программе.

```
// Продемонстрировать применение класса Stack.

using System;

// Класс для хранения символов в стеке.
class Stack {
  // Эти члены класса являются закрытыми.
  char[] stck; // массив, содержащий стек
  int tos;     // индекс вершины стека

  // Построить пустой класс Stack для реализации стека заданного размера.
  public Stack(int size) {
    stck = new char[size]; // распределить память для стека
    tos = 0;
  }

  // Поместить символы в стек.
  public void Push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" - Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлечь символ из стека.
  public char Pop() {
    if(tos==0) {
      Console.WriteLine(" - Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Возвратить значение true, если стек заполнен.
  public bool IsFull() {
    return tos==stck.Length;
  }

  // Возвратить значение true, если стек пуст.
  public bool IsEmpty() {
```

```
    return tos==0;
  }

  // Возвратить общую емкость стека.
  public int Capacity() {
    return stck.Length;
  }

  // Возвратить количество объектов, находящихся в данный момент в стеке.
  public int GetNum() {
    return tos;
  }
}

class StackDemo {
  static void Main() {
    Stack stk1 = new Stack(10);
    Stack stk2 = new Stack(10);
    Stack stk3 = new Stack(10);
    char ch;
    int i;

    // Поместить ряд символов в стек stk1.
    Console.WriteLine("Поместить символы A-J в стек stk1.");
    for(i=0; !stk1.IsFull(); i++)
      stk1.Push((char) ('A' + i));

    if(stk1.IsFull()) Console.WriteLine("Стек stk1 заполнен.");

    // Вывести содержимое стека stk1.
    Console.Write("Содержимое стека stk1: ");
    while( !stk1.IsEmpty() ) {
      ch = stk1.Pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    if(stk1.IsEmpty()) Console.WriteLine("Стек stk1 пуст.\n");

    // Поместить дополнительные символы в стек stk1.
    Console.WriteLine("Вновь поместить символы A-J в стек stk1.");
    for(i=0; !stk1.IsFull(); i++)
      stk1.Push((char) ('A' + i));

    // А теперь извлечь элементы из стека stk1 и поместить их в стек stk2.
    // В итоге элементы сохраняются в стеке stk2 в обратном порядке.
    Console.WriteLine("А теперь извлечь символы из стека stk1\n" +
                      "и поместить их в стек stk2.");
     while( !stk1.IsEmpty() ) {
       ch = stk1.Pop();
       stk2.Push(ch);
    }
```

```
    Console.Write("Содержимое стека stk2: ");
    while( !stk2.IsEmpty() ) {
      ch = stk2.Pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

    // Поместить 5 символов в стек.
    Console.WriteLine("Поместить 5 символов в стек stk3.");
    for(i=0; i < 5; i++)
      stk3.Push((char) ('A' + i));

    Console.WriteLine("Емкость стека stk3: " + stk3.Capacity());
    Console.WriteLine("Количество объектов в стеке stk3: " +
            stk3.GetNum());
  }
}
```

При выполнении этой программы получается следующий результат.

```
Поместить символы A-J в стек stk1.
Стек stk1 заполнен.
Содержимое стека stk1: JIHGFEDCBA
Стек stk1 пуст.

Вновь поместить символы A-J в стек stk1.
А теперь извлечь символы из стека stk1
и поместить их в стек stk2.
Содержимое стека stk2: ABCDEFGHIJ

Поместить 5 символов в стек stk3.
Емкость стека stk3: 10
Количество объектов в стеке stk3: 5
```

Передача объектов методам по ссылке

В приведенных до сих пор примерах программ при указании параметров, передаваемых методам, использовались типы значений, например `int` или `double`. Но в методах можно также использовать параметры ссылочного типа, что не только правильно, но и весьма распространено в ООП. Подобным образом объекты могут передаваться методам по ссылке. В качестве примера рассмотрим следующую программу.

```
// Пример передачи объектов методам по ссылке.

using System;

class MyClass {
  int alpha, beta;

  public MyClass(int i, int j) {
    alpha = i;
    beta = j;
  }
```

```
  // Возвратить значение true, если параметр ob
  // имеет те же значения, что и вызывающий объект.
  public bool SameAs(MyClass ob) {
    if((ob.alpha == alpha) & (ob.beta == beta))
      return true;
    else return false;
  }

  // Сделать копию объекта ob.
  public void Copy(MyClass ob) {
    alpha = ob.alpha;
    beta = ob.beta;
  }

  public void Show() {
    Console.WriteLine("alpha: {0}, beta: {1}",
                      alpha, beta);
  }
}

class PassOb {
  static void Main() {
    MyClass ob1 = new MyClass(4, 5);
    MyClass ob2 = new MyClass(6, 7);

    Console.Write("ob1: ");
    ob1.Show();

    Console.Write("ob2: ");
    ob2.Show();

    if(ob1.SameAs(ob2))
      Console.WriteLine("ob1 и ob2 имеют одинаковые значения.");
    else
    Console.WriteLine("ob1 и ob2 имеют разные значения.");

    Console.WriteLine();

    // А теперь сделать объект ob1 копией объекта ob2.
    ob1.Copy(ob2);

    Console.Write("ob1 после копирования: ");
    ob1.Show();

    if(ob1.SameAs(ob2))
      Console.WriteLine("ob1 и ob2 имеют одинаковые значения.");
    else
      Console.WriteLine("ob1 и ob2 имеют разные значения.");
  }
}
```

Выполнение этой программы дает следующий результат.

```
ob1: alpha: 4, beta: 5
ob2: alpha: 6, beta: 7
```

```
ob1 и ob2 имеют разные значения.

ob1 после копирования: alpha: 6, beta: 7
ob1 и ob2 имеют одинаковые значения.
```

Каждый из методов `SameAs()` и `Copy()` в приведенной выше программе получает ссылку на объект типа `MyClass` в качестве аргумента. Метод `SameAs()` сравнивает значения переменных экземпляра `alpha` и `beta` в вызывающем объекте со значениями аналогичных переменных в объекте, передаваемом посредством параметра `ob`. Данный метод возвращает логическое значение `true` только в том случае, если оба объекта имеют одинаковые значения этих переменных экземпляра. А метод `Copy()` присваивает значения переменных `alpha` и `beta` из объекта, передаваемого по ссылке посредством параметра `ob`, переменным `alpha` и `beta` из вызывающего объекта. Как показывает данный пример, с точки зрения синтаксиса объекты передаются методам по ссылке таким же образом, как и значения обычных типов.

Способы передачи аргументов методу

Как показывает приведенный выше пример, передача объекта методу по ссылке делается достаточно просто. Но в этом примере показаны не все нюансы данного процесса. В некоторых случаях последствия передачи объекта по ссылке будут отличаться от тех результатов, к которым приводит передача значения обычного типа. Для выяснения причин этих отличий рассмотрим два способа передачи аргументов методу.

Первым способом является *вызов по значению*. В этом случае значение аргумента копируется в формальный параметр метода. Следовательно, изменения, вносимые в параметр метода, не оказывают никакого влияния на аргумент, используемый для вызова. А вторым способом передачи аргумента является *вызов по ссылке*. В данном случае параметру метода передается ссылка на аргумент, а не значение аргумента. В методе эта *ссылка* используется для доступа к конкретному аргументу, указываемому при вызове. Это означает, что изменения, вносимые в параметр, будут оказывать влияние на аргумент, используемый для вызова метода.

По умолчанию в C# используется вызов по значению, а это означает, что копия аргумента создается и затем передается принимающему параметру. Следовательно, при передаче значения обычного типа, например `int` или `double`, все, что происходит с параметром, принимающим аргумент, не оказывает никакого влияния за пределами метода. В качестве примера рассмотрим следующую программу.

```
// Передача аргументов обычных типов по значению.

using System;

class Test {
  /* Этот метод не оказывает никакого влияния на
     аргументы, используемые для его вызова. */
  public void NoChange(int i, int j) {
    i = i + j;
    j = -j;
  }
}

class CallByValue {
```

```
  static void Main() {
    Test ob = new Test();

    int a = 15, b = 20;

    Console.WriteLine("a и b до вызова: " +
                      a + " " + b);

    ob.NoChange(a, b);

    Console.WriteLine("a и b после вызова: " +
                      a + " " + b);
  }
}
```

Вот какой результат дает выполнение этой программы.

```
a и b до вызова: 15 20
a и b после вызова: 15 20
```

Как видите, операции, выполняемые в методе `NoChange()`, не оказывают никакого влияния на значения аргументов `a` и `b`, используемых для вызова данного метода. Это опять же объясняется тем, что параметрам `i` и `j` переданы копии значений аргументов `a` и `b`, а сами аргументы `a` и `b` совершенно не зависят от параметров `i` и `j`. В частности, присваивание параметру `i` нового значения не будет оказывать никакого влияния на аргумент `a`.

Дело несколько усложняется при передаче методу ссылки на объект. В этом случае сама ссылка по-прежнему передается по значению. Следовательно, создается копия ссылки, а изменения, вносимые в параметр, не оказывают никакого влияния на аргумент. (Так, если организовать ссылку параметра на новый объект, то это изменение не повлечет за собой никаких последствий для объекта, на который ссылается аргумент.) Но главное отличие вызова по ссылке заключается в том, что изменения, происходящие с объектом, на который ссылается параметр, *окажут влияние* на тот объект, на который ссылается аргумент. Попытаемся выяснить причины подобного влияния.

Напомним, что при создании переменной типа класса формируется только ссылка на объект. Поэтому при передаче этой ссылки методу принимающий ее параметр будет ссылаться на тот же самый объект, на который ссылается аргумент. Это означает, что и аргумент, и параметр ссылаются на один и тот же объект и что объекты, по существу, передаются методам по ссылке. Таким образом, объект в методе *будет* оказывать влияние на объект, используемый в качестве аргумента. Для примера рассмотрим следующую программу.

```
// Передача объектов по ссылке.

using System;

class Test {
  public int a, b;

  public Test(int i, int j) {
    a = i;
    b = j;
  }
```

```
  /* Передать объект. Теперь переменные ob.a и ob.b из объекта,
     используемого в вызове метода, будут изменены. */
  public void Change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}

class CallByRef {
  static void Main() {
    Test ob = new Test(15, 20);

    Console.WriteLine("ob.a и ob.b до вызова: " +
                      ob.a + " " + ob.b);

    ob.Change(ob);

    Console.WriteLine("ob.a и ob.b после вызова: " +
                      ob.a + " " + ob.b);
  }
}
```

Выполнение этой программы дает следующий результат.

```
ob.a и ob.b до вызова: 15 20
ob.a и ob.b после вызова: 35 -20
```

Как видите, действия в методе `Change()` оказали в данном случае влияние на объект, использовавшийся в качестве аргумента.

Итак, подведем краткий итог. Когда объект передается методу по ссылке, сама ссылка передается по значению, а следовательно, создается копия этой ссылки. Но эта копия будет по-прежнему ссылаться на тот же самый объект, что и соответствующий аргумент. Это означает, что объекты передаются методам неявным образом по ссылке.

Использование модификаторов параметров `ref` и `out`

Как пояснялось выше, аргументы простых типов, например `int` или `char`, передаются методу по значению. Это означает, что изменения, вносимые в параметр, принимающий значение, не будут оказывать никакого влияния на аргумент, используемый для вызова. Но такое поведение можно изменить, используя ключевые слова `ref` и `out` для передачи значений обычных типов по ссылке. Это позволяет изменить в самом методе аргумент, указываемый при его вызове.

Прежде чем переходить к особенностям использования ключевых слов `ref` и `out`, полезно уяснить причины, по которым значение простого типа иногда требуется передавать по ссылке. В общем, для этого существуют две причины: разрешить методу изменить содержимое его аргументов или же возвратить несколько значений. Рассмотрим каждую из этих причин более подробно.

Нередко требуется, чтобы метод оперировал теми аргументами, которые ему передаются. Характерным тому примером служит метод `Swap()`, осуществляющий перестановку значений своих аргументов. Но поскольку аргументы простых типов пере-

даются по значению, то, используя выбираемый в C# по умолчанию механизм вызова по значению для передачи аргумента параметру, невозможно написать метод, меняющий местами значения двух его аргументов, например типа `int`. Это затруднение разрешает модификатор `ref`.

Как вам должно быть уже известно, значение возвращается из метода вызывающей части программы с помощью оператора `return`. Но метод может одновременно возвратить лишь *одно* значение. А что, если из метода требуется возвратить два или более фрагментов информации, например, целую и дробную части числового значения с плавающей точкой? Такой метод можно написать, используя модификатор `out`.

Использование модификатора параметра `ref`

Модификатор параметра `ref` принудительно организует вызов по ссылке, а не по значению. Этот модификатор указывается как при объявлении, так и при вызове метода. Для начала рассмотрим простой пример. В приведенной ниже программе создается метод `Sqr()`, возвращающий вместо своего аргумента квадрат его целочисленного значения. Обратите особое внимание на применение и местоположение модификатора `ref`.

```
// Использовать модификатор ref для передачи значения обычного типа по ссылке.

using System;

class RefTest {
  // Этот метод изменяет свой аргумент. Обратите
  // внимание на применение модификатора ref.
  public void Sqr(ref int i) {
    i = i * i;
  }
}

class RefDemo {
  static void Main() {
    RefTest ob = new RefTest();

   int a = 10;

   Console.WriteLine("a до вызова: " + a);

    ob.Sqr(ref a); // обратите внимание на применение модификатора ref

    Console.WriteLine("a после вызова: " + a);
  }
}
```

Как видите, модификатор `ref` указывается перед объявлением параметра в самом методе и перед аргументом при вызове метода. Ниже приведен результат выполнения данной программы, который подтверждает, что значение аргумента `a` действительно было изменено с помощью метода `Sqr()`.

```
a до вызова: 10
a после вызова: 100
```

Теперь, используя модификатор `ref`, можно написать метод, переставляющий местами значения двух своих аргументов простого типа. В качестве примера ниже приведена программа, в которой метод `Swap()` выполняет перестановку значений двух своих целочисленных аргументов, когда он вызывается.

```
// Поменять местами два значения.

using System;

class ValueSwap {
  // Этот метод меняет местами свои аргументы.
  public void Swap(ref int a, ref int b) {
    int t;
    t = a;
    a = b;
    b = t;
  }
}

class ValueSwapDemo {
  static void Main() {
    ValueSwap ob = new ValueSwap();

    int x = 10, y = 20;

    Console.WriteLine("x и y до вызова: " + x + " " + y);

    ob.Swap(ref x, ref y);

    Console.WriteLine("x и y после вызова: " + x + " " + y);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
x и y до вызова: 10 20
x и y после вызова: 20 10
```

В отношении модификатора `ref` необходимо иметь в виду следующее. Аргументу, передаваемому по ссылке с помощью этого модификатора, должно быть присвоено значение *до* вызова метода. Дело в том, что в методе, получающем такой аргумент в качестве параметра, предполагается, что параметр ссылается на действительное значение. Следовательно, при использовании модификатора `ref` в методе нельзя задать первоначальное значение аргумента.

Использование модификатора параметра `out`

Иногда ссылочный параметр требуется использовать для получения значения из метода, а не для передачи ему значения. Допустим, что имеется метод, выполняющий некоторую функцию, например, открытие сетевого сокета и возврат кода успешного или неудачного завершения данной операции в качестве ссылочного параметра. В этом случае методу не передается никакой информации, но в то же время он должен возвратить определенную информацию. Главная трудность при этом состоит в том,

что параметр типа `ref` должен быть инициализирован определенным значением до вызова метода. Следовательно, чтобы воспользоваться параметром типа `ref`, придется задать для аргумента фиктивное значение и тем самым преодолеть данное ограничение. Правда, в C# имеется более подходящий вариант выхода из подобного затруднения — воспользоваться модификатором параметра `out`.

Модификатор параметра `out` подобен модификатору `ref`, за одним исключением: он служит только для передачи значения за пределы метода. Поэтому переменной, используемой в качестве параметра `out`, не нужно (да и бесполезно) присваивать какое-то значение. Более того, в методе параметр `out` считается *неинициализированным*, т.е. предполагается, что у него отсутствует первоначальное значение. Это означает, что значение должно быть присвоено данному параметру в методе *до* его завершения. Следовательно, после вызова метода параметр `out` будет содержать некоторое значение.

Ниже приведен пример применения модификатора параметра `out`. В этом примере программы для разделения числа с плавающей точкой на целую и дробную части используется метод `GetParts()` из класса `Decompose`. Обратите внимание на то, как возвращается каждая часть исходного числа.

```
// Использовать модификатор параметра out.

using System;

class Decompose {

  /* Разделить числовое значение с плавающей точкой на
     целую и дробную части. */
  public int GetParts(double n, out double frac) {
    int whole;

    whole = (int) n;
    frac = n - whole; // передать дробную часть числа через параметр frac
    return whole;     // возвратить целую часть числа
  }
}

class UseOut {
  static void Main() {
    Decompose ob = new Decompose();
    int i;
    double f;

    i = ob.GetParts(10.125, out f);

    Console.WriteLine("Целая часть числа равна " + i);
    Console.WriteLine("Дробная часть числа равна " + f);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Целая часть числа равна 10
Дробная часть числа равна 0.125
```

Метод GetParts() возвращает два фрагмента информации. Во-первых, целую часть исходного числового значения переменной n обычным образом с помощью оператора return. И во-вторых, дробную часть этого значения посредством параметра frac типа out. Как показывает данный пример, используя модификатор параметра out, можно организовать возврат двух значений из одного и того же метода.

Разумеется, никаких ограничений на применение параметров out в одном методе не существует. С их помощью из метода можно возвратить сколько угодно фрагментов информации. Рассмотрим пример применения двух параметров out. В этом примере программы метод HasComFactor() выполняет две функции. Во-первых, он определяет общий множитель (кроме 1) для двух целых чисел, возвращая логическое значение true, если у них имеется общий множитель, а иначе — логическое значение false. И во-вторых, он возвращает посредством параметров типа out наименьший и наибольший общий множитель двух чисел, если таковые обнаруживаются.

```
// Использовать два параметра типа out.

using System;

class Num {
  /* Определить, имеется ли у числовых значений переменных x и v
     общий множитель. Если имеется, то возвратить наименьший и
     наибольший множители посредством параметров типа out. */
  public bool HasComFactor(int x, int y,
                           out int least, out int greatest) {
    int i;
    int max = x < y ? x : y;
    bool first = true;

    least = 1;
    greatest = 1;

    // Найти наименьший и наибольший общий множитель.
    for(i=2; i <= max/2 + 1; i++) {
      if( ((y%i)==0) & ((x%i)==0) ) {
        if(first) {
          least = i;
          first = false;
        }
        greatest = i;
      }
    }

    if(least != 1) return true;
    else return false;
    }
  }

class DemoOut {
  static void Main() {
    Num ob = new Num();
    int lcf, gcf;

    if(ob.HasComFactor(231, 105, out lcf, out gcf)) {
```

```
      Console.WriteLine("Наименьший общий множитель " +
                        "чисел 231 и 105 равен " + lcf);
      Console.WriteLine("Наибольший общий множитель " +
                        "чисел 231 и 105 равен " + gcf);
    }
    else
      Console.WriteLine("Общий множитель у чисел 35 и 49 отсутствует.");

    if(ob.HasComFactor(35, 51, out lcf, out gcf)) {
      Console.WriteLine("Наименьший общий множитель " +
                        "чисел 35 и 51 равен " + lcf);
     Console.WriteLine("Наибольший общий множитель " +
                       "чисел 35 и 51 равен " + gcf);
    }
    else
      Console.WriteLine("Общий множитель у чисел 35 и 51 отсутствует.");
  }
}
```

Обратите внимание на то, что значения присваиваются переменным `lcf` и `gcf` в методе `Main()` до вызова метода `HasComFactor()`. Если бы параметры метода `HasComFactor()` были типа `ref`, а не `out`, это привело бы к ошибке. Данный метод возвращает логическое значение `true` или `false`, в зависимости от того, имеется ли общий множитель у двух целых чисел. Если он имеется, то посредством параметров типа `out` возвращаются наименьший и наибольший общий множитель этих чисел. Ниже приведен результат выполнения данной программы.

```
Наименьший общий множитель чисел 231 и 105 равен 3
Наибольший общий множитель чисел 231 и 105 равен 21
Общий множитель у чисел 35 и 51 отсутствует.
```

Использование модификаторов `ref` и `out` для ссылок на объекты

Применение модификаторов `ref` и `out` не ограничивается только передачей значений обычных типов. С их помощью можно также передавать ссылки на объекты. Если модификатор `ref` или `out` указывает на ссылку, то сама ссылка передается по ссылке. Это позволяет изменить в методе объект, на который указывает ссылка. Рассмотрим в качестве примера следующую программу, в которой ссылочные параметры типа `ref` служат для смены объектов, на которые указывают ссылки.

```
// Поменять местами две ссылки.

using System;

class RefSwap {
  int a, b;

  public RefSwap(int i, int j) {
    a = i;
    b = j;
  }
```

```
  public void Show() {
    Console.WriteLine("a: {0}, b: {1}", a, b);
  }

  // Этот метод изменяет свои аргументы.
  public void Swap(ref RefSwap ob1, ref RefSwap ob2) {
    RefSwap t;

    t = ob1;
    ob1 = ob2;
    ob2 = t;
  }
}

class RefSwapDemo {
  static void Main() {
    RefSwap x = new RefSwap(1, 2);
    RefSwap y = new RefSwap(3, 4);

    Console.Write("x до вызова: ");
    x.Show();

    Console.Write("y до вызова: ");
    y.Show();

    Console.WriteLine();

    // Смена объектов, на которые ссылаются аргументы x и y.
    x.Swap(ref x, ref y);

    Console.Write("x после вызова: ");
    x.Show();

    Console.Write("y после вызова: ");
    y.Show();
  }
}
```

При выполнении этой программы получается следующий результат.

```
x до вызова: a: 1, b: 2
y до вызова: a: 3, b: 4

x после вызова: a: 3, b: 4
y после вызова: a: 1, b: 2
```

В данном примере в методе `Swap()` выполняется смена объектов, на которые ссылаются два его аргумента. До вызова метода `Swap()` аргумент `x` ссылается на объект, содержащий значения 1 и 2, тогда как аргумент `y` ссылается на объект, содержащий значения 3 и 4. А после вызова метода `Swap()` аргумент `x` ссылается на объект, содержащий значения 3 и 4, тогда как аргумент `y` ссылается на объект, содержащий значения 1 и 2. Если бы не параметры типа `ref`, то перестановка в методе `Swap()` не имела бы никаких последствий за пределами этого метода. Для того чтобы убедиться в этом, исключите параметры типа `ref` из метода `Swap()`.

Использование переменного числа аргументов

При создании метода обычно заранее известно число аргументов, которые будут переданы ему, но так бывает не всегда. Иногда возникает потребность создать метод, которому можно было бы передать произвольное число аргументов. Допустим, что требуется метод, обнаруживающий наименьшее среди ряда значений. Такому методу можно было бы передать не менее двух, трех, четырех или еще больше значений. Но в любом случае метод должен возвратить наименьшее из этих значений. Такой метод нельзя создать, используя обычные параметры. Вместо этого придется воспользоваться специальным типом параметра, обозначающим произвольное число параметров. И это делается с помощью создаваемого параметра типа `params`.

Для объявления массива параметров, способного принимать от нуля до нескольких аргументов, служит модификатор `params`. Число элементов массива параметров будет равно числу аргументов, передаваемых методу. А для получения аргументов в программе организуется доступ к данному массиву.

Ниже приведен пример программы, в которой модификатор `params` используется для создания метода `MinVal()`, возвращающего наименьшее среди ряда заданных значений.

```
// Продемонстрировать применение модификатора params.

using System;

class Min {
  public int MinVal(params int[] nums) {
    int m;

    if(nums.Length == 0) {
      Console.WriteLine("Ошибка: нет аргументов.");
      return 0;
    }

    m = nums[0];
    for(int i=1; i < nums.Length; i++)
      if(nums[i] < m) m = nums[i];

    return m;
  }
}

class ParamsDemo {
  static void Main() {
    Min ob = new Min();
    int min;
    int a = 10, b = 20;

    // Вызвать метод с двумя значениями.
    min = ob.MinVal(a, b);
    Console.WriteLine("Наименьшее значение равно " + min);

    // Вызвать метод с тремя значениями.
    min = ob.MinVal(a, b, -1);
```

```
    Console.WriteLine("Наименьшее значение равно " + min);

    // Вызвать метод с пятью значениями.
    min = ob.MinVal(18, 23, 3, 14, 25);
    Console.WriteLine("Наименьшее значение равно " + min);

    // Вызвать метод с массивом целых значений.
    int[] args = { 45, 67, 34, 9, 112, 8 };
    min = ob.MinVal(args);
    Console.WriteLine("Наименьшее значение равно " + min);
  }
}
```

При выполнении этой программы получается следующий результат.

```
Наименьшее значение равно 10
Наименьшее значение равно -1
Наименьшее значение равно 3
Наименьшее значение равно 8
```

Всякий раз, когда вызывается метод `MinVal()`, ему передаются аргументы в массиве `nums`. Длина этого массива равна числу передаваемых аргументов. Поэтому с помощью метода `MinVal()` можно обнаружить наименьшее среди любого числа значений.

Обратите внимание на последний вызов метода `MinVal()`. Вместо отдельных значений в данном случае передается массив, содержащий ряд значений. И такая передача аргументов вполне допустима. Когда создается параметр типа `params`, он воспринимает список аргументов переменной длины или же массив, содержащий аргументы.

Несмотря на то что параметру типа `params` может быть передано любое число аргументов, все они должны иметь тип массива, указываемый этим параметром. Например, вызов метода `MinVal()`

```
min = ob.MinVal(1, 2.2); // Неверно!
```

считается недопустимым, поскольку нельзя автоматически преобразовать тип `double` (значение 2.2) в тип `int`, указанный для массива `nums` в методе `MinVal()`.

Пользоваться модификатором `params` следует осторожно, соблюдая граничные условия, так как параметр типа `params` может принимать любое число аргументов — даже *нулевое*! Например, вызов метода `MinVal()` в приведенном ниже фрагменте кода считается правильным с точки зрения синтаксиса C#.

```
min = ob.MinVal(); // нет аргументов
min = ob.MinVal(3); // 1 аргумент
```

Именно поэтому в методе `MinVal()` организована проверка на наличие в массиве `nums` хотя бы одного элемента перед тем, как пытаться получить доступ к этому элементу. Если бы такой проверки не было, то при вызове метода `MinVal()` без аргументов возникла бы исключительная ситуация во время выполнения. (Подробнее об исключительных ситуациях речь пойдет в главе 13.) Больше того, код метода `MinVal()` написан таким образом, чтобы его можно было вызывать с одним аргументом. В этом случае возвращается этот единственный аргумент.

У метода могут быть как обычные параметры, так и параметр переменной длины. В качестве примера ниже приведена программа, в которой метод `ShowArgs()`

принимает один параметр типа string, а также целочисленный массив в качестве параметра типа params.

```
// Использовать обычный параметр вместе с параметром
// переменной длины типа params.

using System;

class MyClass {
  public void ShowArgs(string msg, params int[] nums) {
    Console.Write(msg + ": ");

    foreach(int i in nums)
      Console.Write(i + " ");

    Console.WriteLine();
  }
}

class ParamsDemo2 {
  static void Main() {
    MyClass ob = new MyClass();

    ob.ShowArgs("Это ряд целых чисел",
                1, 2, 3, 4, 5);

    ob.ShowArgs("А это еще два целых числа ",
                17, 20);
  }
}
```

Вот какой результат дает выполнение этой программы.

```
Это ряд целых чисел: 1, 2, 3, 4, 5
А это еще два целых числа: 17, 20
```

В тех случаях, когда у метода имеются обычные параметры, а также параметр переменной длины типа params, он должен быть указан последним в списке параметров данного метода. Но в любом случае параметр типа params должен быть единственным.

Возврат объектов из методов

Метод может возвратить данные любого типа, в том числе и тип класса. Ниже в качестве примера приведен вариант класса Rect, содержащий метод Enlarge(), в котором строится прямоугольник с теми же сторонами, что и у вызывающего объекта прямоугольника, но пропорционально увеличенными на указанный коэффициент.

```
// Возвратить объект из метода.

using System;

class Rect {
  int width;
  int height;
```

```
  public Rect(int w, int h) {
    width = w;
    height = h;
  }

  public int Area() {
    return width * height;
  }

  public void Show() {
    Console.WriteLine(width + " " + height);
  }

  /* Метод возвращает прямоугольник со сторонами, пропорционально
     увеличенными на указанный коэффициент по сравнению с вызывающим
     объектом прямоугольника. */
  public Rect Enlarge(int factor) {
    return new Rect(width * factor, height * factor);
  }
}

class RetObj {
  static void Main() {
    Rect r1 = new Rect(4, 5);

    Console.Write("Размеры прямоугольника r1: ");
    r1.Show();
    Console.WriteLine("Площадь прямоугольника r1: " + r1.Area());

    Console.WriteLine();

    // Создать прямоугольник в два раза больший прямоугольника r1.
    Rect r2 = r1.Enlarge(2);

    Console.Write("Размеры прямоугольника r2: ");
    r2.Show();
    Console.WriteLine("Площадь прямоугольника r2: " + r2.Area());
  }
}
```

Выполнение этой программы дает следующий результат.

```
Размеры прямоугольника r1: 4 5
Площадь прямоугольника r1: 20

Размеры прямоугольника r2: 8 10
Площадь прямоугольника r2: 80
```

Когда метод возвращает объект, последний продолжает существовать до тех пор, пока не останется ссылок на него. После этого он подлежит сборке как "мусор". Следовательно, объект не уничтожается только потому, что завершается создавший его метод.

Одним из практических примеров применения возвращаемых данных типа объектов служит *фабрика класса*, которая представляет собой метод, предназначенный для построения объектов его же класса. В ряде случаев предоставлять пользователям класса доступ к его конструктору нежелательно из соображений безопасности или же потому, что построение объекта зависит от некоторых внешних факторов. В подобных случаях для построения объектов используется фабрика класса. Обратимся к простому примеру.

```
// Использовать фабрику класса.

using System;

class MyClass {
  int a, b; // закрытые члены класса

  // Создать фабрику для класса MyClass.
  public MyClass Factory(int i, int j) {
    MyClass t = new MyClass();

    t.a = i;
    t.b = j;

    return t; // возвратить объект
  }

  public void Show() {
    Console.WriteLine("a и b: " + a + " " + b);
  }
}

class MakeObjects {
  static void Main() {
    MyClass ob = new MyClass();
    int i, j;

    // Сформировать объекты, используя фабрику класса.
    for(i=0, j=10; i < 10; i++, j--) {
      MyClass anotherOb = ob.Factory(i, j); // создать объект
      anotherOb.Show();
    }

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
a и b: 0 10
a и b: 1 9
a и b: 2 8
a и b: 3 7
a и b: 4 6
a и b: 5 5
a и b: 6 4
```

```
a и b: 7 3
a и b: 8 2
a и b: 9 1
```

Рассмотрим данный пример более подробно. В этом примере конструктор для класса `MyClass` не определяется, и поэтому доступен только конструктор, вызываемый по умолчанию. Это означает, что значения переменных `a` и `b` нельзя задать с помощью конструктора. Но в фабрике класса `Factory()` можно создать объекты, в которых задаются значения переменных `a` и `b`. Более того, переменные `a` и `b` являются закрытыми, и поэтому их значения могут быть заданы только с помощью фабрики класса `Factory()`.

В методе `Main()` получается экземпляр объекта класса `MyClass`, а его фабричный метод используется в цикле `for` для создания десяти других объектов. Ниже приведена строка кода, в которой создаются эти объекты.

```
MyClass anotherOb = ob.Factory(i, j); // создать объект
```

На каждом шаге итерации цикла создается переменная ссылки на объект `anotherOb`, которой присваивается ссылка на объект, формируемый фабрикой класса. По завершении каждого шага итерации цикла переменная `anotherOb` выходит за пределы области своего действия, а объект, на который она ссылается, утилизируется.

Возврат массива из метода

В C# массивы реализованы в виде объектов, а это означает, что метод может также возвратить массив. (В этом отношении C# отличается от C++, где не допускается возврат массивов из методов.) В качестве примера ниже приведена программа, в которой метод `FindFactors()` возвращает массив, содержащий множители переданного ему аргумента.

```
// Возвратить массив из метода.

using System;

class Factor {
  /* Метод возвращает массив facts, содержащий множители аргумента num.
     При возврате из метода параметр numfactors типа out будет содержать
     количество обнаруженных множителей. */
  public int[] FindFactors(int num, out int numfactors) {
    int[] facts = new int[80]; // размер массива 80 выбран произвольно
    int i, j;

    // Найти множители и поместить их в массив facts.
    for(i=2, j=0; i < num/2 + 1; i++)
      if( (num%i)==0 ) {
        facts[j] = i;
        j++;
      }

    numfactors = j;
    return facts;
  }
}
```

```
class FindFactors {
  static void Main() {
    Factor f = new Factor();
    int numfactors;
    int[] factors;

    factors = f.FindFactors(1000, out numfactors);

    Console.WriteLine("Множители числа 1000: ");
    for(int i=0; i < numfactors; i++)
      Console.Write(factors[i] + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Множители числа 1000:
2 4 5 8 10 20 25 40 50 100 125 200 250 500
```

В классе `Factor` метод `FindFactors()` объявляется следующим образом.

```
public int[] FindFactors(int num, out int numfactors) {
```

Обратите внимание на то, как указывается возвращаемый массив типа `int`. Этот синтаксис можно обобщить. Всякий раз, когда метод возвращает массив, он указывается аналогичным образом, но с учетом его типа и размерности. Например, в следующей строке кода объявляется метод `someMeth()`, возвращающий двумерный массив типа `double`.

```
public double[,] someMeth() { // ...
```

Перегрузка методов

В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы *перегружаются*, а сам процесс называется *перегрузкой методов*. Перегрузка методов относится к одному из способов реализации полиморфизма в C#.

В общем, для перегрузки метода достаточно объявить разные его варианты, а об остальном позаботится компилятор. Но при этом необходимо соблюсти следующее важное условие: тип или число параметров у каждого метода должны быть разными. Совершенно недостаточно, чтобы два метода отличались только типами возвращаемых значений. Они должны также отличаться типами или числом своих параметров. (Во всяком случае, типы возвращаемых значений дают недостаточно сведений компилятору C#, чтобы решить, какой именно метод следует использовать.) Разумеется, перегружаемые методы могут отличаться и типами возвращаемых значений. Когда вызывается перегружаемый метод, то выполняется тот его вариант, параметры которого соответствуют (по типу и числу) передаваемым аргументам.

Ниже приведен простой пример, демонстрирующий перегрузку методов.

```
// Продемонстрировать перегрузку методов.

using System;

class Overload {
  public void OvlDemo() {
    Console.WriteLine("Без параметров");
}

  // Перегрузка метода OvlDemo с одним целочисленным параметром.
  public void OvlDemo(int a) {
    Console.WriteLine("Один параметр: " + a);
  }

  // Перегрузка метода OvlDemo с двумя целочисленными параметрами.
  public int OvlDemo(int a, int b) {
    Console.WriteLine("Два параметра: " + a + " " + b);
    return a + b;
  }

  // Перегрузка метода OvlDemo с двумя параметрами типа double.
  public double OvlDemo(double a, double b) {
    Console.WriteLine("Два параметра типа double: " +
                      a + " "+ b);
    return a + b;
  }
}

class OverloadDemo {
  static void Main() {
    Overload ob = new Overload();
    int resI;
    double resD;

    // Вызвать все варианты метода OvlDemo().
    ob.OvlDemo();
    Console.WriteLine();

    ob.OvlDemo(2);
    Console.WriteLine();

    resI = ob.OvlDemo(4, 6);
    Console.WriteLine("Результат вызова метода ob.OvlDemo(4, 6): " + resI
    Console.WriteLine();

    resD = ob.OvlDemo(1.1, 2.32);
    Console.WriteLine("Результат вызова метода ob.OvlDemo(1.1, 2.32): " +
resD);
  }
}
```

Вот к какому результату приводит выполнение приведенного выше кода.

```
Без параметров

Один параметр: 2

Два параметра: 4 6
Результат вызова метода ob.OvlDemo(4, 6): 10

Два параметра типа double: 1.1 2.32
Результат вызова метода ob.OvlDemo(1.1, 2.32): 3.42
```

Как видите, метод `OvlDemo()` перегружается четыре раза. Первый его вариант не получает параметров, второй получает один целочисленный параметр, третий — два целочисленных параметра, а четвертый — два параметра типа `double`. Обратите также внимание на то, что два первых варианта метода `OvlDemo()` возвращают значение типа `void`, а по существу, не возвращают никакого значения, а два других — возвращают конкретное значение. И это совершенно допустимо, но, как пояснялось выше, тип возвращаемого значения не играет никакой роли для перегрузки метода. Следовательно, попытка использовать два разных (по типу возвращаемого значения) варианта метода `OvlDemo()` в приведенном ниже фрагменте кода приведет к ошибке.

```
// Одно объявление метода OvlDemo(int) вполне допустимо.
public void OvlDemo(int a) {
  Console.WriteLine("Один параметр: " + a);
}

/* Ошибка! Два объявления метода OvlDemo(int) не допускаются,
   хотя они и возвращают разнотипные значения. */
public int OvlDemo(int a) {
  Console.WriteLine("Один параметр: " + a);
  return a * a;
}
```

Как следует из комментариев к приведенному выше коду, отличий в типах значений, возвращаемых обоими вариантами метода `OvlDemo()`, оказывается недостаточно для перегрузки данного метода.

И как пояснялось в главе 3, в C# предусмотрен ряд неявных (т.е. автоматических) преобразований типов. Эти преобразования распространяются также на параметры перегружаемых методов. В качестве примера рассмотрим следующую программу.

```
// Неявные преобразования типов могут повлиять на
// решение перегружать метод.

using System;

class Overload2 {
  public void MyMeth(int x) {
    Console.WriteLine("В методе MyMeth(int): " + x);
  }

  public void MyMeth(double x) {
    Console.WriteLine("В методе MyMeth(double): " + x);
  }
}
```

```
class TypeConv {
  static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.MyMeth(i); // вызвать метод ob.MyMeth(int)
    ob.MyMeth(d); // вызвать метод ob.MyMeth(double)

    ob.MyMeth(b); // вызвать метод ob.MyMeth(int) -- с преобразованием типа
    ob.MyMeth(s); // вызвать метод ob.MyMeth(int) -- с преобразованием типа
    ob.MyMeth(f); // вызвать метод ob.MyMeth(double) -- с преобразованием типа
  }
}
```

При выполнении этой программы получается следующий результат.

```
В методе MyMeth(int): 10
В методе MyMeth(double): 10.1
В методе MyMeth(int): 99
В методе MyMeth(int): 10
В методе MyMeth(double): 11.5
```

В данном примере определены только два варианта метода `MyMeth()`: с параметром типа `int` и с параметром типа `double`. Тем не менее методу `MyMeth()` можно передать значение типа `byte`, `short` или `float`. Так, если этому методу передается значение типа `byte` или `short`, то компилятор C# автоматически преобразует это значение в тип `int` и в итоге вызывается вариант `MyMeth(int)` данного метода. А если ему передается значение типа `float`, то оно преобразуется в тип `double` и в результате вызывается вариант `MyMeth(double)` данного метода.

Следует, однако, иметь в виду, что неявные преобразования типов выполняются лишь в том случае, если отсутствует точное соответствие типов параметра и аргумента. В качестве примера ниже приведена чуть измененная версия предыдущей программы, в которую добавлен вариант метода `MyMeth()`, где указывается параметр типа `byte`.

```
// Добавить метод MyMeth(byte).

using System;

class Overload2 {
  public void MyMeth(byte x) {
    Console.WriteLine("В методе MyMeth(byte): " + x);
  }

  public void MyMeth(int x) {
    Console.WriteLine("В методе MyMeth(int): " + x);
  }

  public void MyMeth(double x) {
```

```
    Console.WriteLine("В методе MyMeth(double): " + x);
  }
}

class TypeConv {
  static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.MyMeth(i); // вызвать метод ob.MyMeth(int)
    ob.MyMeth(d); // вызвать метод ob.MyMeth(double)

    ob.MyMeth(b); // вызвать метод ob.MyMeth(byte) --
                  // на этот раз без преобразования типа

    ob.MyMeth(s); // вызвать метод ob.MyMeth(int) -- с преобразованием типа
    ob.MyMeth(f); // вызвать метод ob.MyMeth(double) -- с преобразованием типа
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
В методе MyMeth(int): 10
В методе MyMeth(double): 10.1
В методе MyMeth(byte): 99
В методе MyMeth(int): 10
В методе MyMeth(double): 11.5
```

В этой программе присутствует вариант метода `MyMeth()`, принимающий аргумент типа `byte`, поэтому при вызове данного метода с аргументом типа `byte` выбирается его вариант `MyMeth(byte)` без автоматического преобразования в тип `int`.

Оба модификатора параметров, `ref` и `out`, также учитываются, когда принимается решение о перегрузке метода. В качестве примера ниже приведен фрагмент кода, в котором определяются два совершенно разных метода.

```
public void MyMeth(int x) {
  Console.WriteLine("В методе MyMeth(int): " + x);
}

public void MyMeth(ref int x) {
  Console.WriteLine("В методе MyMeth(ref int): " + x);
}
```

Следовательно, при обращении

```
ob.MyMeth(i)
```

вызывается метод `MyMeth(int x)`, но при обращении

```
ob.MyMeth(ref i)
```

вызывается метод `MyMeth(ref int x)`.

Несмотря на то что модификаторы параметров `ref` и `out` учитываются, когда принимается решение о перегрузке метода, отличие между ними не столь существенно. Например, два следующих варианта метода `MyMeth()` оказываются недействительными.

```
// Неверно!
public void MyMeth(out int x) { // ...
public void MyMeth(ref int x) { // ...
```

В данном случае компилятор не в состоянии различить два варианта одного и того же метода `MyMeth()` только на основании того, что в одном из них используется параметр `out`, а в другом — параметр `ref`.

Перегрузка методов поддерживает свойство полиморфизма, поскольку именно таким способом в C# реализуется главный принцип полиморфизма: один интерфейс — множество методов. Для того чтобы стало понятнее, как это делается, обратимся к конкретному примеру. В языках программирования, не поддерживающих перегрузку методов, каждому методу должно быть присвоено уникальное имя. Но в программировании зачастую возникает потребность реализовать по сути один и тот же метод для обработки разных типов данных. Допустим, что требуется функция, определяющая абсолютное значение. В языках, не поддерживающих перегрузку методов, обычно приходится создавать три или более вариантов такой функции с несколько отличающимися, но все же разными именами. Например, в C функция `abs()` возвращает абсолютное значение целого числа, функция `labs()` — абсолютное значение длинного целого числа, а функция `fabs()` — абсолютное значение числа с плавающей точкой обычной (одинарной) точности.

В C перегрузка не поддерживается, и поэтому у каждой функции должно быть свое, особое имя, несмотря на то, что все упомянутые выше функции, по существу, делают одно и то же — определяют абсолютное значение. Но это принципиально усложняет положение, поскольку приходится помнить имена всех трех функций, хотя они реализованы по одному и тому же основному принципу. Подобные затруднения в C# не возникают, поскольку каждому методу, определяющему абсолютное значение, может быть присвоено одно и то же имя. И действительно, в состав библиотеки классов для среды .NET Framework входит метод `Abs()`, который перегружается в классе `System.Math` для обработки данных разных числовых типов. Компилятор C# сам определяет, какой именно вариант метода `Abs()` следует вызывать, исходя из типа передаваемого аргумента.

Главная ценность перегрузки заключается в том, что она обеспечивает доступ к связанным вместе методам по общему имени. Следовательно, имя `Abs` обозначает общее выполняемое действие, а компилятор сам выбирает конкретный вариант метода по обстоятельствам. Благодаря полиморфизму несколько имен сводятся к одному. Несмотря на всю простоту рассматриваемого здесь примера, продемонстрированный в нем принцип полиморфизма можно расширить, чтобы выяснить, каким образом перегрузка помогает справляться с намного более сложными ситуациями в программировании.

Когда метод перегружается, каждый его вариант может выполнять какое угодно действие. Для установления взаимосвязи между перегружаемыми методами не существует какого-то одного правила, но с точки зрения правильного стиля программирования перегрузка методов подразумевает подобную взаимосвязь. Следовательно, использовать одно и то же имя для несвязанных друг с другом методов не следует, хотя это и возможно. Например, имя `Sqr` можно было бы выбрать для методов, возвращающих квадрат и квадратный корень числа с плавающей точкой. Но ведь это

принципиально разные операции. Такое применение перегрузки методов противоречит ее первоначальному назначению. На практике перегружать следует только тесно связанные операции.

В C# определено понятие *сигнатуры*, обозначающее имя метода и список его параметров. Применительно к перегрузке это понятие означает, что в одном классе не должно существовать двух методов с одной и той же сигнатурой. Следует подчеркнуть, что в сигнатуру не входит тип возвращаемого значения, поскольку он не учитывается, когда компилятор C# принимает решение о перегрузке метода. В сигнатуру не входит также модификатор `params`.

Перегрузка конструкторов

Как и методы, конструкторы также могут перегружаться. Это дает возможность конструировать объекты самыми разными способами. В качестве примера рассмотрим следующую программу.

```
// Продемонстрировать перегрузку конструктора.

using System;

class MyClass {
  public int x;

  public MyClass() {
    Console.WriteLine("В конструкторе MyClass().");
    x = 0;
  }

  public MyClass(int i) {
    Console.WriteLine("В конструкторе MyClass(int).");
    x = i;
  }

  public MyClass(double d) {
    Console.WriteLine("В конструкторе MyClass(double).");
    x = (int) d;
  }

  public MyClass(int i, int j) {
    Console.WriteLine("В конструкторе MyClass(int, int).");
    x = i * j;
  }
}

class OverloadConsDemo {
  static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass(88);
    MyClass t3 = new MyClass(17.23);
    MyClass t4 = new MyClass(2, 4);

    Console.WriteLine("t1.x: " + t1.x);
```

```
    Console.WriteLine("t2.x: " + t2.x);
    Console.WriteLine("t3.x: " + t3.x);
    Console.WriteLine("t4.x: " + t4.x);
  }
}
```

При выполнении этой программы получается следующий результат.

```
В конструкторе MyClass().
В конструкторе MyClass(int).
В конструкторе MyClass(double).
В конструкторе MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

В данном примере конструктор `MyClass()` перегружается четыре раза, всякий раз конструируя объект по-разному. Подходящий конструктор вызывается каждый раз, исходя из аргументов, указываемых при выполнении оператора `new`. Перегрузка конструктора класса предоставляет пользователю этого класса дополнительные преимущества в конструировании объектов.

Одна из самых распространенных причин для перегрузки конструкторов заключается в необходимости предоставить возможность одним объектам инициализировать другие. В качестве примера ниже приведен усовершенствованный вариант разработанного ранее класса `Stack`, позволяющий конструировать один стек из другого.

```
// Класс для хранения символов в стеке.

using System;

class Stack {
  // Эти члены класса являются закрытыми.
  char[] stck; // массив, содержащий стек
  int tos;     // индекс вершины стека

  // Сконструировать пустой объект класса Stack по заданному размеру стека.
  public Stack(int size) {
    stck = new char[size]; // распределить память для стека
    tos = 0;
  }

  // Сконструировать объект класса Stack из существующего стека.
  public Stack(Stack ob) {
    // Распределить память для стека.
    stck = new char[ob.stck.Length];

    // Скопировать элементы в новый стек.
    for(int i=0; i < ob.tos; i++)
      stck[i] = ob.stck[i];

    // Установить переменную tos для нового стека.
    tos = ob.tos;
  }
```

```
  // Поместить символы в стек.
  public void Push(char ch) {
    if(tos==stck.Length) {
      Console.WriteLine(" - Стек заполнен.");
      return;
    }

    stck[tos] = ch;
    tos++;
  }

  // Извлечь символ из стека.
  public char Pop() {
    if(tos==0) {
      Console.WriteLine(" - Стек пуст.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }

  // Возвратить значение true, если стек заполнен.
  public bool IsFull() {
    return tos==stck.Length;
  }

  // Возвратить значение true, если стек пуст.
  public bool IsEmpty() {
    return tos==0;
  }

  // Возвратить общую емкость стека.
  public int Capacity() {
    return stck.Length;
  }

  // Возвратить количество объектов, находящихся в настоящий момент в стеке.
  public int GetNum() {
    return tos;
  }
}

// Продемонстрировать применение класса Stack.

class StackDemo {
  static void Main() {
    Stack stk1 = new Stack(10);
    char ch;
    int i;

    // Поместить ряд символов в стек stk1.
    Console.WriteLine("Поместить символы A-J в стек stk1.");
    for(i=0; !stk1.IsFull(); i++)
```

```
      stk1.Push((char) ('A' + i));

    // Создать копию стека stck1.
    Stack stk2 = new Stack(stk1);

    // Вывести содержимое стека stk1.
    Console.Write("Содержимое стека stk1: ");
    while( !stk1.IsEmpty() ) {
      ch = stk1.Pop();
      Console.Write(ch);
    }

    Console.WriteLine();

    Console.Write("Содержимое стека stk2: ");
    while ( !stk2.IsEmpty() ) {
      ch = stk2.Pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

  }
}
```

Результат выполнения этой программы приведен ниже.

```
Поместить символы A-J в стек stk1.
Содержимое стека stk1: JIHGFEDCBA
Содержимое стека stk2: JIHGFEDCBA
```

В классе `StackDemo` сначала конструируется первый стек (`stk1`), заполняемый символами. Затем этот стек используется для конструирования второго стека (`stk2`). Это приводит к выполнению следующего конструктора класса `Stack`.

```
// Сконструировать объект класса Stack из существующего стека.
public Stack(Stack ob) {
  // Распределить память для стека.
  stck = new char[ob.stck.Length];

  // Скопировать элементы в новый стек.
  for(int i=0; i < ob.tos; i++)
    stck[i] = ob.stck[i];

  // Установить переменную tos для нового стека.
  tos = ob.tos;
}
```

В этом конструкторе сначала распределяется достаточный объем памяти для массива, чтобы хранить в нем элементы стека, передаваемого в качестве аргумента `ob`. Затем содержимое массива, образующего стек `ob`, копируется в новый массив, после чего соответственно устанавливается переменная `tos`, содержащая индекс вершины стека. По завершении работы конструктора новый и исходный стеки существуют как отдельные, хотя и одинаковые объекты.

Вызов перегружаемого конструктора с помощью ключевого слова this

Когда приходится работать с перегружаемыми конструкторами, то иногда очень полезно предоставить возможность одному конструктору вызывать другой. В C# это дается с помощью ключевого слова this. Ниже приведена общая форма такого вызова.

```
имя_конструктора(список_параметров1) : this(список_параметров2) {
  // ... Тело конструктора, которое может быть пустым.
}
```

В исходном конструкторе сначала выполняется перегружаемый конструктор, список параметров которого соответствует критерию *список_параметров2*, а затем все остальные операторы, если таковые имеются в исходном конструкторе. Ниже приведен соответствующий пример.

```
// Продемонстрировать вызов конструктора с помощью ключевого слова this.

using System;

class XYCoord {
  public int x, y;

  public XYCoord() : this(0, 0) {
    Console.WriteLine("В конструкторе XYCoord()");
  }

  public XYCoord(XYCoord obj) : this(obj.x, obj.y) {
    Console.WriteLine("В конструкторе XYCoord(obj)");
  }

  public XYCoord(int i, int j) {
    Console.WriteLine("В конструкторе XYCoord(int, int)");
    x = i;
    y = j;
  }
}

class OverloadConsDemo {
  static void Main() {
    XYCoord t1 = new XYCoord();
    XYCoord t2 = new XYCoord(8, 9);
    XYCoord t3 = new XYCoord(t2);

    Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);
    Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);
    Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);
  }
}
```

Выполнение этого кода приводит к следующему результату.

```
В конструкторе XYCoord(int, int)
В конструкторе XYCoord()
В конструкторе XYCoord(int, int)
В конструкторе XYCoord(int, int)
```

```
В конструкторе XYCoord(obj)
t1.x, t1.y: 0, 0
t2.x, t2.y: 8, 9
t3.x, t3.y: 8, 9
```

Код в приведенном выше примере работает следующим образом. Единственным конструктором, фактически инициализирующим поля `x` и `y` в классе `XYCoord`, является конструктор `XYCoord(int, int)`. А два других конструктора просто вызывают этот конструктор с помощью ключевого слова `this`. Например, когда создается объект `t1`, то вызывается его конструктор `XYCoord()`, что приводит к вызову `this(0, 0)`, который в данном случае преобразуется в вызов конструктора `XYCoord(0, 0)`. То же самое происходит и при создании объекта `t2`.

Вызывать перегружаемый конструктор с помощью ключевого слова `this` полезно, в частности, потому, что он позволяет исключить ненужное дублирование кода. В приведенном выше примере нет никакой необходимости дублировать во всех трех конструкторах одну и ту же последовательность инициализации, и благодаря применению ключевого слова `this` такое дублирование исключается. Другое преимущество организации подобного вызова перезагружаемого конструктора заключается в возможности создавать конструкторы с задаваемыми "по умолчанию" аргументами, когда эти аргументы не указаны явно. Ниже приведен пример создания еще одного конструктора `XYCoord`.

```
public XYCoord(int x) : this(x, x) { }
```

По умолчанию в этом конструкторе для координаты `y` автоматически устанавливается то же значение, что и для координаты `y`. Конечно, пользоваться такими конструкциями с задаваемыми "по умолчанию" аргументами следует благоразумно и осторожно, чтобы не ввести в заблуждение пользователей классов.

Инициализаторы объектов

Инициализаторы объектов предоставляют еще один способ создания объекта и инициализации его полей и свойств. (Подробнее о свойствах речь пойдет в главе 10.) Если используются инициализаторы объектов, то вместо обычного вызова конструктора класса указываются имена полей или свойств, инициализируемых первоначально задаваемым значением. Следовательно, синтаксис инициализатора объекта предоставляет альтернативу явному вызову конструктора класса. Синтаксис инициализатора объекта используется главным образом при создании анонимных типов в LINQ-выражениях. (Подробнее об анонимных типах и LINQ-выражениях — в главе 19.) Но поскольку инициализаторы объектов можно, а иногда и должно использовать в именованном классе, то ниже представлены основные положения об инициализации объектов.

Обратимся сначала к простому примеру.

```
// Простой пример, демонстрирующий применение инициализаторов объектов.

using System;

class MyClass {
  public int Count;
  public string Str;
}
```

```
class ObjInitDemo {
  static void Main() {
    // Сконструировать объект типа MyClass, используя инициализаторы
объектов.
    MyClass obj = new MyClass { Count = 100, Str = "Тестирование" };

    Console.WriteLine(obj.Count + " " + obj.Str);
  }
}
```

Выполнение этого кода дает следующий результат.

```
100 Тестирование
```

Как показывает результат выполнения приведенного выше кода, переменная экземпляра `obj.Count` инициализирована значением `100`, а переменная экземпляра `obj.Str` — символьной строкой `"Тестирование"`. Но обратите внимание на то, что в классе `MyClass` отсутствуют явно определяемые конструкторы и не используется обычный синтаксис конструкторов. Вместо этого объект `obj` класса `MyClass` создается с помощью следующей строки кода.

```
MyClass obj = new MyClass { Count = 100, Str = "Тестирование" };
```

В этой строке кода имена полей указываются явно вместе с их первоначальными значениями. Это приводит к тому, что сначала конструируется экземпляр объекта типа `MyClass` (с помощью неявно вызываемого по умолчанию конструктора), а затем задаются первоначальные значения переменных `Count` и `Str` данного экземпляра.

Следует особо подчеркнуть, что порядок указания инициализаторов особого значения не имеет. Например, объект `obj` можно было бы инициализировать и так, как показано ниже.

```
MyClass obj = new MyClass { Str = "Тестирование", Count = 100 };
```

В этой строке кода инициализация переменной экземпляра `Str` предшествует инициализации переменной экземпляра `Count`, а в приведенном выше коде все происходило наоборот. Но в любом случае результат получается одинаковым.

Ниже приведена общая форма синтаксиса инициализации объектов:

```
new имя_класса {имя = выражение, имя = выражение, ...}
```

где *имя* обозначает имя поля или свойства, т.е. доступного члена класса, на который указывает *имя_класса*. А *выражение* обозначает инициализирующее выражение, тип которого, конечно, должен соответствовать типу поля или свойства.

Инициализаторы объектов обычно не используются в именованных классах, как, например, в представленном выше классе `MyClass`, хотя это вполне допустимо. Вообще, при обращении с именованными классами используется синтаксис вызова обычного конструктора. И, как упоминалось выше, инициализаторы объектов применяются в основном в анонимных типах, формируемых в LINQ-выражениях.

Необязательные аргументы

В версии C# 4.0 внедрено новое средство, повышающее удобство указания аргументов при вызове метода. Это средство называется *необязательными аргументами* и позволяет определить используемое по умолчанию значение для параметра метода.

Данное значение будет использоваться по умолчанию в том случае, если для параметра не указан соответствующий аргумент при вызове метода. Следовательно, указывать аргумент для такого параметра не обязательно. Необязательные аргументы позволяют упростить вызов методов, где к некоторым параметрам применяются аргументы, выбираемые по умолчанию. Их можно также использовать в качестве "сокращенной" формы перегрузки методов.

Применение необязательного аргумента разрешается при создании *необязательного параметра*. Для этого достаточно указать используемое по умолчанию значение параметра с помощью синтаксиса, аналогичного инициализации переменной. Используемое по умолчанию значение должно быть константным выражением. В качестве примера рассмотрим следующее определение метода.

```
static void OptArgMeth(int alpha, int beta=10, int gamma = 20) {
```

В этой строке кода объявляются два необязательных параметра: `beta` и `gamma`, причем параметру `beta` по умолчанию присваивается значение 10, а параметру `gamma` — значение 20. Эти значения используются по умолчанию, если для данных параметров не указываются аргументы при вызове метода. Следует также иметь в виду, что параметр `alpha` не является необязательным. Напротив, это обычный параметр, для которого всегда нужно указывать аргумент.

Принимая во внимание приведенное выше объявление метода `OptArgMeth()`, последний можно вызвать следующими способами.

```
// Передать все аргументы явным образом.
OptArgMeth(1, 2, 3);

// Сделать аргумент gamma необязательным.
OptArgMeth(1, 2);
// Сделать оба аргумента beta и gamma необязательными.
OptArgMeth(1);
```

При первом вызове параметру `alpha` передается значение 1, параметру `beta` — значение 2, а параметру `gamma` — значение 3. Таким образом, все три аргумента задаются явным образом, а значения, устанавливаемые по умолчанию, не используются. При втором вызове параметру `alpha` передается значение 1, а параметру `beta` — значение 2, но параметру `gamma` присваивается устанавливаемое по умолчанию значение 20. И наконец, при третьем вызове упомянутого выше метода параметру `alpha` передается значение 1, а параметрам `beta` и `gamma` присваиваются устанавливаемые по умолчанию значения. Следует, однако, иметь в виду, что параметр `beta` не получит устанавливаемое по умолчанию значение, если то же самое не произойдет с параметром `gamma`. Если первый аргумент устанавливается по умолчанию, то и все остальные аргументы должны быть установлены по умолчанию.

Весь описанный выше процесс демонстрируется в приведенном ниже примере программы.

```
// Продемонстрировать необязательные аргументы.

using System;

class OptionArgDemo {
  static void OptArgMeth(int alpha, int beta=10, int gamma = 20) {
    Console.WriteLine("Это аргументы alpha, beta и gamma: " +
                      alpha + " " + beta + " " + gamma);
```

```
  }

  static void Main() {
    // Передать все аргументы явным образом.
    OptArgMeth(1, 2, 3);

    // Сделать аргумент gamma необязательным.
    OptArgMeth(1, 2);

    // Сделать оба аргумента beta и gamma необязательными.
    OptArgMeth(1);
  }
}
```

Результат выполнения данной программы лишь подтверждает применение используемых по умолчанию аргументов.

```
Это аргументы alpha, beta и gamma: 1 2 3
Это аргументы alpha, beta и gamma: 1 2 20
Это аргументы alpha, beta и gamma: 1 10 20
```

Как следует из приведенного выше результата, если аргумент не указан, то используется его значение, устанавливаемое по умолчанию.

Следует иметь в виду, что все необязательные аргументы должны непременно указываться *справа* от обязательных. Например, следующее объявление оказывается недействительным.

```
int Sample(string name = "пользователь", int userId) { // Ошибка!
```

Для исправления ошибки в этом объявлении необходимо указать аргумент `userId` до аргумента `name`. Раз уж вы начали объявлять необязательные аргументы, то указывать после них обязательные аргументы нельзя. Например, следующее объявление также оказывается неверным.

```
int Sample(int accountId, string name = "пользователь", int userId) { // Ошибка!
```

Аргумент `name` объявляется как необязательный, и поэтому аргумент `userId` следует указать до аргумента `name` (или же сделать его также необязательным).

Помимо методов, необязательные аргументы можно применять в конструкторах, индексаторах и делегатах. (Об индексаторах и делегатах речь пойдет далее в этой книге.)

Преимущество необязательных аргументов заключается, в частности, в том, что они упрощают программирующему обращение со сложными вызовами методов и конструкторов. Ведь нередко в методе приходится задавать больше параметров, чем обычно требуется. И в подобных случаях некоторые из этих параметров могут быть сделаны необязательными благодаря аккуратному применению необязательных аргументов. Это означает, что передавать нужно лишь те аргументы, которые важны в данном конкретном случае, а не все аргументы, которые в противном случае должны быть обязательными. Такой подход позволяет рационализировать метод и упростить программирующему обращение с ним.

Необязательные аргументы и перегрузка методов

В некоторых случаях необязательные аргументы могут стать альтернативой перегрузке методов. Для того чтобы стало понятнее, почему это возможно, обратимся еще

раз к примеру метода `OptArgMeth()`. До появления в C# необязательных аргументов нам пришлось бы создать три разных варианта метода `OptArgMeth()`, чтобы добиться таких же функциональных возможностей, как и у рассмотренного выше варианта этого метода. Все эти варианты пришлось бы объявить следующим образом.

```
static void OptArgMeth(int alpha)
static void OptArgMeth(int alpha, int beta)
static void OptArgMeth(int alpha, int beta, int gamma)
```

Эти перегружаемые варианты метода `OptArgMeth()` позволяют вызывать его с одним, двумя или тремя аргументами. (Если значения параметров `beta` и `gamma` не передаются, то они предоставляются в теле перегружаемых вариантов данного метода.) Безусловно, в такой реализации функциональных возможностей метода `OptArgMeth()` с помощью перегрузки нет ничего дурного. Но в данном случае целесообразнее все же воспользоваться необязательными аргументами, хотя такой подход не всегда оказывается более совершенным, чем перегрузка метода.

Необязательные аргументы и неоднозначность

При использовании необязательных аргументов может возникнуть такое затруднение, как неоднозначность. Нечто подобное может произойти при перегрузке метода с необязательными параметрами. В некоторых случаях компилятор может оказаться не в состоянии определить, какой именно вариант метода следует вызывать, когда необязательные аргументы не заданы. В качестве примера рассмотрим два следующих варианта метода `OptArgMeth()`.

```
static void OptArgMeth(int alpha, int beta=10, int gamma = 20) {
  Console.WriteLine("Это аргументы alpha, beta и gamma: " +
                    alpha + " " + beta + " " + gamma);
}

static void OptArgMeth(int alpha, double beta=10.0, double gamma = 20.0) {
  Console.WriteLine("Это аргументы alpha, beta и gamma: " +
                    alpha + " " + beta + " " + gamma);
}
```

Обратите внимание на то, что единственное отличие в обоих вариантах рассматриваемого здесь метода состоит в типах параметров `beta` и `gamma`, которые оказываются необязательными. В первом варианте оба параметра относятся к типу `int`, а во втором — к типу `double`. С учетом этих вариантов перегрузки метода `OptArgMeth()` следующий его вызов приводит к неоднозначности.

```
OptArgMeth(1); // Ошибка из-за неоднозначности!
```

Этот вызов приводит к неоднозначности потому, что компилятору неизвестно, какой именно вариант данного метода использовать: тот, где параметры `beta` и `gamma` имеют тип `int`, или же тот, где они имеют тип `double`. Но самое главное, что конкретный вызов метода `OptArgMeth()` может привести к неоднозначности, даже если она и не присуща его перегрузке.

В связи с тем что перегрузка методов, допускающих применение необязательных аргументов, может привести к неоднозначности, очень важно принимать во внимание последствия такой перегрузки. В некоторых случаях, возможно, придется отказаться от применения необязательных аргументов, чтобы исключить неоднозначность и тем самым предотвратить использование метода непреднамеренным образом.

Практический пример использования необязательных аргументов

Для того чтобы показать на практике, насколько необязательные аргументы упрощают вызовы некоторых типов методов, рассмотрим следующий пример программы. В этой программе объявляется метод `Display()`, выводящий на экран символьную строку полностью или частично.

```
// Использовать необязательный аргумент, чтобы упростить вызов метода.

using System;

class UseOptArgs {

  // Вывести на экран символьную строку полностью или частично.
  static void Display(string str, int start = 0, int stop = -1) {

    if(stop < 0)
      stop = str.Length;

    // Проверить условие выхода за заданные пределы.
    if(stop > str.Length | start > stop | start < 0)
      return;

    for(int i=start; i < stop; i++)
      Console.Write(str[i]);

    Console.WriteLine();
  }

  static void Main() {
     Display("это простой тест");
     Display("это простой тест", 12);
    Display("это простой тест", 4, 14);
  }
}
```

Выполнение этой программы дает следующий результат.

```
это простой тест
тест
простой те
```

Внимательно проанализируем метод `Display()`. Выводимая на экран символьная строка передается в первом аргументе данного метода. Это обязательный аргумент, а два других аргумента — необязательные. Они задают начальный и конечный индексы для вывода части символьной строки. Если параметру `stop` не передается значение, то по умолчанию он принимает значение –1, указывающее на то, что конечной точкой вывода служит конец символьной строки. Если же параметру `start` не передается значение, то по умолчанию он принимает значение 0. Следовательно, в отсутствие одного из необязательных аргументов символьная строка выводится на экран полностью. В противном случае она выводится на экран частично. Это означает, что если вызвать метод `Display()` с одним аргументом (т.е. с выводимой строкой), то символьная строка будет выведена на экран полностью. Если же вызвать метод `Display()` с двумя аргументами, то на экран будут выведены символы, начиная с позиции, определяемой аргументом

start, и до самого конца строки. А если вызвать метод Display() с тремя аргументами, то на экран будут выведены символы из строки, начиная с позиции, определяемой аргументом start, и заканчивая позицией, определяемой аргументом stop.

Несмотря на всю простоту данного примера, он, тем не менее, демонстрирует значительное преимущество, которое дают необязательные аргументы. Это преимущество заключается в том, что при вызове метода можно указывать только те аргументы, которые требуются. А передавать явным образом устанавливаемые по умолчанию значения не нужно.

Прежде чем переходить к следующей теме, остановимся на следующем важном моменте. Необязательные аргументы оказываются весьма эффективным средством лишь в том случае, если они используются правильно. Они предназначены для того, чтобы метод выполнял свои функции эффективно, а пользоваться им можно было бы просто и удобно. В этом отношении устанавливаемые по умолчанию значения всех аргументов должны упрощать обычное применение метода. В противном случае необязательные аргументы способны нарушить структуру кода и ввести в заблуждение тех, кто им пользуется. И наконец, устанавливаемое по умолчанию значение необязательного параметра не должно наносить никакого вреда. Иными словами, неумышленное использование необязательного аргумента не должно приводить к необратимым, отрицательным последствиям. Так, если забыть указать аргумент при вызове метода, то это не должно привести к удалению важного файла данных!

Именованные аргументы

Еще одним средством, связанным с передачей аргументов методу, является *именованный аргумент*. Именованные аргументы были внедрены в версии C# 4.0. Как вам должно быть уже известно, при передаче аргументов методу порядок их следования, как правило, должен совпадать с тем порядком, в котором параметры определены в самом методе. Иными словами, значение аргумента присваивается параметру по его позиции в списке аргументов. Данное ограничение призваны преодолеть именованные аргументы. Именованный аргумент позволяет указать имя того параметра, которому присваивается его значение. И в этом случае порядок следования аргументов уже не имеет никакого значения. Таким образом, именованные аргументы в какой-то степени похожи на упоминавшиеся ранее инициализаторы объектов, хотя и отличаются от них своим синтаксисом.

Для указания аргумента по имени служит следующая форма синтаксиса.

имя_параметра : *значение*

Здесь *имя_параметра* обозначает имя того параметра, которому передается *значение*. Разумеется, *имя_параметра* должно обозначать имя действительного параметра для вызываемого метода.

Ниже приведен простой пример, демонстрирующий применение именованных аргументов. В этом примере создается метод IsFactor(), возвращающий логическое значение true, если первый его параметр нацело делится на второй параметр.

```
// Применить именованные аргументы.

using System;

class NamedArgsDemo {
```

```
  // Выяснить, делится ли одно значение нацело на другое.
  static bool IsFactor(int val, int divisor) {
    if((val % divisor) == 0) return true;
    return false;
  }

  static void Main() {
    // Ниже демонстрируются разные способы вызова метода IsFactor().

    // Вызов с использованием позиционных аргументов.
    if(IsFactor(10, 2))
      Console.WriteLine("2 - множитель 10.");

    // Вызов с использованием именованных аргументов.
    if(IsFactor(val: 10, divisor: 2))
      Console.WriteLine("2 - множитель 10.");

    // Для именованного аргумента порядок указания не имеет значения.
    if(IsFactor(divisor: 2, val: 10))
      Console.WriteLine("2 - множитель 10.");

    // Применить как позиционный, так и именованный аргумент.
    if(IsFactor(10, divisor: 2))
      Console.WriteLine("2 - множитель 10.");
  }
}
```

Выполнение этого кода дает следующий результат.

```
2 - множитель 10.
2 - множитель 10.
2 - множитель 10.
2 - множитель 10.
```

Как видите, при каждом вызове метода `IsFactor()` получается один и тот же результат.

Помимо демонстрации именованного аргумента в действии, приведенный выше пример кода иллюстрирует две важные особенности именованных аргументов. Во-первых, порядок следования аргументов не имеет никакого значения. Например, два следующих вызова метода `IsFactor()` совершенно равнозначны.

```
IsFactor(val :10, divisor: 2)
IsFactor(divisor: 2, val: 10)
```

Независимость от порядка следования является главным преимуществом именованных аргументов. Это означает, что запоминать (или даже знать) порядок следования параметров в вызываемом методе совсем не обязательно. Для работы с COM-интерфейсами это может быть очень удобно. И во-вторых, позиционные аргументы можно указывать вместе с именованными в одном и том же вызове, как показано в следующем примере.

```
IsFactor(10, divisor: 2)
```

Следует, однако, иметь в виду, что при совместном использовании именованных и позиционных аргументов все позиционные аргументы должны быть указаны перед любыми именованными аргументами.

Именованные аргументы можно также применять вместе с необязательными аргументами. Покажем это на примере вызова метода `Display()`, рассматривавшегося в предыдущем разделе.

```
// Указать все аргументы по имени.
Display(stop: 10, str: "это простой тест", start: 0);

// Сделать аргумент start устанавливаемым по умолчанию.
Display(stop: 10, str: "это простой тест");

// Указать строку по позиции, аргумент stop – по имени by name,
// тогда как аргумент start – устанавливаемым по умолчанию
Display("это простой тест", stop: 10);
```

Вообще говоря, комбинация именованных и необязательных аргументов позволяет упростить вызовы сложных методов со многими параметрами.

Синтаксис именованных аргументов более многословен, чем у обычных позиционных аргументов, и поэтому для вызова методов чаще всего применяются позиционные аргументы. Но в тех случаях, когда это уместно, именованные аргументы могут быть использованы довольно эффективно.

ПРИМЕЧАНИЕ

Помимо методов, именованные и необязательные аргументы могут применяться в конструкторах, индексаторах и делегатах. (Об индексаторах и делегатах речь пойдет далее в этой книге.)

Метод `Main()`

В представленных до сих пор примерах программ использовалась одна форма метода `Main()`. Но у него имеется также целый ряд перегружаемых форм. Одни из них могут служить для возврата значений, другие — для получения аргументов. В этом разделе рассматриваются и те и другие формы.

Возврат значений из метода `Main()`

По завершении программы имеется возможность возвратить конкретное значение из метода `Main()` вызывающему процессу (зачастую операционной системе). Для этой цели служит следующая форма метода `Main()`.

```
static int Main()
```

Обратите внимание на то, что в этой форме метода `Main()` объявляется возвращаемый тип `int` вместо типа `void`.

Как правило, значение, возвращаемое методом `Main()`, указывает на нормальное завершение программы или на аварийное ее завершение из-за сложившихся ненормальных условий выполнения. Условно нулевое возвращаемое значение обычно указывает на нормальное завершение программы, а все остальные значения обозначают тип возникшей ошибки.

Передача аргументов методу Main()

Многие программы принимают так называемые *аргументы командной строки*, т.е. информацию, которая указывается в командной строке непосредственно после имени программы при ее запуске на выполнение. В программах на C# такие аргументы передаются затем методу Main(). Для получения аргументов служит одна из приведенных ниже форм метода Main().

```
static void Main(string[ ] args)
static int Main(string[ ] args)
```

В первой форме метод Main() возвращает значение типа void, а во второй — целое значение, как пояснялось выше. Но в обеих формах аргументы командной строки сохраняются в виде символьных строк в массиве типа string, который передается методу Main(). Длина этого массива (args) должна быть равна числу аргументов командной строки, которое может быть и нулевым.

В качестве примера ниже приведена программа, выводящая все аргументы командной строки, вместе с которыми она вызывается.

```
// Вывести все аргументы командной строки.

using System;

class CLDemo {
  static void Main(string[] args) {
    Console.WriteLine("Командная строка содержит " +
                      args.Length +
                      " аргумента.");

    Console.WriteLine("Вот они: ");
      for(int i=0; i < args.Length; i++)
         Console.WriteLine(args[i]);
  }
}
```

Если программа CLDemo запускается из командной строки следующим образом:

```
CLDemo один два три
```

то ее выполнение дает такой результат.

```
Командная строка содержит 3 аргумента.
Вот они:
один
два
три
```

Для того чтобы стало понятнее, каким образом используются аргументы командной строки, рассмотрим еще один пример программы, в которой применяется простой подстановочный шифр для шифровки или расшифровки сообщений. Шифруемое или расшифровываемое сообщение указывается в командной строке. Применяемый шифр действует довольно просто. Для шифровки слова значение каждой его буквы инкрементируется на 1. Следовательно, Буква "А" становится буквой "Б" и т.д. А для расшифровки слова значение каждой его буквы декрементируется на 1. Разумеется, такой шифр не имеет никакой практической ценности, поскольку его нетрудно разгадать. Тем не менее он может стать приятным развлечением для детей.

```
// Зашифровать и расшифровать сообщение, используя
// простой подстановочный шифр.

using System;

class Cipher {
  static int Main(string[] args) {

    // Проверить наличие аргументов.
    if(args.Length < 2) {
      Console.WriteLine("ПРИМЕНЕНИЕ: " +
                        "слово1: <зашифровать>/<расшифровать> " +
                        "[слово2... словоN]");
      return 1; // возвратить код неудачного завершения программы
    }

    // Если аргументы присутствуют, то первым аргументом должно быть
    // слово <зашифровать> или же слово <расшифровать>.
    if(args[0] != "зашифровать" & args[0] != "расшифровать") {
      Console.WriteLine("Первым аргументом должно быть слово " +
                        "<зашифровать> или <расшифровать>.");
      return 1; // возвратить код неудачного завершения программы
    }

    // Зашифровать или расшифровать сообщение.
    for(int n=1; n < args.Length; n++) {
      for(int i=0; i < args[n].Length; i++) {
        if(args[0] == "зашифровать")
          Console.Write((char) (args[n][i] + 1) );
        else
          Console.Write((char) (args[n][i] - 1) );
      }
      Console.Write(" ");
    }

    Console.WriteLine();

    return 0;
  }
}
```

Для того чтобы воспользоваться этой программой, укажите в командной строке имя программы, затем командное слово "зашифровать" или "расшифровать" и далее сообщение, которое требуется зашифровать или расшифровать. Ниже приведены два примера выполнения данной программы, при условии, что она называется `Cipher`.

```
C:\Cipher зашифровать один два
пейо егб

C:\Cipher расшифровать пейо егб
один два
```

Данная программа отличается двумя интересными свойствами. Во-первых, обратите внимание на то, как в ней проверяется наличие аргументов командной строки перед тем, как продолжить выполнение. Это очень важное свойство, которое можно

обобщить. Если в программе принимается во внимание наличие одного или более аргументов командной строки, то в ней должна быть непременно организована проверка факта передачи ей предполагаемых аргументов, иначе программа будет работать неправильно. Кроме того, в программе должна быть организована проверка самих аргументов перед тем, как продолжить выполнение. Так, в рассматриваемой здесь программе проверяется наличие командного слова "зашифровать" или "расшифровать" в качестве первого аргумента командной строки.

И во-вторых, обратите внимание на то, как программа возвращает код своего завершения. Если предполагаемые аргументы командной строки отсутствуют или указаны неправильно, программа возвращает код 1, указывающий на ее аварийное завершение. В противном случае возвращается код 0, когда программа завершается нормально.

Рекурсия

В C# допускается, чтобы метод вызывал самого себя. Этот процесс называется *рекурсией*, а метод, вызывающий самого себя, — *рекурсивным*. Вообще, рекурсия представляет собой процесс, в ходе которого нечто определяет самое себя. В этом отношении она чем-то напоминает циклическое определение. Рекурсивный метод отличается главным образом тем, что он содержит оператор, в котором этот метод вызывает самого себя. Рекурсия является эффективным механизмом управления программой.

Классическим примером рекурсии служит вычисление факториала числа. Факториал числа *N* представляет собой произведение всех целых чисел от 1 до *N*. Например, факториал числа 3 равен 1×2×3, или 6. В приведенном ниже примере программы демонстрируется рекурсивный способ вычисления факториала числа. Для сравнения в эту программу включен также нерекурсивный вариант вычисления факториала числа.

```
// Простой пример рекурсии.

using System;

class Factorial {

  // Это рекурсивный метод.
  public int FactR(int n) {
    int result;

    if(n==1) return 1;
    result = FactR(n-1) * n;
    return result;
  }

  // Это итерационный метод.
  public int FactI(int n) {
    int t, result;

    result = 1;
    for(t=1; t <= n; t++) result *= t;
    return result;
  }
}
```

```
class Recursion {
  static void Main() {
    Factorial f = new Factorial();
    Console.WriteLine("Факториалы, рассчитанные рекурсивным методом.");
    Console.WriteLine("Факториал числа 3 равен " + f.FactR(3));
    Console.WriteLine("Факториал числа 4 равен " + f.FactR(4));
    Console.WriteLine("Факториал числа 5 равен " + f.FactR(5));
    Console.WriteLine();

    Console.WriteLine("Факториалы, рассчитанные итерационным методом.");
    Console.WriteLine("Факториал числа 3 равен " + f.FactR(3));
    Console.WriteLine("Факториал числа 4 равен " + f.FactR(4));
    Console.WriteLine("Факториал числа 5 равен " + f.FactR(5));
  }
}
```

При выполнении этой программы получается следующий результат.

```
Факториалы, рассчитанные рекурсивным методом.
Факториал числа 3 равен 6
Факториал числа 4 равен 24
Факториал числа 5 равен 120

Факториалы, рассчитанные итерационным методом.
Факториал числа 3 равен 6
Факториал числа 4 равен 24
Факториал числа 5 равен 120
```

Принцип действия нерекурсивного метода `FactI()` вполне очевиден. В нем используется цикл, в котором числа, начиная с 1, последовательно умножаются друг на друга, постепенно образуя произведение, дающее факториал.

А рекурсивный метод `FactR()` действует по более сложному принципу. Если метод `FactR()` вызывается с аргументом `1`, то он возвращает значение 1. В противном случае он возвращает произведение `FactR(n-1)*n`. Для вычисления этого произведения метод `FactR()` вызывается с аргументом `n-1`. Этот процесс повторяется до тех пор, пока значение аргумента `n` не станет равным 1, после чего из предыдущих вызовов данного метода начнут возвращаться полученные значения. Например, когда вычисляется факториал числа 2, то при первом вызове метода `FactR()` происходит второй его вызов с аргументом `1`. Из этого вызова возвращается значение 1, которое затем умножается на 2 (первоначальное значение аргумента `n`). В итоге возвращается результат 2, равный факториалу числа 2 (1×2). Было бы любопытно ввести в метод `FactR()` операторы, содержащие вызовы метода `WriteLine()`, чтобы наглядно показать уровень рекурсии при каждом вызове метода `FactR()`, а также вывести промежуточные результаты вычисления факториала заданного числа.

Когда метод вызывает самого себя, в системном стеке распределяется память для новых локальных переменных и параметров, и код метода выполняется с этими новыми переменными и параметрами с самого начала. При рекурсивном вызове метода не создается его новая копия, а лишь используются его новые аргументы. А при возврате из каждого рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и выполнение возобновляется с точки вызова в методе. Рекурсивные методы можно сравнить по принципу действия с постепенно сжимающейся и затем распрямляющейся пружиной.

Ниже приведен еще один пример рекурсии для вывода символьной строки в обратном порядке. Эта строка задается в качестве аргумента рекурсивного метода DisplayRev().

```
// Вывести символьную строку в обратном порядке, используя рекурсию.

using System;

class RevStr {

  // Вывести символьную строку в обратном порядке.
  public void DisplayRev(string str) {
    if(str.Length > 0)
      DisplayRev(str.Substring(1, str.Length-1));
    else
      return;

    Console.Write(str[0]);
  }
}

class RevStrDemo {
  static void Main() {
    string s = "Это тест";
    RevStr rsOb = new RevStr();

    Console.WriteLine("Исходная строка: " + s);

    Console.Write("Перевернутая строка: ");
    rsOb.DisplayRev(s);

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Исходная строка: Это тест
Перевернутая строка: тсет отЭ
```

Всякий раз, когда вызывается метод DisplayRev(), в нем происходит проверка длины символьной строки, представленной аргументом str. Если длина строки не равна нулю, то метод DisplayRev() вызывается рекурсивно с новой строкой, которая меньше исходной строки на один символ. Этот процесс повторяется до тех пор, пока данному методу не будет передана строка нулевой длины. После этого начнется раскручиваться в обратном порядке механизм всех рекурсивных вызовов метода DisplayRev(). При возврате из каждого такого вызова выводится первый символ строки, представленной аргументом str, а в итоге вся строка выводится в обратном порядке.

Рекурсивные варианты многих процедур могут выполняться немного медленнее, чем их итерационные эквиваленты из-за дополнительных затрат системных ресурсов на неоднократные вызовы метода. Если же таких вызовов окажется слишком много, то в конечном итоге может быть переполнен системный стек. А поскольку параметры и локальные переменные рекурсивного метода хранятся в системном стеке и при каж-

дом новом вызове этого метода создается их новая копия, то в какой-то момент стек может оказаться исчерпанным. В этом случае возникает исключительная ситуация, и общеязыковая исполняющая среда (CLR) генерирует соответствующее исключение. Но беспокоиться об этом придется лишь в том случае, если рекурсивная процедура выполняется неправильно.

Главное преимущество рекурсии заключается в том, что она позволяет реализовать некоторые алгоритмы яснее и проще, чем итерационным способом. Например, алгоритм быстрой сортировки довольно трудно реализовать итерационным способом. А некоторые задачи, например искусственного интеллекта, очевидно, требуют именно рекурсивного решения.

При написании рекурсивных методов следует непременно указать в соответствующем месте условный оператор, например `if`, чтобы организовать возврат из метода без рекурсии. В противном случае возврата из вызванного однажды рекурсивного метода может вообще не произойти. Подобного рода ошибка весьма характерна для реализации рекурсии в практике программирования. В этом случае рекомендуется пользоваться операторами, содержащими вызовы метода `WriteLine()`, чтобы следить за происходящим в рекурсивном методе и прервать его выполнение, если в нем обнаружится ошибка.

Применение ключевого слова `static`

Иногда требуется определить такой член класса, который будет использоваться независимо от всех остальных объектов этого класса. Как правило, доступ к члену класса организуется посредством объекта этого класса, но в то же время можно создать член класса для самостоятельного применения без ссылки на конкретный экземпляр объекта. Для того чтобы создать такой член класса, достаточно указать в самом начале его объявления ключевое слово `static`. Если член класса объявляется как `static`, то он становится доступным до создания любых объектов своего класса и без ссылки на какой-нибудь объект. С помощью ключевого слова `static` можно объявлять как переменные, так и методы. Наиболее характерным примером члена типа `static` служит метод `Main()`, который объявляется таковым потому, что он должен вызываться операционной системой в самом начале выполняемой программы.

Для того чтобы воспользоваться членом типа `static` за пределами класса, достаточно указать имя этого класса с оператором-точкой. Но создавать объект для этого не нужно. В действительности член типа `static` оказывается доступным не по ссылке на объект, а по имени своего класса. Так, если требуется присвоить значение 10 переменной `count` типа `static`, являющейся членом класса `Timer`, то для этой цели можно воспользоваться следующей строкой кода.

```
Timer.count = 10;
```

Эта форма записи подобна той, что используется для доступа к обычным переменным экземпляра посредством объекта, но в ней указывается имя класса, а не объекта. Аналогичным образом можно вызвать метод типа `static`, используя имя класса и оператор-точку.

Переменные, объявляемые как `static`, по существу, являются глобальными. Когда же объекты объявляются в своем классе, то копия переменной типа `static` не создается. Вместо этого все экземпляры класса совместно пользуются одной и той же

переменной типа static. Такая переменная инициализируется перед ее применением в классе. Когда же ее инициализатор не указан явно, то она инициализируется нулевым значением, если относится к числовому типу данных, пустым значением, если относится к ссылочному типу, или же логическим значением false, если относится к типу bool. Таким образом, переменные типа static всегда имеют какое-то значение.

Метод типа static отличается от обычного метода тем, что его можно вызывать по имени его класса, не создавая экземпляр объекта этого класса. Пример такого вызова уже приводился ранее. Это был метод Sqrt() типа static, относящийся к классу System.Math из стандартной библиотеки классов C#.

Ниже приведен пример программы, в которой объявляются переменная и метод типа static.

```
// Использовать модификатор static.

using System;

class StaticDemo {

  // Переменная типа static.
  public static int Val = 100;

  // Метод типа static.
  public static int ValDiv2() {
    return Val/2;
  }
}

class SDemo {
  static void Main() {

    Console.WriteLine("Исходное значение переменной " +
                      "StaticDemo.Val равно " + StaticDemo.Val);

    StaticDemo.Val = 8;
    Console.WriteLine("Текущее значение переменной" +
                      "StaticDemo.Val равно " + StaticDemo.Val);
    Console.WriteLine("StaticDemo.ValDiv2(): " + StaticDemo.ValDiv2());
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Исходное значение переменной StaticDemo.Val равно 100
Текущее значение переменной StaticDemo.Val равно 8
StaticDemo.ValDiv2(): 4
```

Как следует из приведенного выше результата, переменная типа static инициализируется до создания любого объекта ее класса.

На применение методов типа static накладывается ряд следующих ограничений.

- В методе типа static должна отсутствовать ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта.

- В методе типа `static` допускается непосредственный вызов только других методов типа `static`, но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа `static` не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать.
- Аналогичные ограничения накладываются на данные типа `static`. Для метода типа `static` непосредственно доступными оказываются только другие данные типа `static`, определенные в его классе. Он, в частности, не может оперировать переменной экземпляра своего класса, поскольку у него отсутствуют объекты, которыми он мог бы оперировать.

Ниже приведен пример класса, в котором недопустим метод `ValDivDenom()` типа `static`.

```
class StaticError {
  public int Denom = 3; // обычная переменная экземпляра
  public static int Val = 1024; // статическая переменная

  /* Ошибка! Непосредственный доступ к нестатической
     переменной из статического метода недопустим. */
  static int ValDivDenom() {
    return Val/Denom; // не подлежит компиляции!
  }
}
```

В данном примере кода `Denom` является обычной переменной, которая недоступна из метода типа `static`. Но в то же время в этом методе можно воспользоваться переменной `Val`, поскольку она объявлена как `static`.

Аналогичная ошибка возникает при попытке вызвать нестатический метод из статического метода того же самого класса, как в приведенном ниже примере.

```
using System;

class AnotherStaticError {

  // Нестатический метод.
  void NonStaticMeth() {
    Console.WriteLine("В методе NonStaticMeth().");
  }

  /* Ошибка! Непосредственный вызов нестатического
     метода из статического метода недопустим. */
  static void staticMeth() {
  NonStaticMeth(); // не подлежит компиляции!
  }
}
```

В данном случае попытка вызвать нестатический метод (т.е. метод экземпляра) из статического метода приводит к ошибке во время компиляции.

Следует особо подчеркнуть, что из метода типа `static` нельзя вызывать методы экземпляра и получать доступ к переменным экземпляра его класса, как это обычно делается посредством объектов данного класса. И объясняется это тем, что

без указания конкретного объекта переменная или метод экземпляра оказываются недоступными. Например, приведенный ниже фрагмент кода считается совершенно верным.

```
class MyClass {

  // Нестатический метод.
  void NonStaticMeth() {
    Console.WriteLine("В методе NonStaticMeth().");
  }

  /* Нестатический метод может быть вызван из
     статического метода по ссылке на объект. */
  public static void staticMeth(MyClass ob) {
  ob.NonStaticMeth(); // все верно!
  }
}
```

В данном примере метод `NonStaticMeth()` вызывается из метода `staticMeth()` по ссылке на объект `ob` типа `MyClass`.

Поля типа `static` не зависят от конкретного объекта, и поэтому они удобны для хранения информации, применимой ко всему классу. Ниже приведен пример программы, демонстрирующей подобную ситуацию. В этой программе поле типа `static` служит для хранения количества существующих объектов.

```
// Использовать поле типа static для подсчета
// экземпляров существующих объектов.

using System;

class CountInst {
  static int count = 0;

  // Инкрементировать подсчет, когда создается объект.
  public CountInst() {
  count++;
  }

// Декрементировать подсчет, когда уничтожается объект.
  ~CountInst() {
  count--;
  }

  public static int GetCount() {
    return count;
  }
}

class CountDemo {
  static void Main() {
    CountInst ob;

    for(int i=0; i < 10; i++) {
```

```
      ob = new CountInst();
      Console.WriteLine("Текущий подсчет: " + CountInst.GetCount());
    }
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Текущий подсчет: 1
Текущий подсчет: 2
Текущий подсчет: 3
Текущий подсчет: 4
Текущий подсчет: 5
Текущий подсчет: 6
Текущий подсчет: 7
Текущий подсчет: 8
Текущий подсчет: 9
Текущий подсчет: 10
```

Всякий раз, когда создается объект типа `CountInst`, инкрементируется поле `count` типа `static`. Но всякий раз, когда такой объект утилизируется, поле `count` декрементируется. Следовательно, поле `count` всегда содержит количество существующих в настоящий момент объектов. И это становится возможным только благодаря использованию поля типа `static`. Аналогичный подсчет нельзя организовать с помощью переменной экземпляра, поскольку он имеет отношение ко всему классу, а не только к конкретному экземпляру объекта этого класса.

Ниже приведен еще один пример применения статических членов класса. Ранее в этой главе было показано, как объекты создаются с помощью фабрики класса. В том примере фабрика была нестатическим методом, а это означало, что фабричный метод можно было вызывать только по ссылке на объект, который нужно было предварительно создать. Но фабрику класса лучше реализовать как метод типа `static`, что даст возможность вызывать этот фабричный метод, не создавая ненужный объект. Именно это улучшение и отражено в приведенном ниже измененном примере программы, реализующей фабрику класса.

```
// Использовать статическую фабрику класса.

using System;

class MyClass {
  int a, b;

  // Создать фабрику для класса MyClass.
  static public MyClass Factory(int i, int j) {
    MyClass t = new MyClass();

    t.a = i;
    t.b = j;

    return t; // возвратить объект
  }

  public void Show() {
    Console.WriteLine("a и b: " + a + " " + b);
```

```
  }
}

class MakeObjects {
  static void Main() {
    int i, j;

    // Сформировать объекты, используя фабрику.
    for(i=0, j=10; i < 10; i++, j--) {
      MyClass ob = MyClass.Factory(i, j); // создать объект
      ob.Show();
    }

    Console.WriteLine();
  }
}
```

В этом варианте программы фабричный метод `Factory()` вызывается по имени его класса в следующей строке кода.

```
MyClass ob = MyClass.Factory(i, j); // создать объект
```

Теперь нет необходимости создавать объект класса `MyClass`, перед тем как пользоваться фабрикой этого класса.

Статические конструкторы

Конструктор можно также объявить как `static`. Статический конструктор, как правило, используется для инициализации компонентов, применяемых ко всему классу, а не к отдельному экземпляру объекта этого класса. Поэтому члены класса инициализируются статическим конструктором до создания каких-либо объектов этого класса. Ниже приведен простой пример применения статического конструктора.

```
// Применить статический конструктор.

using System;

class Cons {
  public static int alpha;
  public int beta;

  // Статический конструктор.
  static Cons() {
    alpha = 99;
    Console.WriteLine("В статическом конструкторе.");
  }

  // Конструктор экземпляра.
  public Cons() {
    beta = 100;
    Console.WriteLine("В конструкторе экземпляра.");
  }
}
```

```
class ConsDemo {
  static void Main() {
    Cons ob = new Cons();
    Console.WriteLine("Cons.alpha: " + Cons.alpha);
    Console.WriteLine("ob.beta: " + ob.beta);
  }
}
```

При выполнении этого кода получается следующий результат.

```
В статическом конструкторе.
В конструкторе экземпляра.
Cons.alpha: 99
ob.beta: 100
```

Обратите внимание на то, что конструктор типа `static` вызывается автоматически, когда класс загружается впервые, причем до конструктора экземпляра. Из этого можно сделать более общий вывод: статический конструктор должен выполняться до любого конструктора экземпляра. Более того, у статических конструкторов отсутствуют модификаторы доступа — они пользуются доступом по умолчанию, а следовательно, их нельзя вызывать из программы.

Статические классы

Класс можно объявлять как `static`. Статический класс обладает двумя основными свойствами. Во-первых, объекты статического класса создавать нельзя. И во-вторых, статический класс должен содержать только статические члены. Статический класс создается по приведенной ниже форме объявления класса, видоизмененной с помощью ключевого слова `static`.

```
static class имя_класса { // ...
```

В таком классе все члены должны быть объявлены как `static`. Ведь если класс становится статическим, то это совсем не означает, что статическими становятся и все его члены.

Статические классы применяются главным образом в двух случаях. Во-первых, статический класс требуется при создании *метода расширения*. Методы расширения связаны в основном с языком LINQ и поэтому подробнее рассматриваются в главе 19. И во-вторых, статический класс служит для хранения совокупности связанных друг с другом статических методов. Именно это его применение и рассматривается ниже.

В приведенном ниже примере программы класс `NumericFn` типа `static` служит для хранения ряда статических методов, оперирующих числовым значением. А поскольку все члены класса `NumericFn` объявлены как `static`, то этот класс также объявлен как `static`, чтобы исключить получение экземпляров его объектов. Таким образом, класс `NumericFn` выполняет организационную роль, предоставляя удобные средства для группирования логически связанных методов.

```
// Продемонстрировать применение статического класса.

using System;
```

```
static class NumericFn {
  // Возвратить обратное числовое значение.
  static public double Reciprocal(double num) {
    return 1/num;
  }

  // Возвратить дробную часть числового значения.
  static public double FracPart(double num) {
    return num - (int) num;
  }

  // Возвратить логическое значение true, если числовое
  // значение переменной num окажется четным.
  static public bool IsEven(double num) {
    return (num % 2) == 0 ? true : false;
  }

  // Возвратить логическое значение true, если числовое
  // значение переменной num окажется нечетным.
  static public bool IsOdd(double num) {
    return !IsEven(num);
  }
}

class StaticClassDemo {
  static void Main() {
    Console.WriteLine("Обратная величина числа 5 равна " +
                      NumericFn.Reciprocal(5.0));
    Console.WriteLine("Дробная часть числа 4.234 равна " +
                      NumericFn.FracPart(4.234));

    if(NumericFn.IsEven(10))
      Console.WriteLine("10 — четное число.");

    if(NumericFn.IsOdd(5))
      Console.WriteLine("5 — нечетное число.");

    // Далее следует попытка создать экземпляр объекта класса NumericFn,
    // что может стать причиной появления ошибки.
    // NumericFn ob = new NumericFn(); // Ошибка!
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Обратная величина числа 5 равна 0.2
Дробная часть числа 4.234 равна 0.234
10 — четное число.
5 — нечетное число.
```

Обратите внимание на то, что последняя строка приведенной выше программы закомментирована. Класс `NumericFn` является статическим, и поэтому любая попытка создать объект этого класса может привести к ошибке во время компиляции. Ошибкой будет также считаться попытка сделать нестатическим член класса `NumericFn`.

И последнее замечание: несмотря на то, что для статического класса не допускается наличие конструктора экземпляра, у него может быть статический конструктор.

ГЛАВА 9

Перегрузка операторов

В языке C# допускается определять назначение оператора по отношению к создаваемому классу. Этот процесс называется *перегрузкой операторов*. Благодаря перегрузке расширяется сфера применения оператора в классе. При этом действие оператора полностью контролируется и может меняться в зависимости от конкретного класса. Например, оператор + может использоваться для ввода объекта в связный список в одном классе, где определяется такой список, тогда как в другом классе его назначение может оказаться совершенно иным.

Когда оператор перегружается, ни одно из его первоначальных назначений не теряется. Он просто выполняет еще одну, новую операцию относительно конкретного объекта. Поэтому перегрузка оператора +, например, для обработки связного списка не меняет его назначение по отношению к целым числам, т.е. к их сложению.

Главное преимущество перегрузки операторов заключается в том, что она позволяет плавно интегрировать класс нового типа в среду программирования. Подобного рода расширяемость типов является важной составляющей эффективности такого объектно-ориентированного языка программирования, как C#. Как только для класса определяются операторы, появляется возможность оперировать объектами этого класса, используя обычный синтаксис выражений в C#. Перегрузка операторов является одной из самых сильных сторон языка C#.

Основы перегрузки операторов

Перегрузка операторов тесно связана с перегрузкой методов. Для перегрузки оператора служит ключевое слово `operator`, определяющее *операторный метод*, который, в свою очередь, определяет действие оператора относительно своего класса.

Существуют две формы операторных методов (`operator`): одна — для унарных операторов, другая — для бинарных. Ниже приведена общая форма для каждой разновидности этих методов.

```
// Общая форма перегрузки унарного оператора.
public static возвращаемый_тип operator op(тип_параметра операнд)
{
  // операции
}

// Общая форма перегрузки бинарного оператора.
public static возвращаемый_тип operator op(тип_параметра1 операнд1,
                                            тип_параметра1 операнд2)
{
  // операции
}
```

Здесь вместо *op* подставляется перегружаемый оператор, например + или /; а *возвращаемый_тип* обозначает конкретный тип значения, возвращаемого указанной операцией. Это значение может быть любого типа, но зачастую оно указывается такого же типа, как и у класса, для которого перегружается оператор. Такая корреляция упрощает применение перегружаемых операторов в выражениях. Для унарных операторов *операнд* обозначает передаваемый операнд, а для бинарных операторов то же самое обозначают *операнд1* и *операнд2*. Обратите внимание на то, что операторные методы должны иметь оба типа, `public` и `static`.

Тип операнда унарных операторов должен быть таким же, как и у класса, для которого перегружается оператор. А в бинарных операторах хотя бы один из операндов должен быть такого же типа, как и у его класса. Следовательно, в C# не допускается перегрузка любых операторов для объектов, которые еще не были созданы. Например, назначение оператора + нельзя переопределить для элементов типа `int` или `string`.

И еще одно замечание: в параметрах оператора нельзя использовать модификатор `ref` или `out`.

Перегрузка бинарных операторов

Для того чтобы продемонстрировать принцип действия перегрузки операторов, начнем с простого примера, в котором перегружаются два оператора — + и -. В приведенной ниже программе создается класс `ThreeD`, содержащий координаты объекта в трехмерном пространстве. Перегружаемый оператор + складывает отдельные координаты одного объекта типа `ThreeD` с координатами другого. А перегружаемый оператор - вычитает координаты одного объекта из координат другого.

```
// Пример перегрузки бинарных операторов.

using System;
```

```
// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить бинарный оператор +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложить координаты двух точек и возвратить результат. */
    result.x = op1.x + op2.x; // Эти операторы выполняют
    result.y = op1.y + op2.y; // целочисленное сложение,
    result.z = op1.z + op2.z; //сохраняя свое исходное назначение.

    return result;
  }

  // Перегрузить бинарный оператор -.
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();
    /* Обратите внимание на порядок следования операндов:
       op1 — левый операнд, а op2 — правый операнд. */
    result.x = op1.x - op2.x; // Эти операторы
    result.y = op1.y - op2.y; // выполняют целочисленное
    result.z = op1.z - op2.z; // вычитание

    return result;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c;

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.WriteLine();

    Console.Write("Координаты точки b: ");
    b.Show();
    Console.WriteLine();
```

```
    c = a + b; // сложить координаты точек a и b
    Console.Write("Результат сложения a + b: ");
    c.Show();
    Console.WriteLine();

    c = a + b + c; // сложить координаты точек a, b и c
    Console.Write("Результат сложения a + b + c: ");
    c.Show();
    Console.WriteLine();

    c = c - a; // вычесть координаты точки a
    Console.Write("Результат вычитания c - a: ");
    c.Show();
    Console.WriteLine();

    c = c - b; // вычесть координаты точки b
    Console.Write("Результат вычитания c - b: ");
    c.Show();
    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения a + b + c: 22, 24, 26

Результат вычитания c - a: 21, 22, 23

Результат вычитания c - b: 11, 12, 13
```

Внимательно проанализируем приведенную выше программу, начиная с перегружаемого оператора +. Когда оператор + оперирует двумя объектами типа `ThreeD`, то величины их соответствующих координат складываются, как показано в объявлении операторного метода `operator+()`. Следует, однако, иметь в виду, что этот оператор не видоизменяет значения своих операндов, а лишь возвращает новый объект типа `ThreeD`, содержащий результат операции сложения координат. Для того чтобы стало понятнее, почему операция + не меняет содержимое объектов, выступающих в роли ее операндов, обратимся к примеру обычной операции арифметического сложения: 10 + 12. Результат этой операции равен 22, но она не меняет ни число 10, ни число 12. Несмотря на то что ни одно из правил не препятствует перегруженному оператору изменить значение одного из своих операндов, все же лучше, чтобы действия этого оператора соответствовали его обычному назначению.

Обратите внимание на то, что метод `operator+()` возвращает объект типа `ThreeD`. Этот метод мог бы возвратить значение любого допустимого в C# типа, но благодаря тому что он возвращает объект типа `ThreeD`, оператор + можно использовать в таких составных выражениях, как `a+b+c`. В данном случае выражение `a+b` дает результат типа `ThreeD`, который можно затем сложить с объектом `c` того же типа. Если бы

выражение a+b давало результат другого типа, то вычислить составное выражение a+b+c было бы просто невозможно.

Следует также подчеркнуть, что когда отдельные координаты точек складываются в операторе operator+(), то в результате такого сложения получаются целые значения, поскольку отдельные координаты x, y и z представлены целыми величинами. Но сама перегрузка оператора + для объектов типа ThreeD не оказывает никакого влияния на операцию сложения целых значений, т.е. она не меняет первоначальное назначение этого оператора.

А теперь проанализируем операторный метод operator-(). Оператор - действует так же, как и оператор +, но для него важен порядок следования операндов. Напомним, что сложение носит коммутативный характер (от перестановки слагаемых сумма не меняется), чего нельзя сказать о вычитании: A – B не то же самое, что и B – A! Для всех двоичных операторов первым параметром операторного метода является левый операнд, а вторым параметром — правый операнд. Поэтому, реализуя перегружаемые варианты некоммутативных операторов, следует помнить, какой именно операнд должен быть указан слева и какой — справа.

Перегрузка унарных операторов

Унарные операторы перегружаются таким же образом, как и бинарные. Главное отличие заключается, конечно, в том, что у них имеется лишь один операнд. В качестве примера ниже приведен метод, перегружающий оператор унарного минуса для класса ThreeD.

```
// Перегрузить оператор унарного минуса.
public static ThreeD operator -(ThreeD op)
{
  ThreeD result = new ThreeD();

  result.x = -op.x;
  result.y = -op.y;
  result.z = -op.z;

  return result;
}
```

В данном примере создается новый объект, в полях которого сохраняются отрицательные значения операнда перегружаемого унарного оператора, после чего этот объект возвращается операторным методом. Обратите внимание на то, что сам операнд не меняется. Это означает, что и в данном случае обычное назначение оператора унарного минуса сохраняется. Например, результатом выражения

```
a = -b
```

является отрицательное значение операнда b, но сам операнд b не меняется.

В C# перегрузка операторов ++ и -- осуществляется довольно просто. Для этого достаточно возвратить инкрементированное или декрементированное значение, но не изменять вызывающий объект. А все остальное возьмет на себя компилятор C#, различая префиксные и постфиксные формы этих операторов. В качестве примера ниже приведен операторный метод operator++() для класса ThreeD.

```
// Перегрузить унарный оператор ++.
public static ThreeD operator ++(ThreeD op)
{
  ThreeD result = new ThreeD();

  // Возвратить результат инкрементирования.
  result.x = op.x + 1;
  result.y = op.y + 1;
  result.z = op.z + 1;

  return result;
}
```

Ниже приведен расширенный вариант предыдущего примера программы, в котором демонстрируется перегрузка унарных операторов - и ++.

```
// Пример перегрузки бинарных и унарных операторов.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить бинарный оператор +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложить координаты двух точек и возвратить результат. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;
    return result;
  }

  // Перегрузить бинарный оператор -.
  public static ThreeD operator -(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Обратить внимание на порядок следования операндов:
       op1 — левый операнд, op2 — правый операнд. */
    result.x = op1.x - op2.x;
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
  }

  // Перегрузить унарный оператор -.
  public static ThreeD operator -(ThreeD op)
```

```
  {
    ThreeD result = new ThreeD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
  }

  // Перегрузить унарный оператор ++.
  public static ThreeD operator ++(ThreeD op)
  {
    ThreeD result = new ThreeD();

    // Возвратить результат инкрементирования.
    result.x = op.x + 1;
    result.y = op.y + 1;
    result.z = op.z + 1;

    return result;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.WriteLine();

    Console.Write("Координаты точки b: ");
    b.Show();
    Console.WriteLine();

c = a + b; // сложить координаты точек a и b
    Console.Write("Результат сложения a + b: ");
    c.Show();
    Console.WriteLine();

    c = a + b + c; // сложить координаты точек a, b и c
    Console.Write("Результат сложения a + b + c: ");
    c.Show();
    Console.WriteLine();
```

```
    c = c - a; // вычесть координаты точки a
    Console.Write("Результат вычитания c - a: ");
    c.Show();
    Console.WriteLine();

    c = c - b; // вычесть координаты точки b
    Console.Write("Результат вычитания c - b: ");
    c.Show();
    Console.WriteLine();

    c = -a; // присвоить точке c отрицательные координаты точки a
    Console.Write("Результат присваивания -a: ");
    c.Show();
    Console.WriteLine();

    c = a++; // присвоить точке c координаты точки a,
             // а затем инкрементировать их
    Console.WriteLine("Если c = a++");
    Console.Write("то координаты точки c равны ");
    c.Show();
    Console.Write("а координаты точки a равны  ");
    a.Show();

    // Установить исходные координаты (1,2,3) точки a
    a = new ThreeD(1, 2, 3);
    Console.Write("\nУстановка исходных координат точки a: ");
    a.Show();

    c = ++a; // инкрементировать координаты точки a,
             // а затем присвоить их точке c
    Console.WriteLine("\nЕсли c = ++a");
    Console.Write("то координаты точки c равны ");
    c.Show();
    Console.Write("а координаты точки a равны  ");
    a.Show();
  }
}
```

Вот к какому результату приводит выполнение данной программы.

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения a + b + c: 22, 24, 26

Результат вычитания c - a: 21, 22, 23

Результат вычитания c - b: 11, 12, 13

Результат присваивания -a: -1, -2, -3
Если c = a++
то координаты точки c равны 1, 2, 3
```

```
a координаты точки a равны  2, 3, 4

Установка исходных координат точки a: 1, 2, 3

Если c = ++a ,
то координаты точки c равны 2, 3, 4
a координаты точки a равны  2, 3, 4
```

Выполнение операций со встроенными в C# типами данных

Для любого заданного класса и оператора имеется также возможность перегрузить сам операторный метод. Это, в частности, требуется для того, чтобы разрешить операции с типом класса и другими типами данных, в том числе и встроенными. Вновь обратимся к классу `ThreeD`. На примере этого класса ранее было показано, как оператор + перегружается для сложения координат одного объекта типа `ThreeD` с координатами другого. Но это далеко не единственный способ определения операции сложения для класса `ThreeD`. Так, было бы не менее полезно прибавить целое значение к каждой координате объекта типа `ThreeD`. Подобная операция пригодилась бы для переноса осей координат. Но для ее выполнения придется перегрузить оператор + еще раз, как показано ниже.

```
// Перегрузить бинарный оператор + для сложения объекта
// типа ThreeD и целого значения типа int.
public static ThreeD operator +(ThreeD op1, int op2)
{
  ThreeD result = new ThreeD();
  result.x = op1.x + op2;
  result.y = op1.y + op2;
  result.z = op1.z + op2;

  return result;
}
```

Как видите, второй параметр операторного метода имеет тип `int`. Следовательно, в этом методе разрешается сложение целого значения с каждым полем объекта типа `ThreeD`. Такая операция вполне допустима, потому что, как пояснялось выше, при перегрузке бинарного оператора один из его операндов должен быть того же типа, что и класс, для которого этот оператор перегружается. Но у второго операнда этого оператора может быть любой другой тип.

Ниже приведен вариант класса `ThreeD` с двумя перегружаемыми методами оператора +.

```
// Перегрузить бинарный оператор + дважды:
// один раз – для сложения объектов класса ThreeD,
// а другой раз – для сложения объекта типа ThreeD и целого значения типа int.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты
```

```
  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить бинарный оператор + для сложения объектов класса ThreeD.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложить координаты двух точек и возвратить результат. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Перегрузить бинарный оператор + для сложения
  // объекта типа ThreeD и целого значения типа int.
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    Console.Write("Координаты точки a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // сложить объекты класса ThreeD
    Console.Write("Результат сложения a + b: ");
    c.Show();
    Console.WriteLine();

    c = b + 10; // сложить объект типа ThreeD и целое значение типа int
```

```
    Console.Write("Результат сложения b + 10: ");
    c.Show();
  }
}
```

При выполнении этого кода получается следующий результат.

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения b + 10: 20, 20, 20
```

Как подтверждает приведенный выше результат, когда оператор + применяется к двум объектам класса `ThreeD`, то складываются их координаты. А когда он применяется к объекту типа `ThreeD` и целому значению, то координаты этого объекта увеличиваются на заданное целое значение.

Продемонстрированная выше перегрузка оператора +, безусловно, расширяет полезные функции класса `ThreeD`, тем не менее, она делает это не до конца. И вот почему. Метод `operator+(ThreeD, int)` позволяет выполнять операции, подобные следующей.

```
ob1 = ob2 + 10;
```

Но, к сожалению, он не позволяет выполнять операции, аналогичные следующей.

```
ob1 = 10 + ob2;
```

Дело в том, что второй целочисленный аргумент данного метода обозначает правый операнд бинарного оператора +, но в приведенной выше строке кода целочисленный аргумент указывается слева. Для того чтобы разрешить выполнение такой операции сложения, придется перегрузить оператор + еще раз. В этом случае первый параметр операторного метода должен иметь тип `int`, а второй параметр — тип `ThreeD`. Таким образом, в одном варианте метода `operator+()` выполняется сложение объекта типа `ThreeD` и целого значения, а во втором — сложение целого значения и объекта типа `ThreeD`. Благодаря такой перегрузке оператора + (или любого другого бинарного оператора) допускается появление встроенного типа данных как с левой, так и с правой стороны данного оператора. Ниже приведен еще один вариант класса `ThreeD`, в котором бинарный оператор + перегружается описанным выше образом.

```
// Перегрузить бинарный оператор + трижды:
// один раз — для сложения объектов класса ThreeD,
// второй раз — для сложения объекта типа ThreeD и целого значения типа int,
// а третий раз — для сложения целого значения типа int и объекта типа ThreeD.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты
  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
```

```
  // Перегрузить бинарный оператор + для сложения объектов класса ThreeD.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    /* Сложить координаты двух точек и возвратить результат. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Перегрузить бинарный оператор + для сложения
  // объекта типа ThreeD и целого значения типа int.
  public static ThreeD operator +(ThreeD op1, int op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

   return result;
  }

  // Перегрузить бинарный оператор + для сложения
  // целого значения типа int и объекта типа ThreeD.
  public static ThreeD operator +(int op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;

    return result;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
   Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    Console.Write("Координаты точки a: ");
    a.Show();
```

```
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // сложить объекты класса ThreeD
    Console.Write("Результат сложения a + b: ");
    c.Show();
    Console.WriteLine();

    c = b + 10; // сложить объект типа ThreeD и целое значение типа int
    Console.Write("Результат сложения b + 10: ");
    c.Show();
    Console.WriteLine();

    c = 15 + b; // сложить целое значение типа int и объект типа ThreeD
    Console.Write("Результат сложения 15 + b: ");
    c.Show();
  }
}
```

Выполнение этого кода дает следующий результат.

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10

Результат сложения a + b: 11, 12, 13

Результат сложения b + 10: 20, 20, 20

Результат сложения 15 + b: 25, 25, 25
```

Перегрузка операторов отношения

Операторы отношения, например == и <, могут также перегружаться, причем очень просто. Как правило, перегруженный оператор отношения возвращает логическое значение `true` и `false`. Это вполне соответствует правилам обычного применения подобных операторов и дает возможность использовать их перегружаемые разновидности в условных выражениях. Если же возвращается результат другого типа, то тем самым сильно ограничивается применимость операторов отношения.

Ниже приведен очередной вариант класса `ThreeD`, в котором перегружаются операторы < и >. В данном примере эти операторы служат для сравнения объектов `ThreeD`, исходя из их расстояния до начала координат. Один объект считается больше другого, если он находится дальше от начала координат. А кроме того, один объект считается меньше другого, если он находится ближе к началу координат. Такой вариант реализации позволяет, в частности, определить, какая из двух заданных точек находится на большей сфере. Если же ни один из операторов не возвращает логическое значение `true`, то обе точки находятся на одной и той же сфере. Разумеется, возможны и другие алгоритмы упорядочения.

```
// Перегрузить операторы < и >.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить оператор <.
  public static bool operator <(ThreeD op1, ThreeD op2)
  {
    if(Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) <
       Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
      return true;
    else
      return false;
  }

  // Перегрузить оператор >.
  public static bool operator >(ThreeD op1, ThreeD op2)
  {
    if(Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) >
       Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
      return true;
    else
    return false;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(1, 2, 3);
    ThreeD d = new ThreeD(6, 7, 5);

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.Write("Координаты точки c: ");
    c.Show();
    Console.Write("Координаты точки d: ");
    d.Show();
    Console.WriteLine();
```

```
    if(a > c) Console.WriteLine("a > c истинно");
    if(a < c) Console.WriteLine("a < c истинно");
    if(a > b) Console.WriteLine("a > b истинно");
    if(a < b) Console.WriteLine("a < b истинно");

    if(a > d) Console.WriteLine("a > d истинно");
    else if(a < d) Console.WriteLine("a < d истинно");
    else Console.WriteLine("Точки a и d находятся на одном расстоянии " +
                           "от начала отсчета");
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 1, 2, 3
Координаты точки d: 6, 7, 5

a ≥ c истинно
a < b истинно
Точки a и d находятся на одном расстоянии от начала отсчета
```

На перегрузку операторов отношения накладывается следующее важное ограничение: они должны перегружаться попарно. Так, если перегружается оператор <, то следует перегрузить и оператор >, и наоборот. Ниже приведены составленные в пары перегружаемые операторы отношения.

==	!=
<	>
<=	>=

И еще одно замечание: если перегружаются операторы == и !=, то для этого обычно требуется также переопределить методы `Object.Equals()` и `Object.GetHashCode()`. Эти методы и способы их переопределения подробнее рассматриваются в главе 11.

Перегрузка операторов `true` и `false`

Ключевые слова `true` и `false` можно также использовать в качестве унарных операторов для целей перегрузки. Перегружаемые варианты этих операторов позволяют определить назначение ключевых слов `true` и `false` специально для создаваемых классов. После перегрузки этих ключевых слов в качестве унарных операторов для конкретного класса появляется возможность использовать объекты этого класса для управления операторами `if`, `while`, `for` и `do-while` или же в условном выражении `?`.

Операторы `true` и `false` должны перегружаться попарно, а не раздельно. Ниже приведена общая форма перегрузки этих унарных операторов.

```
public static bool operator true(тип_параметра операнд)
{
  // Возврат логического значения true или false.
```

```
}

public static bool operator false(тип_параметра операнд)
{
  // Возврат логического значения true или false.
}
```

Обратите внимание на то, что и в том и в другом случае возвращается результат типа `bool`.

Ниже приведен пример программы, демонстрирующий реализацию операторов `true` и `false` в классе `ThreeD`. В каждом из этих операторов проверяется следующее условие: если хотя бы одна из координат объекта типа `ThreeD` равна нулю, то этот объект истинен, а если все три его координаты равны нулю, то такой объект ложен. В данном примере программы реализован также оператор декремента исключительно в целях демонстрации.

```
// Перегрузить операторы true и false для класса ThreeD.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить оператор true.
  public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return true; // хотя бы одна координата не равна нулю
      else
        return false;
  }

  // Перегрузить оператор false.
  public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
      return true; // все координаты равны нулю
    else
      return false;
  }

  // Перегрузить унарный оператор --.
  public static ThreeD operator --(ThreeD op)
  {
    ThreeD result = new ThreeD();

    // Возвратить результат декрементирования.
    result.x = op.x - 1;
    result.y = op.y - 1;
    result.z = op.z - 1;

    return result;
```

```
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.Write("Координаты точки c: ");
    c.Show();
    Console.WriteLine();

    if(a) Console.WriteLine("Точка a истинна.");
    else Console.WriteLine("Точка a ложна.");

    if(b) Console.WriteLine("Точка b истинна.");
    else Console.WriteLine("Точка b ложна.");

    if(c) Console.WriteLine("Точка c истинна.");
    else Console.WriteLine("Точка c ложна.");

    Console.WriteLine();

    Console.WriteLine("Управление циклом с помощью объекта класса
ThreeD.");
    do {
      b.Show();
      b--;
    } while(b);
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 0, 0, 0

Точка a истинна
Точка b истинна
Точка c ложна

Управление циклом с помощью объекта класса ThreeD.
10, 10, 10
```

```
9, 9, 9
8, 8, 8
7, 7, 7
6, 6, 6
5, 5, 5
4, 4, 4
3, 3, 3
2, 2, 2
1, 1, 1
```

Обратите внимание на то, как объекты класса `ThreeD` используются для управления условным оператором `if` и оператором цикла `do-while`. Так, в операторах `if` объект типа `ThreeD` проверяется с помощью оператора `true`. Если результат этой проверки оказывается истинным, то оператор `if` выполняется. А в операторе цикла `do-while` объект `b` декрементируется на каждом шаге цикла. Следовательно, цикл повторяется до тех пор, пока проверка объекта `b` дает истинный результат, т.е. этот объект содержит хотя бы одну ненулевую координату. Если же окажется, что объект `b` содержит все нулевые координаты, его проверка с помощью оператора `true` даст ложный результат и цикл завершится.

Перегрузка логических операторов

Как вам должно быть уже известно, в C# предусмотрены следующие логические операторы: `&`, `|`, `!`, `&&` и `||`. Из них перегрузке, безусловно, подлежат только операторы `&`, `|` и `!`. Тем не менее, соблюдая определенные правила, можно извлечь также пользу из укороченных логических операторов `&&` и `||`. Все эти возможности рассматриваются ниже.

Простой способ перегрузки логических операторов

Рассмотрим сначала простейший случай. Если не пользоваться укороченными логическими операторами, то перегрузку операторов `&` и `|` можно выполнять совершенно естественным путем, получая в каждом случае результат типа `bool`. Аналогичный результат, как правило, дает и перегружаемый оператор `!`.

Ниже приведен пример программы, в которой демонстрируется перегрузка логических операторов `!`, `&` и `|` для объектов типа `ThreeD`. Как и в предыдущем примере, объект типа `ThreeD` считается истинным, если хотя бы одна из его координат не равна нулю. Если же все три координаты объекта равны нулю, то он считается ложным.

```
// Простой способ перегрузки логических операторов
// !, | и & для объектов класса ThreeD.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
```

```
  // Перегрузить логический оператор |.
  public static bool operator |(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
      return true;
    else
      return false;
  }

  // Перегрузить логический оператор &.
  public static bool operator &(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
        ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
      return true;
    else
      return false;
  }

  // Перегрузить логический оператор !.
  public static bool operator !(ThreeD op)
  {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return false;
    else return true;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.Write("Координаты точки c: ");
    c.Show();
    Console.WriteLine();

    if(!a) Console.WriteLine("Точка a ложна.");
    if(!b) Console.WriteLine("Точка b ложна.");
    if(!c) Console.WriteLine("Точка c ложна.");

    Console.WriteLine();
```

```
    if(a & b) Console.WriteLine("a & b истинно.");
    else Console.WriteLine("a & b ложно.");

    if(a & c) Console.WriteLine("a & c истинно.");
    else Console.WriteLine("a & c ложно.");

    if(a | b) Console.WriteLine("a | b истинно.");
    else Console.WriteLine("a | b ложно.");

    if(a | c) Console.WriteLine("a | c истинно.");
    else Console.WriteLine("a | c ложно.");
  }
}
```

При выполнении этой программы получается следующий результат.

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 0, 0, 0

Точка c ложна.

a & b истинно.
a & c ложно.
a | b истинно.
a | c истинно.
```

При таком способе перегрузки логических операторов `&`, `|` и `!` методы каждого из них возвращают результат типа `bool`. Это необходимо для того, чтобы использовать рассматриваемые операторы обычным образом, т.е. в тех выражениях, где предполагается результат типа `bool`. Напомним, что для всех встроенных в C# типов данных результатом логической операции должно быть значение типа `bool`. Поэтому вполне разумно предусмотреть возврат значения типа `bool` и в перегружаемых вариантах этих логических операторов. Но, к сожалению, такой способ перегрузки пригоден лишь в том случае, если не требуются укороченные логические операторы.

Как сделать укороченные логические операторы доступными для применения

Для того чтобы применение укороченных логических операторов `&&` и `||` стало возможным, необходимо соблюсти следующие четыре правила. Во-первых, в классе должна быть произведена перегрузка логических операторов `&` и `|`. Во-вторых, перегружаемые методы операторов `&` и `|` должны возвращать значение того же типа, что и у класса, для которого эти операторы перегружаются. В-третьих, каждый параметр должен содержать ссылку на объект того класса, для которого перегружается логический оператор. И в-четвертых, для класса должны быть перегружены операторы `true` и `false`. Если все эти условия выполняются, то укороченные логические операторы автоматически становятся пригодными для применения.

В приведенном ниже примере программы показано, как правильно реализовать логические операторы `&` и `|` в классе `ThreeD`, чтобы сделать доступными для применения укороченные логические операторы `&&` и `||`.

```
/* Более совершенный способ перегрузки логических
   операторов !, | и & для объектов класса ThreeD.
   В этом варианте укороченные логические операторы && и ||
   становятся доступными для применения автоматически. */

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить логический оператор | для укороченного вычисления.
  public static ThreeD operator |(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
        ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
      return new ThreeD(1, 1, 1);
    else
      return new ThreeD(0, 0, 0);
  }

  // Перегрузить логический оператор & для укороченного вычисления.
  public static ThreeD operator &(ThreeD op1, ThreeD op2)
  {
    if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
        ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
      return new ThreeD(1, 1, 1);
    else
      return new ThreeD(0, 0, 0);
  }

  // Перегрузить логический оператор !.
  public static bool operator !(ThreeD op)
  {
    if(op) return false;
    else return true;
  }

  // Перегрузить оператор true.
  public static bool operator true(ThreeD op) {
    if((op.x != 0) || (op.y != 0) || (op.z != 0))
      return true; // хотя бы одна координата не равна нулю
    else
      return false;
  }

  // Перегрузить оператор false.
  public static bool operator false(ThreeD op) {
    if((op.x == 0) && (op.y == 0) && (op.z == 0))
      return true; // все координаты равны нулю
    else
```

```
      return false;
  }

  // Ввести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TrueFalseDemo {
  static void Main() {
    ThreeD a = new ThreeD(5, 6, 7);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD(0, 0, 0);

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.Write("Координаты точки c: ");
    c.Show();
    Console.WriteLine();

    if(a) Console.WriteLine("Точка a истинна.");
    if(b) Console.WriteLine("Точка b истинна.");
    if(c) Console.WriteLine("Точка c истинна.");

    if(!a) Console.WriteLine("Точка a ложна.");
    if(!b) Console.WriteLine("Точка b ложна.");
    if(!c) Console.WriteLine("Точка c ложна.");

    Console.WriteLine();

    Console.WriteLine("Применение логических операторов & и |");
    if(a & b) Console.WriteLine("a & b истинно.");
    else Console.WriteLine("a & b ложно.");

    if(a & c) Console.WriteLine("a & c истинно.");
    else Console.WriteLine("a & c ложно.");

    if(a | b) Console.WriteLine("a | b истинно.");
    else Console.WriteLine("a | b ложно.");

    if(a | c) Console.WriteLine("a | c истинно.");
    else Console.WriteLine("a | c ложно.");

    Console.WriteLine();

    // А теперь применить укороченные логические операторы.
    Console.WriteLine("Применение укороченных" +
                      "логических операторов && и ||");
    if(a && b) Console.WriteLine("a && b истинно.");
    else Console.WriteLine("a && b ложно.");
```

```
    if(a && c) Console.WriteLine("a && c истинно.");
    else Console.WriteLine("a && c ложно.");

    if(a || b) Console.WriteLine("a || b истинно.");
    else Console.WriteLine("a || b ложно.");

    if(a || c) Console.WriteLine("a || c истинно.");
    else Console.WriteLine("a || c ложно.");
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Координаты точки a: 5, 6, 7
Координаты точки b: 10, 10, 10
Координаты точки c: 0, 0, 0

Точка a истинна
Точка b истинна
Точка c ложна.

Применение логических операторов & и |
a & b истинно.
a & c ложно.
a | b истинно.
a | c истинно.

Применение укороченных логических операторов && и ||
a && b истинно.
a && c ложно.
a || b истинно.
a || c истинно.
```

Рассмотрим более подробно, каким образом реализуются логические операторы & и |. Они представлены в следующем фрагменте кода.

```
// Перегрузить логический оператор | для укороченного вычисления.
public static ThreeD operator |(ThreeD op1, ThreeD op2)
{
  if( ((op1.x != 0) || (op1.y != 0) || (op1.z != 0)) |
      ((op2.x != 0) || (op2.y != 0) || (op2.z != 0)) )
    return new ThreeD(1, 1, 1);
  else
    return new ThreeD(0, 0, 0);
}

// Перегрузить логический оператор & для укороченного вычисления.
public static ThreeD operator &(ThreeD op1, ThreeD op2)
{
  if( ((op1.x != 0) && (op1.y != 0) && (op1.z != 0)) &
      ((op2.x != 0) && (op2.y != 0) && (op2.z != 0)) )
    return new ThreeD(1, 1, 1);
  else
    return new ThreeD(0, 0, 0);
}
```

Прежде всего обратите внимание на то, что методы обоих перегружаемых логических операторов теперь возвращают объект типа `ThreeD`. И особенно обратите внимание на то, как формируется этот объект. Если логическая операция дает истинный результат, то создается и возвращается истинный объект типа `ThreeD`, у которого хотя бы одна координата не равна нулю. Если же логическая операция дает ложный результат, то соответственно создается и возвращается ложный объект. Таким образом, результатом вычисления логического выражения `a & b` в следующем фрагменте кода:

```
if(a & b) Console.WriteLine("a & b истинно.");
else Console.WriteLine("a & b ложно.");
```

является объект типа `ThreeD`, который в данном случае оказывается истинным. А поскольку операторы `true` и `false` уже определены, то созданный объект типа `ThreeD` подвергается действию оператора `true` и в конечном итоге возвращается результат типа `bool`. В данном случае он равен `true`, а следовательно, условный оператор `if` успешно выполняется.

Благодаря тому что все необходимые правила соблюдены, укороченные операторы становятся доступными для применения к объектам `ThreeD`. Они действуют следующим образом. Первый операнд проверяется с помощью операторного метода `operator true` (для оператора `||`) или же с помощью операторного метода `operator false` (для оператора `&&`). Если удается определить результат данной операции, то соответствующий перегружаемый оператор (`&` или `|`) далее не выполняется. В противном случае перегружаемый оператор (`&` или `|` соответственно) используется для определения конечного результата. Следовательно, когда применяется укороченный логический оператор `&&` или `||`, то соответствующий логический оператор `&` или `|` вызывается лишь в том случае, если по первому операнду невозможно определить результат вычисления выражения. В качестве примера рассмотрим следующую строку кода из приведенной выше программы.

```
if(a || c) Console.WriteLine("a || c истинно.");
```

В этой строке кода сначала применяется оператор `true` к объекту `a`. В данном случае объект `a` истинен, и поэтому использовать далее операторный метод `|` нет необходимости. Но если переписать данную строку кода следующим образом:

```
if(c || a) Console.WriteLine("c || a истинно.");
```

то оператор `true` был бы сначала применен к объекту `c`, который в данном случае ложен. А это означает, что для определения истинности объекта `a` пришлось бы далее вызывать операторный метод `|`.

Описанный выше способ применения укороченных логических операторов может показаться, на первый взгляд, несколько запутанным, но если подумать, то в таком применении обнаруживается известный практический смысл. Ведь благодаря перегрузке операторов `true` и `false` для класса компилятор получает разрешение на применение укороченных логических операторов, не прибегая к явной их перегрузке. Это дает также возможность использовать объекты в условных выражениях. И вообще, логические операторы `&` и `|` лучше всего реализовывать полностью, если, конечно, не требуется очень узко направленная их реализация.

Операторы преобразования

Иногда объект определенного класса требуется использовать в выражении, включающем в себя данные других типов. В одних случаях для этой цели оказывается

пригодной перегрузка одного или более операторов, а в других случаях — обыкновенное преобразование типа класса в целевой тип. Для подобных ситуаций в C# предусмотрена специальная разновидность операторного метода, называемая *оператором преобразования*. Такой оператор преобразует объект исходного класса в другой тип. Операторы преобразования помогают полностью интегрировать типы классов в среду программирования на C#, разрешая свободно пользоваться классами вместе с другими типами данных, при условии, что определен порядок преобразования в эти типы.

Существуют две формы операторов преобразования: явная и неявная. Ниже они представлены в общем виде:

```
public static explicit operator целевой_тип(исходный_тип v) {return значение;}
public static implicit operator целевой_тип(исходный_тип v) {return значение;}
```

где *целевой_тип* обозначает тот тип, в который выполняется преобразование; *исходный_тип* — тот тип, который преобразуется; *значение* — конкретное значение, приобретаемое классом после преобразования. Операторы преобразования возвращают данные, имеющие *целевой_тип*, причем указывать другие возвращаемые типы данных не разрешается.

Если оператор преобразования указан в неявной форме (`implicit`), то преобразование вызывается автоматически, например, в том случае, когда объект используется в выражении вместе со значением целевого типа. Если же оператор преобразования указан в явной форме (`explicit`), то преобразование вызывается в том случае, когда выполняется приведение типов. Для одних и тех же исходных и целевых типов данных нельзя указывать оператор преобразования одновременно в явной и неявной форме.

Создадим оператор преобразования специально для класса `ThreeD`, чтобы продемонстрировать его применение. Допустим, что требуется преобразовать объект типа `ThreeD` в целое значение, чтобы затем использовать его в целочисленном выражении. Такое преобразование требуется, в частности, для получения произведения всех трех координат объекта. С этой целью мы воспользуемся следующей неявной формой оператора преобразования.

```
public static implicit operator int(ThreeD op1)
{
  return op1.x * op1.y * op1.z;
}
```

Ниже приведен пример программы, демонстрирующей применение этого оператора преобразования.

```
// Пример применения оператора неявного преобразования.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты

  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить бинарный оператор +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
```

```
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Неявное преобразование объекта типа ThreeD к типу int.
  public static implicit operator int(ThreeD op1)
  {
    return op1.x * op1.y * op1.z;
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // сложить координаты точек a и b
    Console.Write("Результат сложения a + b: ");
    c.Show();
    Console.WriteLine();

    i = a; // преобразовать в тип int
    Console.WriteLine("Результат присваивания i = a: " + i);
    Console.WriteLine();

    i = a * 2 - b; // преобразовать в тип int
    Console.WriteLine("Результат вычисления выражения a * 2 - b: " + i
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Координаты точки a: 1, 2, 3

Координаты точки b: 10, 10, 10
```

```
Результат сложения a + b: 11, 12, 13

Результат присваивания i = a: 6

Результат вычисления выражения a * 2 - b: -988
```

Как следует из приведенного выше примера программы, когда объект типа `ThreeD` используется в таком целочисленном выражении, как `i = a`, происходит его преобразование. В этом конкретном случае преобразование приводит к возврату целого значения `6`, которое является произведением координат точки `a`, хранящихся в объекте того же названия. Но если для вычисления выражения преобразование в тип `int` не требуется, то оператор преобразования не вызывается. Именно поэтому операторный метод `operator int()` *не* вызывается при вычислении выражения `c = a + b`.

Но для различных целей можно создать разные операторы преобразования. Так, для преобразования объекта типа `ThreeD` в тип `double` можно было бы определить второй оператор преобразования. При этом каждый вид преобразования выполнялся бы автоматически и независимо от другого.

Оператор неявного преобразования применяется автоматически в следующих случаях: когда в выражении требуется преобразование типов; методу передается объект; осуществляется присваивание и производится явное приведение к целевому типу. С другой стороны, можно создать оператор явного преобразования, вызываемый только тогда, когда производится явное приведение типов. В таком случае оператор явного преобразования не вызывается автоматически. В качестве примера ниже приведен вариант предыдущей программы, переделанный для демонстрации явного преобразования в тип `int`.

```
// Применить явное преобразование.

using System;

// Класс для хранения трехмерных координат.
class ThreeD {
  int x, y, z; // трехмерные координаты
  public ThreeD() { x = y = z = 0; }
  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

  // Перегрузить бинарный оператор +.
  public static ThreeD operator +(ThreeD op1, ThreeD op2)
  {
    ThreeD result = new ThreeD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Выполнить на этот раз явное преобразование типов.
  public static explicit operator int(ThreeD op1)
  {
    return op1.x * op1.y * op1.z;
```

```
  }

  // Вывести координаты X, Y, Z.
  public void Show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class ThreeDDemo {
  static void Main() {
    ThreeD a = new ThreeD(1, 2, 3);
    ThreeD b = new ThreeD(10, 10, 10);
    ThreeD c = new ThreeD();
    int i;

    Console.Write("Координаты точки a: ");
    a.Show();
    Console.WriteLine();
    Console.Write("Координаты точки b: ");
    b.Show();
    Console.WriteLine();

    c = a + b; // сложить координаты точек a и b
    Console.Write("Результат сложения a + b: ");
    c.Show();
    Console.WriteLine();

    i = (int) a; // преобразовать в тип int явно,
                 // поскольку указано приведение типов
    Console.WriteLine("Результат присваивания i = a: " + i);
    Console.WriteLine();

    i = (int)a * 2 - (int)b; // явно требуется приведение типов
    Console.WriteLine("Результат вычисления выражения a * 2 - b: " + i);
  }
}
```

Оператор преобразования теперь указан в явной форме, и поэтому преобразование должно быть явно приведено к типу `int`. Например, следующая строка кода не будет скомпилирована, если исключить приведение типов.

```
i = (int) a; // преобразовать в тип int явно,
             // поскольку указано приведение типов
```

На операторы преобразования накладывается ряд следующих ограничений.

- Исходный или целевой тип преобразования должен относиться к классу, для которого объявлено данное преобразование. В частности, нельзя переопределить преобразование в тип `int`, если оно первоначально указано как преобразование в тип `double`.
- Нельзя указывать преобразование в класс `object` или же из этого класса.
- Для одних и тех же исходных и целевых типов данных нельзя указывать одновременно явное и неявное преобразование.

- Нельзя указывать преобразование базового класса в производный класс. (Подробнее о базовых и производных классах речь пойдет в главе 11.)
- Нельзя указывать преобразование в интерфейс или же из него. (Подробнее об интерфейсах — в главе 12.)

Помимо указанных выше ограничений, имеется ряд рекомендаций, которыми обычно руководствуются при выборе операторов явного или неявного преобразования. Несмотря на все преимущества неявных преобразований, к ним следует прибегать только в тех случаях, когда преобразованию не свойственны ошибки. Во избежание подобных ошибок неявные преобразования должны быть организованы только в том случае, если удовлетворяются следующие условия. Во-первых, информация не теряется, например, в результате усечения, переполнения или потери знака. И во-вторых, преобразование не приводит к исключительной ситуации. Если же неявное преобразование не удовлетворяет этим двум условиям, то следует выбрать явное преобразование.

Рекомендации и ограничения по перегрузке операторов

Действие перегружаемого оператора распространяется на класс, для которого он определяется, и никак не связано с его первоначальным применением к данным встроенных в C# типов. Но ради сохранения ясности структуры и удобочитаемости исходного кода перегружаемый оператор должен, по возможности, отражать основную суть своего первоначального назначения. Например, назначение оператора + для класса `ThreeD` по сути не должно заметно отличаться от его назначения для целочисленных типов данных. Если бы, например, определить оператор + относительно некоторого класса таким образом, чтобы по своему действию он стал больше похожим на оператор /, то вряд ли от этого было бы много проку. Главный принцип перегрузки операторов заключается в следующем: несмотря на то, что перегружаемый оператор может получить любое назначение, ради ясности новое его назначение должно быть так или иначе связано с его первоначальным назначением.

На перегрузку операторов накладывается ряд ограничений. В частности, нельзя изменять приоритет любого оператора или количество операндов, которое требуется для оператора, хотя в операторном методе можно и проигнорировать операнд. Кроме того, имеется ряд операторов, которые нельзя перегружать. А самое главное, что перегрузке не подлежит ни один из операторов присваивания, в том числе и составные, как, например, оператор +=. Ниже перечислены операторы, которые нельзя перегружать. Среди них имеются и такие операторы, которые будут рассматриваться далее в этой книге.

`&&`	`()`	`.`	`?`
`??`	`[]`	`||`	`=`
`=>`	`->`	`as`	`checked`
`default`	`is`	`new`	`sizeof`
`typeof`	`unchecked`		

Несмотря на то что оператор приведения `()` нельзя перегружать явным образом, имеется все же возможность создать упоминавшиеся ранее операторы преобразования, выполняющие ту же самую функцию.

Ограничение, связанное с тем, что некоторые операторы, например +=, нельзя перегружать, на самом деле не является таким уж непреодолимым. Вообще говоря, если оператор определен как перегружаемый и используется в составном операторе присваивания, то обычно вызывается метод этого перегружаемого оператора. Следовательно, при обращении к оператору += в программе автоматически вызывается заранее объявленный вариант метода `operator+()`. Например, в приведенном ниже фрагменте кода метод `operator+()` автоматически вызывается для класса `ThreeD`, а в итоге объект `b` будет содержать координаты 11, 12, 13.

```
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);

b += a; // сложить координаты точек a и b
```

И последнее замечание: несмотря на то, что оператор индексации массива `[]` нельзя перегружать с помощью операторного метода, имеется возможность создать индексаторы, о которых речь пойдет в следующей главе.

Еще один пример перегрузки операторов

Во всех предыдущих примерах программ, представленных в этой главе, для демонстрации перегрузки операторов использовался класс `ThreeD`, и этой цели он служил исправно. Но прежде чем завершить эту главу, было бы уместно рассмотреть еще один пример перегрузки операторов. Общие принципы перегрузки операторов остаются неизменными независимо от применяемого класса, тем не менее, в рассматриваемом ниже примере наглядно демонстрируются сильные стороны такой перегрузки, особенно если это касается расширяемости типов.

В данном примере разрабатывается 4-разрядный целочисленный тип данных и для него определяется ряд операций. Вам, вероятно, известно, что на ранней стадии развития вычислительной техники широко применялся тип данных для обозначения 4-разрядных двоичных величин, называвшихся *полубайтами*, поскольку они составляли половину байта, содержали одну шестнадцатеричную цифру и были удобны для ввода кода полубайтами с пульта ЭВМ, что в те времена считалось привычным занятием для программистов! В наше время этот тип данных применяется редко, но он по-прежнему является любопытным дополнением целочисленных типов данных в C#. По традиции полубайт обозначает целое значение без знака.

В приведенном ниже примере программы тип полубайтовых данных реализуется с помощью класса `Nybble`. В качестве базового для него используется тип `int`, но с ограничением на хранение данных от 0 до 15. В классе `Nybble` определяются следующие операторы.

- Сложение двух объектов типа `Nybble`.
- Сложение значения типа `int` с объектом типа `Nybble`.
- Сложение объекта типа `Nybble` со значением типа `int`.
- Операции сравнения: больше (>) и меньше (<).
- Операция инкремента.
- Преобразование значения типа `int` в объект типа `Nybble`.
- Преобразование объекта типа `Nybble` в значение типа `int`.

Перечисленных выше операций достаточно, чтобы показать, каким образом тип класса Nybble интегрируется в систему типов C#. Но для полноценной реализации этого типа данных придется определить все остальные доступные для него операции. Попробуйте сделать это сами в качестве упражнения.

Ниже полностью приводится класс Nybble, а также класс NybbleDemo, демонстрирующий его применение.

```
// Создать полубайтовый тип 4-разрядных данных под названием Nybble.

using System;

// тип4-разрядных данных.

class Nybble {
  int val; // базовый тип для хранения данных

  public Nybble() { val = 0; }

  public Nybble(int i) {
  val = i;
  val = val & 0xF; // сохранить 4 младших разряда
  }

  // Перегрузить бинарный оператор + для сложения двух объектов типа Nybble.
  public static Nybble operator +(Nybble op1, Nybble op2)
  {
    Nybble result = new Nybble();

    result.val = op1.val + op2.val;

    result.val = result.val & 0xF; // сохранить 4 младших разряда
    return result;
  }

  // Перегрузить бинарный оператор + для сложения
  // объекта типа Nybble и значения типа int.
  public static Nybble operator +(Nybble op1, int op2)
  {
    Nybble result = new Nybble();

    result.val = op1.val + op2;

    result.val = result.val & 0xF; // сохранить 4 младших разряда
    return result;
  }

  // Перегрузить бинарный оператор + для сложения
  // значения типа int и объекта типа Nybble.
  public static Nybble operator +(int op1, Nybble op2)
  {
    Nybble result = new Nybble();

    result.val = op1 + op2.val;
```

```
    result.val = result.val & 0xF; // сохранить 4 младших разряда

    return result;
  }

  // Перегрузить оператор ++.
  public static Nybble operator ++(Nybble op)
  {
    Nybble result = new Nybble();

    result.val = op.val + 1;

    result.val = result.val & 0xF; // сохранить 4 младших разряда
    return result;
  }

  // Перегрузить оператор >.
  public static bool operator >(Nybble op1, Nybble op2)
  {
    if(op1.val > op2.val) return true;
    else return false;
  }

  // Перегрузить оператор <.
  public static bool operator <(Nybble op1, Nybble op2)
  {
    if(op1.val < op2.val) return true;
    else return false;
  }

  // Преобразовать тип Nybble в тип int.
  public static implicit operator int (Nybble op)
  {
    return op.val;
  }

  // Преобразовать тип int в тип Nybble.
  public static implicit operator Nybble (int op)
  {
    return new Nybble(op);
  }
}

class NybbleDemo {
  static void Main() {
    Nybble a = new Nybble(1);
    Nybble b = new Nybble(10);
    Nybble c = new Nybble();
    int t;

    Console.WriteLine("a: " + (int) a);
    Console.WriteLine("b: " + (int) b);

    // Использовать тип Nybble в условном операторе if.
```

```
    if(a < b) Console.WriteLine("a меньше b\n");

    // Сложить два объекта типа Nybble.
    c = a + b;
    Console.WriteLine("c после операции c = a + b: " + (int) c);

    // Сложить значение типа int с объектом типа Nybble.
    a += 5;
    Console.WriteLine("a после операции a += 5: " + (int) a);
    Console.WriteLine();

    // Использовать тип Nybble в выражении типа int.
    t = a * 2 + 3;
    Console.WriteLine("Результат вычисления выражения a * 2 + 3: " + t);
    Console.WriteLine();

    // Продемонстрировать присваивание значения типа int и переполнение.
    a = 19;
    Console.WriteLine("Результат присваивания a = 19: " + (int) a);
    Console.WriteLine();

    // Использовать тип Nybble для управления циклом.
    Console.WriteLine("Управление циклом for " +
                      "с помощью объекта типа Nybble.");
    for(a = 0; a < 10; a++)
      Console.Write((int) a + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
a: 1
b: 10
a меньше b

c после операции c = a + b: 11
a после операции a += 5: 6

Результат вычисления выражения a * 2 + 3: 15

Результат присваивания a = 19: 3

Управление циклом for с помощью объекта типа Nybble.
0 1 2 3 4 5 6 7 8 9
```

Большая часть функций класса `Nybble` не требует особых пояснений. Тем не менее необходимо подчеркнуть ту особую роль, которую операторы преобразования играют в интегрировании класса типа `Nybble` в систему типов C#. В частности, объект типа `Nybble` можно свободно комбинировать с данными других типов в арифметических выражениях, поскольку определены преобразования объекта этого типа в тип `int` и обратно. Рассмотрим для примера следующую строку кода из приведенной выше программы.

```
t = a * 2 + 3;
```

В этом выражении переменная t и значения 2 и 3 относятся к типу int, но в ней присутствует также объект типа Nybble. Оба типа оказываются совместимыми благодаря неявному преобразованию типа Nybble в тип int. В данном случае остальная часть выражения относится к типу int, поэтому объект a преобразуется в тип int с помощью своего метода преобразования.

А благодаря преобразованию типа int в тип Nybble значение типа int может быть присвоено объекту типа Nybble. Например, в следующей строке из приведенной выше программы:

```
a = 19;
```

сначала выполняется оператор преобразования типа int в тип Nybble. Затем создается новый объект типа Nybble, в котором сохраняются 4 младших разряда целого значения 19, а по существу, число 3, поскольку значение 19 превышает диапазон представления чисел для типа Nybble. Далее этот объект присваивается переменной экземпляра a. Без операторов преобразования подобные выражения были бы просто недопустимы.

Кроме того, преобразование типа Nybble в тип Nybble используется в цикле for. Без такого преобразования организовать столь простой цикл for было бы просто невозможно.

ПРИМЕЧАНИЕ

В качестве упражнения попробуйте создать вариант полубайтового типа Nybble, предотвращающий переполнение, если присваиваемое значение оказывается за пределами допустимого диапазона чисел. Для этой цели лучше всего сгенерировать исключение. (Подробнее об исключениях — в главе 13.)

---

10 ГЛАВА

Индексаторы и свойства

В этой главе рассматриваются две особые и тесно связанные друг с другом разновидности членов класса: индексаторы и свойства. Каждый из них по-своему расширяет возможности класса, способствуя более полной его интеграции в систему типов C# и повышая его гибкость. В частности, индексаторы предоставляют механизм для индексирования объектов подобно массивам, а свойства — рациональный способ управления доступом к данным экземпляра класса. Эти члены класса тесно связаны друг с другом, поскольку оба опираются на еще одно доступное в C# средство: аксессор.

Индексаторы

Как вам должно быть уже известно, индексирование массива осуществляется с помощью оператора `[]`. Для создаваемых классов можно определить оператор `[]`, но с этой целью вместо операторного метода создается индексатор, который позволяет индексировать объект, подобно массиву. Индексаторы применяются, главным образом, в качестве средства, поддерживающего создание специализированных массивов, на которые накладывается одно или несколько ограничений. Тем не менее индексаторы могут служить практически любым целям, для которых выгодным оказывается такой же синтаксис, как и у массивов. Индексаторы могут быть одно- или многомерными. Рассмотрим сначала одномерные индексаторы.

Создание одномерных индексаторов

Ниже приведена общая форма одномерного индексатора:

```
тип_элемента this[int индекс] {
    // Аксессор для получения данных.
    get {
        // Возврат значения, которое определяет индекс.
    }

    // Аксессор для установки данных.
    set {
       // Установка значения, которое определяет индекс.
    }
}
```

где *тип_элемента* обозначает конкретный тип элемента индексатора. Следовательно, у каждого элемента, доступного с помощью индексатора, должен быть определенный *тип_элемента*. Этот тип соответствует типу элемента массива. Параметр *индекс* получает конкретный индекс элемента, к которому осуществляется доступ. Формально этот параметр совсем не обязательно должен иметь тип `int`, но поскольку индексаторы, как правило, применяются для индексирования массивов, то чаще всего используется целочисленный тип данного параметра.

В теле индексатора определены два аксессора (т.е. средства доступа к данным): `get` и `set`. Аксессор подобен методу, за исключением того, что в нем не объявляется тип возвращаемого значения или параметры. Аксессоры вызываются автоматически при использовании индексатора, и оба получают *индекс* в качестве параметра. Так, если индексатор указывается в левой части оператора присваивания, то вызывается аксессор `set` и устанавливается элемент, на который указывает параметр *индекс*. В противном случае вызывается аксессор `get` и возвращается значение, соответствующее параметру *индекс*. Кроме того, аксессор `set` получает неявный параметр `value`, содержащий значение, присваиваемое по указанному индексу.

Преимущество индексатора заключается, в частности, в том, что он позволяет полностью управлять доступом к массиву, избегая нежелательного доступа. В качестве примера рассмотрим программу, в которой создается класс `FailSoftArray`, реализующий массив для выявления ошибок нарушения границ массива, а следовательно, для предотвращения исключительных ситуаций, возникающих во время выполнения в связи с индексированием массива за его границами. Для этого массив инкапсулируется в качестве закрытого члена класса, а доступ к нему осуществляется только с помощью индексатора. При таком подходе исключается любая попытка получить доступ к массиву за его границами, причем эта попытка пресекается без катастрофических последствий для программы. А поскольку в классе `FailSoftArray` используется индексатор, то к массиву можно обращаться с помощью обычной формы записи.

```
// Использовать индексатор для создания отказоустойчивого массива.

using System;

class FailSoftArray {
  int[] a; // ссылка на базовый массив

  public int Length;   // открытая переменная длины массива
```

```
  public bool ErrFlag; // обозначает результат последней операции

  // Построить массив заданного размера.
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // Это индексатор для класса FailSoftArray.
  public int this[int index] {
    // Это аксессор get.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // Это аксессор set.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
      return false;
    }
  }

// Продемонстрировать применение отказоустойчивого массива.
class FSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Выявить скрытые сбои.
    Console.WriteLine("Скрытый сбой.");
    for(int i=0; i < (fs.Length * 2); i++)
      fs[i] = i*10;

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
```

```
    Console.WriteLine();

   // А теперь показать сбои.
    Console.WriteLine("\nСбой с уведомлением об ошибках.");
    for(int i=0; i < (fs.Length * 2); i++) {
      fs[i] = i*10;
      if(fs.ErrFlag)
        Console.WriteLine("fs[" + i + "] вне границ");
    }

    for(int i=0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(!fs.ErrFlag) Console.Write(x + " ");
      else
        Console.WriteLine("fs[" + i + "] вне границ");
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Скрытый сбой.
0 10 20 30 40 0 0 0 0 0

Сбой с уведомлением об ошибках.
fs[5] вне границ
fs[6] вне границ
fs[7] вне границ
fs[8] вне границ
fs[9] вне границ
0 10 20 30 40 fs[5] вне границ
fs[6] вне границ
fs[7] вне границ
fs[8] вне границ
fs[9] вне границ
```

Индексатор препятствует нарушению границ массива. Внимательно проанализируем каждую часть кода индексатора. Он начинается со следующей строки.

```
public int this[int index] {
```

В этой строке кода объявляется индексатор, оперирующий элементами типа `int`. Ему передается индекс в качестве параметра `index`. Кроме того, индексатор объявляется открытым (`public`), что дает возможность использовать этот индексатор в коде за пределами его класса.

Рассмотрим следующий код аксессора `get`.

```
get {
  if(ok(index)) {
    ErrFlag = false;
    return a[index];
  } else {
    ErrFlag = true;
    return 0;
  }
}
```

Аксессор get предотвращает ошибки нарушения границ массива, проверяя в первую очередь, находится ли индекс в установленных границах. Эта проверка границ выполняется в методе ok(), который возвращает логическое значение true, если индекс правильный, а иначе — логическое значение false. Так, если указанный индекс находится в установленных границах, то по этому индексу возвращается соответствующий элемент. А если индекс оказывается вне установленных границ, то никаких операций не выполняется, но в то же время не возникает никаких ошибок переполнения. В данном варианте класса FailSoftArray переменная ErrFlag содержит результат каждой операции. Ее содержимое может быть проверено после каждой операции на предмет удачного или неудачного выполнения последней. (В главе 13 будет представлен более совершенный способ обработки ошибок с помощью имеющейся в C# подсистемы обработки исключительных ситуаций, а до тех пор можно вполне обойтись установкой и проверкой признака ошибки.)

А теперь рассмотрим следующий код аксессора set, предотвращающего ошибки нарушения границ массива.

```
set {
  if(ok(index)) {
    a[index] = value;
    ErrFlag = false;
  }
   else ErrFlag = true;
}
```

Если параметр index метода ok() находится в установленных пределах, то соответствующему элементу массива присваивается значение, передаваемое из параметра value. В противном случае устанавливается логическое значение true переменной ErrFlag. Напомним, что value в любом аксессорном методе является неявным параметром, содержащим присваиваемое значение. Его не нужно (да и нельзя) объявлять отдельно.

Наличие обоих аксессоров, get и set, в индексаторе не является обязательным. Так, можно создать индексатор только для чтения, реализовав в нем один лишь аксессор get, или же индексатор только для записи с единственным аксессором set.

Перегрузка индексаторов

Индексатор может быть перегружен. В этом случае для выполнения выбирается тот вариант индексатора, в котором точнее соблюдается соответствие его параметра и аргумента, указываемого в качестве индекса. Ниже приведен пример программы, в которой индексатор массива класса FailSoftArray перегружается для индексов типа double. При этом индексатор типа double округляет свой индекс до ближайшего целого значения.

```
// Перегрузить индексатор массива класса FailSoftArray.

using System;

class FailSoftArray {
  int[] a; // ссылка на базовый массив

  public int Length;   // открытая переменная длины массива
```

```
public bool ErrFlag; // обозначает результат последней операции

// Построить массив заданного размера.
public FailSoftArray(int size) {
  a = new int[size];
  Length = size;
}

// Это индексатор типа int для массива FailSoftArray.
public int this[int index] {
  // Это аксессор get.
  get {
    if(ok(index)) {
      ErrFlag = false;
      return a[index];
    } else {
      ErrFlag = true;
      return 0;
    }
  }

  // Это аксессор set.
  set {
    if(ok(index)) {
      a[index] = value;
      ErrFlag = false;
    }
    else ErrFlag = true;
  }
}

/* Это еще один индексатор для массива FailSoftArray.
   Он округляет свой аргумент до ближайшего целого индекса. */
public int this[double idx] {
  // Это аксессор get.
  get {
    int index;

    // Округлить до ближайшего целого.
    if( (idx - (int) idx) < 0.5) index = (int) idx;
    else index = (int) idx + 1;

    if(ok(index)) {
      ErrFlag = false;
      return a[index];
    } else {
      ErrFlag = true;
      return 0;
    }
  }

  // Это аксессор set.
  set {
    int index;
```

```
      // Округлить до ближайшего целого.
      if( (idx - (int) idx) < 0.5) index = (int) idx;
      else index = (int) idx + 1;

      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
      return false;
  }
}

// Продемонстрировать применение отказоустойчивого массива.
class FSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Поместить ряд значений в массив fs.
    for(int i=0; i < fs.Length; i++)
      fs[i] = i;

    // А теперь воспользоваться индексами
    // типа int и double для обращения к массиву.
    Console.WriteLine("fs[1]: " + fs[1]);
    Console.WriteLine("fs[2]: " + fs[2]);

    Console.WriteLine("fs[1.1]: " + fs[1.1]);
    Console.WriteLine("fs[1.6]: " + fs[1.6]);
  }
}
```

При выполнении этой программы получается следующий результат.

```
fs[1]: 1
fs[2]: 2
fs[1.1]: 1
fs[1.6]: 2
```

Как показывает приведенный выше результат, индексы типа `double` округляются до ближайшего целого значения. В частности, индекс `1.1` округляется до 1, а индекс `1.6` — до 2.

Представленный выше пример программы наглядно демонстрирует правомочность перегрузки индексаторов, но на практике она применяется нечасто. Как правило, индексаторы перегружаются для того, чтобы использовать объект определенного класса в качестве индекса, вычисляемого каким-то особым образом.

Индексаторы без базового массива

Следует особо подчеркнуть, что индексатор совсем не обязательно должен оперировать массивом. Его основное назначение — предоставить пользователю функциональные возможности, аналогичные массиву. В качестве примера в приведенной ниже программе демонстрируется индексатор, выполняющий роль массива только для чтения, содержащего степени числа 2 от 0 до 15. Обратите внимание на то, что в этой программе отсутствует конкретный массив. Вместо этого индексатор просто вычисляет подходящее значение для заданного индекса.

```
// Индексаторы совсем не обязательно должны оперировать отдельными
массивами.

using System;

class PwrOfTwo {

  /* Доступ к логическому массиву, содержащему степени
     числа 2 от 0 до 15. */
  public int this[int index] {
    // Вычислить и возвратить степень числа 2.
    get {
      if((index >= 0) && (index < 16)) return pwr(index);
      else return -1;
    }

    // Аксессор set отсутствует.
  }

  int pwr(int p) {
    int result = 1;

    for(int i=0; i < p; i++)
      result *= 2;

    return result;
  }
}

class UsePwrOfTwo {
  static void Main() {
    PwrOfTwo pwr = new PwrOfTwo();

    Console.Write("Первые 8 степеней числа 2: ");
    for(int i=0; i < 8; i++)
      Console.Write(pwr[i] + " ");
    Console.WriteLine();

    Console.Write("А это некоторые ошибки: ");
    Console.Write(pwr[-1] + " " + pwr[17]);

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Первые 8 степеней числа 2: 1 2 4 8 16 32 64 128
А это некоторые ошибки: -1 -1
```

Обратите внимание на то, что в индексатор класса `PwrOfTwo` включен только аксессор `get`, но в нем отсутствует аксессор `set`. Как пояснялось выше, такой индексатор служит только для чтения. Следовательно, объект класса `PwrOfTwo` может указываться только в правой части оператора присваивания, но не в левой его части. Например, попытка ввести следующую строку кода в приведенную выше программу не приведет к желаемому результату.

```
pwr[0] = 11; // не подлежит компиляции
```

Такой оператор присваивания станет причиной появления ошибки во время компиляции, поскольку для индексатора не определен аксессор `set`.

На применение индексаторов накладываются два существенных ограничения. Во-первых, значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра `ref` или `out`, поскольку в индексаторе не определено место в памяти для его хранения. И во-вторых, индексатор должен быть членом своего класса и поэтому не может быть объявлен как `static`.

Многомерные индексаторы

Индексаторы можно создавать и для многомерных массивов. В качестве примера ниже приведен двумерный отказоустойчивый массив. Обратите особое внимание на объявление индексатора в этом примере.

```
// Двумерный отказоустойчивый массив.

using System;

class FailSoftArray2D {
  int[,] a; // ссылка на базовый двумерный массив
  int rows, cols; // размеры массива
  public int Length;   // открытая переменная длины массива
  public bool ErrFlag; // обозначает результат последней операции

  // Построить массив заданных размеров.
  public FailSoftArray2D(int r, int c) {
    rows = r;
    cols = c;
    a = new int[rows, cols];
    Length = rows * cols;
  }

  // Это индексатор для класса FailSoftArray2D.
  public int this[int index1, int index2] {
    // Это аксессор get.
    get {
      if(ok(index1, index2)) {
        ErrFlag = false;
        return a[index1, index2];
      } else {
```

```
        ErrFlag = true;
        return 0;
      }
    }

    // Это аксессор set.
    set {
      if(ok(index1, index2)) {
        a[index1, index2] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Возвратить логическое значение true, если
  // индексы находятся в установленных пределах.
  private bool ok(int index1, int index2) {
    if(index1 >= 0 & index1 < rows &
      index2 >= 0 & index2 < cols)
        return true;

    return false;
  }
}

// Продемонстрировать применение двумерного индексатора.
class TwoDIndexerDemo {
  static void Main() {
    FailSoftArray2D fs = new FailSoftArray2D(3, 5);
    int x;

    // Выявить скрытые сбои.
    Console.WriteLine("Скрытый сбой.");
    for(int i=0; i < 6; i++)
      fs[i, i] = i*10;

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // А теперь показать сбои.
    Console.WriteLine("\nСбой с уведомлением об ошибках.");
    for(int i=0; i < 6; i++) {
      fs[i,i] = i*10;
      if(fs.ErrFlag)
        Console.WriteLine("fs[" + i + ", " + i + "] вне границ");
    }

    for(int i=0; i < 6; i++) {
      x = fs[i,i];
      if(!fs.ErrFlag) Console.Write(x + " ");
```

```
    else
      Console.WriteLine("fs[" + i + ", " + i + "] вне границ");
    }
  }
}
```

Вот к какому результату приводит выполнение этого кода:

```
Скрытый сбой.
0 10 20 0 0 0

Сбой с уведомлением об ошибках.
fs[3, 3] вне границ
fs[4, 4] вне границ
fs[5, 5] вне границ
0 10 20 fs[3, 3] вне границ
fs[4, 4] вне границ
fs[5, 5] вне границ
```

Свойства

Еще одной разновидностью члена класса является *свойство*. Как правило, свойство сочетает в себе поле с методами доступа к нему. Как было показано в приведенных ранее примерах программ, поле зачастую создается, чтобы стать доступным для пользователей объекта, но при этом желательно сохранить управление над операциями, разрешенными для этого поля, например, ограничить диапазон значений, присваиваемых данному полю. Этой цели можно, конечно, добиться и с помощью закрытой переменной, а также методов доступа к ее значению, но свойство предоставляет более совершенный и рациональный путь для достижения той же самой цели.

Свойства очень похожи на индексаторы. В частности, свойство состоит из имени и аксессоров `get` и `set`. Аксессоры служат для получения и установки значения переменной. Главное преимущество свойства заключается в том, что его имя может быть использовано в выражениях и операторах присваивания аналогично имени обычной переменной, но в действительности при обращении к свойству по имени автоматически вызываются его аксессоры `get` и `set`. Аналогичным образом используются аксессоры `get` и `set` индексатора.

Ниже приведена общая форма свойства:

```
тип имя{
  get{
    // код аксессора для чтения из поля
  }

  set{
    // код аксессора для записи в поле
  }
```

где *тип* обозначает конкретный тип свойства, например `int`, а *имя* — присваиваемое свойству имя. Как только свойство будет определено, любое обращение к свойству по имени приведет к автоматическому вызову соответствующего аксессора. Кроме того, аксессор `set` принимает неявный параметр `value`, который содержит значение, присваиваемое свойству.

Следует, однако, иметь в виду, что свойства не определяют место в памяти для хранения полей, а лишь управляют доступом к полям. Это означает, что само свойство не предоставляет поле, и поэтому поле должно быть определено независимо от свойства. (Исключение из этого правила составляет *автоматически реализуемое* свойство, рассматриваемое далее.)

Ниже приведен простой пример программы, в которой определяется свойство `MyProp`, предназначенное для доступа к полю `prop`. В данном примере свойство допускает присваивание только положительных значений.

```
// Простой пример применения свойства.

using System;

class SimpProp {
  int prop; // поле, управляемое свойством MyProp

  public SimpProp() { prop = 0; }
    /* Это свойство обеспечивает доступ к закрытой переменной экземпляра prop.
       Оно допускает присваивание только положительных значений. */
  public int MyProp {
    get {
      return prop;
    }
    set {
      if(value >= 0) prop = value;
    }
  }
}

// Продемонстрировать применение свойства.
class PropertyDemo {
  static void Main() {
    SimpProp ob = new SimpProp();

    Console.WriteLine("Первоначальное значение ob.MyProp: " + ob.MyProp);

    ob.MyProp = 100; // присвоить значение
     Console.WriteLine("Текущее значение ob.MyProp: " + ob.MyProp);

    // Переменной prop нельзя присвоить отрицательное значение.
    Console.WriteLine("Попытка присвоить значение " +
                      "-10 свойству ob.MyProp");
    ob.MyProp = -10;
    Console.WriteLine("Текущее значение ob.MyProp: " + ob.MyProp);
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Первоначальное значение ob.MyProp: 0
Текущее значение ob.MyProp: 100
Попытка присвоить значение -10 свойству ob.MyProp
Текущее значение ob.MyProp: 100
```

Рассмотрим приведенный выше код более подробно. В этом коде определяется одно закрытое поле prop и свойство MyProp, управляющее доступом к полю prop. Как пояснялось выше, само свойство не определяет место в памяти для хранения поля, а только управляет доступом к полю. Кроме того, поле prop является закрытым, а значит, оно доступно *только* через свойство MyProp.

Свойство MyProp указано как public, а следовательно, оно доступно из кода за пределами его класса. И в этом есть своя логика, поскольку данное свойство обеспечивает доступ к полю prop, которое является закрытым. Аксессор get этого свойства просто возвращает значение из поля prop, тогда как аксессор set устанавливает значение в поле prop в том и только в том случае, если это значение оказывается положительным. Таким образом, свойство MyProp контролирует значения, которые могут храниться в поле prop. В этом, собственно, и состоит основное назначение свойств.

Тип свойства MyProp определяется как для чтения, так и для записи, поскольку оно позволяет читать и записывать данные в базовое поле. Тем не менее свойства можно создавать доступными только для чтения или только для записи. Так, если требуется создать свойство, доступное только для чтения, то достаточно определить единственный аксессор get. А если нужно создать свойство, доступное только для записи, то достаточно определить единственный аксессор set.

Воспользуемся свойством для дальнейшего усовершенствования отказоустойчивого массива. Как вам должно быть уже известно, у всех массивов имеется соответствующее свойство длины (Length). До сих пор в классе FailSoftArray для этой цели использовалось открытое целочисленное поле Length. Но это далеко не самый лучший подход, поскольку он допускает установку значений, отличающихся от длины отказоустойчивого массива. (Например, программист, преследующий злонамеренные цели, может умышленно ввести неверное значение в данном поле.) Для того чтобы исправить это положение, превратим поле Length в свойство "только для чтения", как показано в приведенном ниже, измененном варианте класса FailSoftArray.

```
// Добавить свойство Length в класс FailSoftArray.

using System;

class FailSoftArray {
  int[] a; // ссылка на базовый массив
  int len; // длина массива — служит основанием для свойства Length

  public bool ErrFlag; // обозначает результат последней операции

  // Построить массив заданного размера.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Свойство Length только для чтения.
  public int Length {
    get {
      return len;
    }
  }
```

```
  // Это индексатор для класса FailSoftArray.
  public int this[int index] {
    // Это аксессор get.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // Это аксессор set.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
      return false;
    }
  }

// Продемонстрировать применение усовершенствованного
// отказоустойчивого массива.
class ImprovedFSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // Разрешить чтение свойства Length.
    for(int i=0; i < fs.Length; i++)
      fs[i] = i*10;

    for(int i=0; i < fs.Length; i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // fs.Length = 10; // Ошибка, запись запрещена!
  }
}
```

Теперь `Length` — это свойство, в котором местом для хранения данных служит закрытая переменная `len`. А поскольку в этом свойстве определен единственный ак-

сессор get, то оно доступно только для чтения. Это означает, что значение свойства Length можно только читать, но не изменять. Для того чтобы убедиться в этом, попробуйте удалить символы комментария в начале следующей строки из приведенного выше кода.

```
// fs.Length = 10; // Ошибка, запись запрещена!
```

При попытке скомпилировать данный код вы получите сообщение об ошибке, уведомляющее о том, что Length является свойством, доступным только для чтения.

Добавлением свойства Length в класс FailSoftArray усовершенствование рассматриваемого здесь примера кода с помощью свойств далеко не исчерпывается. Еще одним членом данного класса, подходящим для превращения в свойство, служит переменная ErrFlag, поскольку ее применение должно быть ограничено только чтением. Ниже приведен окончательно усовершенствованный вариант класса FailSoftArray, в котором создается свойство Error, использующее в качестве места для хранения данных исходную переменную ErrFlag, ставшую теперь закрытой.

```
// Превратить переменную ErrFlag в свойство.

using System;

class FailSoftArray {
  int[] a; // ссылка на базовый массив
  int len; // длина массива
  bool ErrFlag; // теперь это частная переменная,
                // обозначающая результат последней операции

  // Построить массив заданного размера.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Свойство Length только для чтения.
  public int Length {
    get {
      return len;
    }
  }

  // Свойство Error только для чтения.
  public bool Error {
    get {
      return ErrFlag;
    }
  }

  // Это индексатор для класса FailSoftArray.
  public int this[int index] {
    // Это аксессор get.
    get {
      if(ok(index)) {
        ErrFlag = false;
        return a[index];
```

```
      } else {
        ErrFlag = true;
        return 0;
      }
    }

    // Это аксессор set.
    set {
      if(ok(index)) {
        a[index] = value;
        ErrFlag = false;
      }
      else ErrFlag = true;
    }
  }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
      return false;
    }
  }

// Продемонстрировать применение отказоустойчивого массива.
class FinalFSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Использовать свойство Error.
    for(int i=0; i < fs.Length + 1; i++) {
      fs[i] = i*10;
      if(fs.Error)
        Console.WriteLine("Ошибка в индексе " + i);
    }
  }
}
```

Создание свойства `Error` стало причиной двух следующих изменений в классе `FailSoftArray`. Во-первых, переменная `ErrFlag` была сделана закрытой, поскольку теперь она служит базовым местом хранения данных для свойства `Error`, а следовательно, она не должна быть доступна непосредственно. И во-вторых, было введено свойство `Error` "только для чтения". Теперь свойство `Error` будет опрашиваться в тех программах, где требуется организовать обнаружение ошибок. Именно это и было продемонстрировано выше в методе `Main()`, где намеренно сгенерирована ошибка нарушения границ массива, а для ее обнаружения использовано свойство `Error`.

Автоматически реализуемые свойства

Начиная с версии C# 3.0, появилась возможность для реализации очень простых свойств, не прибегая к явному определению переменной, которой управляет свойство. Вместо этого базовую переменную для свойства автоматически предоставляет компилятор. Такое свойство называется *автоматически реализуемым* и принимает следующую общую форму:

```
тип имя { get; set; }
```

где *тип* обозначает конкретный тип свойства, а *имя* — присваиваемое свойству имя. Обратите внимание на то, что после обозначений аксессоров `get` и `set` сразу же следует точка с запятой, а тело у них отсутствует. Такой синтаксис предписывает компилятору создать автоматически переменную, иногда еще называемую *поддерживающим полем*, для хранения значения. Такая переменная недоступна непосредственно и не имеет имени. Но в то же время она может быть доступна через свойство.

Ниже приведен пример объявления свойства, автоматически реализуемого под именем `UserCount`.

```
public int UserCount { get; set; }
```

Как видите, в этой строке кода переменная явно не объявляется. И как пояснялось выше, компилятор автоматически создает анонимное поле, в котором хранится значение. А в остальном автоматически реализуемое свойство `UserCount` подобно всем остальным свойствам.

Но в отличие от обычных свойств автоматически реализуемое свойство не может быть доступным только для чтения или только для записи. При объявлении этого свойства в любом случае необходимо указывать оба аксессора — `get` и `set`. Хотя добиться желаемого (т.е. сделать автоматически реализуемое свойство доступным только для чтения или только для записи) все же можно, объявив ненужный аксессор как `private` (подробнее об этом — в разделе "Применение модификаторов доступа в аксессорах").

Несмотря на очевидные удобства автоматически реализуемых свойств, их применение ограничивается в основном теми ситуациями, в которых не требуется управление установкой или получением значений из поддерживающих полей. Напомним, что поддерживающее поле недоступно напрямую. Это означает, что на значение, которое может иметь автоматически реализуемое свойство, нельзя наложить никаких ограничений. Следовательно, имена автоматически реализуемых свойств просто заменяют собой имена самих полей, а зачастую именно это и требуется в программе. Автоматически реализуемые свойства могут оказаться полезными и в тех случаях, когда с помощью свойств функциональные возможности программы открываются для сторонних пользователей, и для этой цели могут даже применяться специальные средства проектирования.

Применение инициализаторов объектов в свойствах

Как пояснялось в главе 8, *инициализатор объекта* применяется в качестве альтернативы явному вызову конструктора при создании объекта. С помощью инициализаторов объектов задаются начальные значения полей или свойств, которые требуется инициализировать. При этом синтаксис инициализаторов объектов оказывается одинаковым как для свойств, так и для полей. В качестве примера ниже приведена программа из главы 8, измененная с целью продемонстрировать применение инициализаторов объектов в свойствах. Напомним, что в версии этой программы из главы 8 использовались поля, а приведенная ниже версия отличается лишь тем, что в ней поля `Count` и `Str` превращены в свойства. В то же время синтаксис инициализаторов объектов не изменился.

```
// Применить инициализаторы объектов в свойствах.

using System;
```

```
class MyClass {
  // Теперь это свойства.
  public int Count { get; set; }
  public string Str { get; set; }
}

class ObjInitDemo {
  static void Main() {
    // Сконструировать объект типа MyClass с помощью инициализаторов
объектов.
    MyClass obj =
      new MyClass { Count = 100, Str = "Тестирование" };

    Console.WriteLine(obj.Count + " " + obj.Str);
  }
}
```

Как видите, свойства `Count` и `Str` устанавливаются в выражениях с инициализатором объекта. Приведенная выше программа дает такой же результат, как и программа из главы 8, а именно:

```
100 Тестирование
```

Как пояснялось в главе 8, синтаксис инициализатора объекта оказывается наиболее пригодным для работы с анонимными типами, формируемыми в LINQ-выражениях. А в остальных случаях чаще всего используется синтаксис обычных конструкторов.

Ограничения, присущие свойствам

Свойствам присущ ряд существенных ограничений. Во-первых, свойство не определяет место для хранения данных, и поэтому не может быть передано методу в качестве параметра `ref` или `out`. Во-вторых, свойство не подлежит перегрузке. Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило. И наконец, свойство не должно изменять состояние базовой переменной при вызове аксессора `get`. И хотя это ограничительное правило не соблюдается компилятором, его нарушение считается семантической ошибкой. Действие аксессора `get` не должно носить характер вмешательства в функционирование переменной.

Применение модификаторов доступа в аксессорах

По умолчанию доступность аксессоров `set` и `get` оказывается такой же, как и у индексатора и свойства, частью которых они являются. Так, если свойство объявляется как `public`, то по умолчанию его аксессоры `set` и `get` также становятся открытыми (`public`). Тем не менее для аксессора `set` или `get` можно указать собственный модификатор доступа, например `private`. Но в любом случае доступность аксессора, определяемая таким модификатором, должна быть более ограниченной, чем доступность, указываемая для его свойства или индексатора.

Существует целый ряд причин, по которым требуется ограничить доступность аксессора. Допустим, что требуется предоставить свободный доступ к значению свойства,

но вместе с тем дать возможность устанавливать это свойство только членам его класса. Для этого достаточно объявить аксессор данного свойства как `private`. В приведенном ниже примере используется свойство `MyProp`, аксессор `set` которого указан как `private`.

```
// Применить модификатор доступа в аксессоре.

using System;

class PropAccess {
  int prop; // поле, управляемое свойством MyProp

  public PropAccess() { prop = 0; }
  /* Это свойство обеспечивает доступ к закрытой переменной экземпляра prop.
     Оно разрешает получать значение переменной prop из любого кода,
     но устанавливать его — только членам своего класса. */
  public int MyProp {
    get {
      return prop;
    }

    private set { // теперь это закрытый аксессор
      prop = value;
    }
  }

  // Этот член класса инкрементирует значение свойства MyProp.
  public void IncrProp() {
    MyProp++; // Допускается в том же самом классе.
  }
}

// Продемонстрировать применение модификатора доступа в аксессоре свойства.
class PropAccessDemo {
  static void Main() {
    PropAccess ob = new PropAccess();

    Console.WriteLine("Первоначальное значение ob.MyProp: " + ob.MyProp);

//    ob.MyProp = 100; // недоступно для установки

    ob.IncrProp();
    Console.WriteLine("Значение ob.MyProp после инкрементирования: " +
                      ob.MyProp);
  }
}
```

В классе `PropAccess` аксессор `set` указан как `private`. Это означает, что он доступен только другим членам данного класса, например методу `IncrProp()`, но недоступен для кода за пределами класса `PropAccess`. Именно поэтому попытка присвоить свойству `ob.MyProp` значение в классе `PropAccessDemo` закомментирована.

Вероятно, ограничение доступа к аксессорам оказывается наиболее важным для работы с автоматически реализуемыми свойствами. Как пояснялось выше, создать

автоматически реализуемое свойство только для чтения или же только для записи нельзя, поскольку оба аксессора, get и set, должны быть указаны при объявлении такого свойства. Тем не менее добиться желаемого результата все же можно, объявив один из аксессоров автоматически реализуемого свойства как private. В качестве примера ниже приведено объявление автоматически реализуемого свойства Length для класса FailSoftArray, которое фактически становится доступным только для чтения.

```
public int Length { get; private set; }
```

Свойство Length может быть установлено только из кода в его классе, поскольку его аксессор set объявлен как private. А изменять свойство Length за пределами его класса не разрешается. Это означает, что за пределами своего класса свойство, по существу, оказывается доступным только для чтения. Аналогичным образом можно объявить и свойство Error, как показано ниже.

```
public bool Error { get; private set; }
```

Благодаря этому свойство Error становится доступным для чтения, но не для установки за пределами класса FailSoftArray.

Для опробования автоматически реализуемых вариантов свойств Length и Error в классе FailSoftArray удалим сначала переменные len и ErrFlag, поскольку они больше не нужны, а затем заменим каждое применение переменных len и ErrFlag свойствами Length и Error в классе FailSoftArray. Ниже приведен обновленный вариант класса FailSoftArray вместе с методом Main(), демонстрирующим его применение.

```
// Применить автоматически реализуемые и доступные
// только для чтения свойства Length и Error.

using System;

class FailSoftArray {
  int[] a; // ссылка на базовый массив

  // Построить массив по заданному размеру.
  public FailSoftArray(int size) {
  a = new int[size];
  Length = size;
  }

  // Автоматически реализуемое и доступное только для чтения свойство Length.
  public int Length { get; private set; }

  // Автоматически реализуемое и доступное только для чтения свойство Error.
  public bool Error { get; private set; }

  // Это индексатор для массива FailSoftArray.
  public int this[int index] {
    // Это аксессор get.
    get {
```

```
      if(ok(index)) {
        Error = false;
        return a[index];
      } else {
        Error = true;
        return 0;
      }
    }

    // Это аксессор set.
    set {
      if(ok(index)) {
        a[index] = value;
        Error = false;
      }
      else Error = true;
    }
  }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
      return false;
  }
}

// Продемонстрировать применение усовершенствованного
// отказоустойчивого массива.
class FinalFSDemo {
  static void Main() {
    FailSoftArray fs = new FailSoftArray(5);

    // Использовать свойство Error.
    for(int i=0; i < fs.Length + 1; i++) {
      fs[i] = i*10;
      if(fs.Error)
        Console.WriteLine("Ошибка в индексе " + i);
    }
  }
}
```

Этот вариант класса `FailSoftArray` действует таким же образом, как и предыдущий, но в нем отсутствуют поддерживающие поля, объявляемые явно.

На применение модификаторов доступа в аксессорах накладываются следующие ограничения. Во-первых, действию модификатора доступа подлежит только один аксессор: `set` или `get`, но не оба сразу. Во-вторых, модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора. И наконец, модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе. (Подробнее об интерфейсах речь пойдет в главе 12.)

Применение индексаторов и свойств

В предыдущих примерах программ был продемонстрирован основной принцип действия индексаторов и свойств, но их возможности не были раскрыты в полную силу. Поэтому в завершение этой главы обратимся к примеру класса `RangeArray`, в котором индексаторы и свойства используются для создания типа массива с пределами индексирования, определяемыми пользователем.

Как вам должно быть уже известно, индексирование всех массивов в C# начинается с нуля. Но в некоторых приложениях индексирование массива удобнее начинать с любой произвольной точки отсчета: с 1 или даже с отрицательного числа, например от –5 и до 5. Рассматриваемый здесь класс `RangeArray` разработан таким образом, чтобы допускать подобного рода индексирование массивов.

Используя класс `RangeArray`, можно написать следующий фрагмент кода.

```
RangeArray ra = new RangeArray(-5, 10); // массив с индексами от -5 до 10
for(int i=-5; i <= 10; i++) ra[i] = i; // индексирование массива от -5 до
10
```

Нетрудно догадаться, что в первой строке этого кода конструируется объект класса `RangeArray` с пределами индексирования массива от –5 до 10 включительно. Первый аргумент обозначает начальный индекс, а второй — конечный индекс. Как только объект `ra` будет сконструирован, он может быть проиндексирован как массив в пределах от –5 до 10.

Ниже приведен полностью класс `RangeArray` вместе с классом `RangeArrayDemo`, в котором демонстрируется индексирование массива в заданных пределах. Класс `RangeArray` реализован таким образом, чтобы поддерживать массивы типа `int`, но при желании вы можете изменить этот тип на любой другой.

```
/* Создать класс со специально указываемыми пределами индексирования массива.
   Класс RangeArray допускает индексирование массива с любого значения, а не
   только с нуля. При создании объекта класса RangeArray указываются начальный
   и конечный индексы. Допускается также указывать отрицательные индексы.
   Например, можно создать массивы, индексируемые от -5 до 5, от 1 до 10
   или же от 50 до 56. */

using System;

class RangeArray {
  // Закрытые данные.
  int[] a; // ссылка на базовый массив
  int lowerBound; // наименьший индекс
  int upperBound; // наибольший индекс

  // Автоматически реализуемое и доступное только для чтения свойство Length.
public int Length { get; private set; }

  // Автоматически реализуемое и доступное только для чтения свойство Error.
  public bool Error { get; private set; }

  // Построить массив по заданному размеру.
  public RangeArray(int low, int high) {
    high++;
```

```
    if(high <= low) {
      Console.WriteLine("Неверные индексы");
      high = 1; // создать для надежности минимально допустимый массив
      low = 0;
    }
    a = new int[high - low];
    Length = high - low;

    lowerBound = low;
    upperBound = --high;
  }

  // Это индексатор для класса RangeArray.
  public int this[int index] {
    // Это аксессор get.
    get {
      if(ok(index)) {
        Error = false;
        return a[index - lowerBound];
      } else {
        Error = true;
        return 0;
      }
    }

    // Это аксессор set.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
        Error = false;
      }
      else Error = true;
    }
  }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
    return false;
  }
}

// Продемонстрировать применение массива с произвольно
// задаваемыми пределами индексирования.
class RangeArrayDemo {
  static void Main() {
    RangeArray ra = new RangeArray(-5, 5);
    RangeArray ra2 = new RangeArray(1, 10);
    RangeArray ra3 = new RangeArray(-20, -12);

    // Использовать объект ra в качестве массива.
    Console.WriteLine("Длина массива ra: " + ra.Length);
    for(int i = -5; i <= 5; i++)
```

```
      ra[i] = i;

    Console.Write("Содержимое массива ra: ");
    for(int i = -5; i <= 5; i++)
      Console.Write(ra[i] + " ");

    Console.WriteLine("\n");

    // Использовать объект ra2 в качестве массива.
    Console.WriteLine("Длина массива ra2: " + ra2.Length);
    for(int i = 1; i <= 10; i++)
      ra2[i] = i;

    Console.Write("Содержимое массива ra2: ");
    for(int i = 1; i <= 10; i++)
      Console.Write(ra2[i] + " ");

    Console.WriteLine("\n");

    // Использовать объект ra3 в качестве массива.
    Console.WriteLine("Длина массива ra3: " + ra3.Length);
    for(int i = -20; i <= -12; i++)
      ra3[i] = i;

    Console.Write("Содержимое массива ra3: ");
    for(int i = -20; i <= -12; i++)
      Console.Write(ra3[i] + " ");

    Console.WriteLine("\n");
  }
}
```

При выполнении этого кода получается следующий результат.

```
Длина массива ra: 11
Содержимое массива ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Длина массива ra2: 10
Содержимое массива ra2: 1 2 3 4 5 6 7 8 9 10

Длина массива ra3: 9
Содержимое массива ra3: -20 -19 -18 -17 -16 -15 -14 -13 -12
```

Как следует из результата выполнения приведенного выше кода, объекты типа `RangeArray` можно индексировать в качестве массивов, начиная с любой точки отсчета, а не только с нуля. Рассмотрим подробнее саму реализацию класса `RangeArray`.

В начале класса `RangeArray` объявляются следующие закрытые переменные экземпляра.

```
// Закрытые данные.
int[] a; // ссылка на базовый массив
int lowerBound; // наименьший индекс
int upperBound; // наибольший индекс
```

Переменная a служит для обращения к базовому массиву по ссылке. Память для него распределяется конструктором класса RangeArray. Нижняя граница индексирования массива хранится в переменной lowerBound, а верхняя граница — в переменной upperBound.

Далее объявляются автоматически реализуемые свойства Length и Error.

```
// Автоматически реализуемое и доступное только для чтения свойство Length.
public int Length { get; private set; }

// Автоматически реализуемое и доступное только для чтения свойство Error.
public bool Error { get; private set; }
```

Обратите внимание на то, что в обоих свойства аксессор set обозначен как private. Как пояснялось выше, такое объявление автоматически реализуемого свойства, по существу, делает его доступным только для чтения.

Ниже приведен конструктор класса RangeArray.

```
// Построить массив по заданному размеру.
public RangeArray(int low, int high) {
  high++;
  if(high <= low) {
    Console.WriteLine("Неверные индексы");
    high = 1; // создать для надежности минимально допустимый массив
    low = 0;
  }
  a = new int[high - low];
  Length = high - low;

  lowerBound = low;
  upperBound = --high;
}
```

При конструировании объекту класса RangeArray передается нижняя граница массива в качестве параметра low, а верхняя граница — в качестве параметра high. Затем значение параметра high инкрементируется, поскольку пределы индексирования массива изменяются от low до high включительно. Далее выполняется следующая проверка: является ли верхний индекс больше нижнего индекса. Если это не так, то выдается сообщение об ошибке и создается массив, состоящий из одного элемента. После этого для массива распределяется память, а ссылка на него присваивается переменной a. Затем свойство Length устанавливается равным числу элементов массива. И наконец, устанавливаются переменные lowerBound и upperBound.

Далее в классе RangeArray реализуется его индексатор, как показано ниже.

```
// Это индексатор для класса RangeArray.
public int this[int index] {
  // Это аксессор get.
  get {
    if(ok(index)) {
      Error = false;
      return a[index - lowerBound];
    } else {
      Error = true;
      return 0;
  }
```

```
}

// Это аксессор set.
  set {
    if(ok(index)) {
      a[index - lowerBound] = value;
      Error = false;
    }
    else Error = true;
  }
}
```

Этот индексатор подобен тому, что использовался в классе `FailSoftArray`, за одним существенным исключением. Обратите внимание на следующее выражение, в котором индексируется массив `a`.

```
index - lowerBound
```

В этом выражении индекс, передаваемый в качестве параметра `index`, преобразуется в индекс с отсчетом от нуля, пригодный для индексирования массива `a`. Данное выражение действует при любом значении переменной `lowerBound`: положительном, отрицательном или нулевом.

Ниже приведен метод `ok()`.

```
// Возвратить логическое значение true, если
// индекс находится в установленных границах.
private bool ok(int index) {
  if(index >= lowerBound & index <= upperBound) return true;
  return false;
}
```

Этот метод аналогичен использовавшемуся в классе `FailSoftArray`, за исключением того, что в нем контроль границ массива осуществляется по значениям переменных `lowerBound` и `upperBound`.

Класс `RangeArray` демонстрирует лишь одну разновидность специализированного массива, который может быть создан с помощью индексаторов и свойств. Существуют, конечно, и другие. Аналогичным образом можно, например, создать динамические массивы, которые расширяются или сужаются по мере надобности, ассоциативные и разреженные массивы. Попробуйте создать один из таких массивов в качестве упражнения.

11 ГЛАВА
Наследование

Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В языке C# класс, который наследуется, называется *базовым*, а класс, который наследует, — *производным*. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

Основы наследования

Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается базовый класс. Рассмотрим для начала простой пример. Ниже приведен класс `TwoDShape`, содержащий ширину и высоту двухмерного объекта, например квадрата, прямоугольника, треугольника и т.д.

```
// Класс для двумерных объектов.
class TwoDShape {
  public double Width;
```

```
  public double Height;

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}
```

Класс `TwoDShape` может стать базовым, т.е. отправной точкой для создания классов, описывающих конкретные типы двумерных объектов. Например, в приведенной ниже программе класс `TwoDShape` служит для порождения производного класса `Triangle`. Обратите особое внимание на объявление класса `Triangle`.

```
// Пример простой иерархии классов.

using System;

// Класс для двумерных объектов.
class TwoDShape {
  public double Width;
  public double Height;
  public void ShowDim() {

    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}

// Класс Triangle, производный от класса TwoDShape.
class Triangle : TwoDShape {
  public string Style; // тип треугольника

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

class Shapes {
  static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.Width = 4.0;
    t1.Height = 4.0;
    t1.Style = "равнобедренный";

    t2.Width = 8.0;
    t2.Height = 12.0;
    t2.Style = "прямоугольный";

    Console.WriteLine("Сведения об объекте t1: ");
```

```
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Площадь равна " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Площадь равна " + t2.Area());
  }
}
```

При выполнении этой программы получается следующий результат.

```
Сведения об объекте t1:
Треугольник равнобедренный
Ширина и высота равны 4 и 4
Площадь равна 8

Сведения об объекте t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48
```

В классе `Triangle` создается особый тип объекта класса `TwoDShape` (в данном случае — треугольник). Кроме того, в класс `Triangle` входят все члены класса `TwoDShape`, к которым, в частности, добавляются методы `Area()` и `ShowStyle()`. Так, описание типа треугольника сохраняется в переменной `Style`, метод `Area()` рассчитывает и возвращает площадь треугольника, а метод `ShowStyle()` отображает тип треугольника.

Обратите внимание на синтаксис, используемый в классе `Triangle` для наследования класса `TwoDShape`.

```
class Triangle : TwoDShape {
```

Этот синтаксис может быть обобщен. Всякий раз, когда один класс наследует от другого, после имени базового класса указывается имя производного класса, отделяемое двоеточием. В C# синтаксис наследования класса удивительно прост и удобен в использовании.

В класс `Triangle` входят все члены его базового класса `TwoDShape`, и поэтому в нем переменные `Width` и `Height` доступны для метода `Area()`. Кроме того, объекты `t1` и `t2` в методе `Main()` могут обращаться непосредственно к переменным `Width` и `Height`, как будто они являются членами класса `Triangle`. На рис. 11.1 схематически показано, каким образом класс `TwoDShape` вводится в класс `Triangle`.

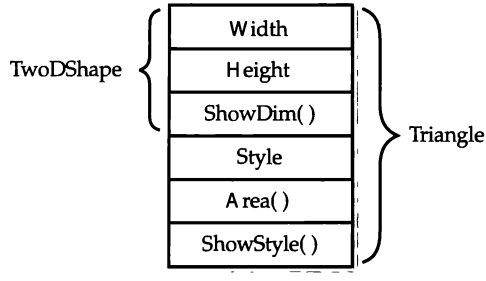


Рис. 11.1. Схематическое представление класса `Triangle`

Несмотря на то что класс TwoDShape является базовым для класса Triangle, в то же время он представляет собой совершенно независимый и самодостаточный класс. Если класс служит базовым для производного класса, то это совсем не означает, что он не может быть использован самостоятельно. Например, следующий фрагмент кода считается вполне допустимым.

```
TwoDShape shape = new TwoDShape();
shape.Width = 10;
shape.Height = 20;

shape.ShowDim();
```

Разумеется, объект класса TwoDShape никак не связан с любым из классов, производных от класса TwoDShape, и вообще не имеет к ним доступа.

Ниже приведена общая форма объявления класса, наследующего от базового класса.

```
class имя_производного_класса : имя_базового_класса {
  // тело класса
}
```

Для любого производного класса можно указать только один базовый класс. В C# не предусмотрено наследование нескольких базовых классов в одном производном классе. (В этом отношении C# отличается от C++, где допускается наследование нескольких базовых классов. Данное обстоятельство следует принимать во внимание при переносе кода C++ в C#.) Тем не менее можно создать иерархию наследования, в которой производный класс становится базовым для другого производного класса. (Разумеется, ни один из классов не может быть базовым для самого себя как непосредственно, так и косвенно.) Но в любом случае производный класс наследует все члены своего базового класса, в том числе переменные экземпляра, методы, свойства и индексаторы.

Главное преимущество наследования заключается в следующем: как только будет создан базовый класс, в котором определены общие для множества объектов атрибуты, он может быть использован для создания любого числа более конкретных производных классов. А в каждом производном классе может быть точно выстроена своя собственная классификация. В качестве примера ниже приведен еще один класс, производный от класса TwoDShape и инкапсулирующий прямоугольники.

```
// Класс для прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape {
  // Возвратить логическое значение true, если
  // прямоугольник является квадратом.
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }

  // Возвратить площадь прямоугольника.
  public double Area() {
    return Width * Height;
  }
}
```

В класс Rectangle входят все члены класса TwoDShape, к которым добавлен метод IsSquare(), определяющий, является ли прямоугольник квадратом, а также метод Area(), вычисляющий площадь прямоугольника.

Доступ к членам класса и наследование

Как пояснялось в главе 8, члены класса зачастую объявляются закрытыми, чтобы исключить их несанкционированное или незаконное использование. Но наследование класса не отменяет ограничения, накладываемые на доступ к закрытым членам класса. Поэтому если в производный класс и входят все члены его базового класса, в нем все равно оказываются недоступными те члены базового класса, которые являются закрытыми. Так, если сделать закрытыми переменные класса `TwoDShape`, они станут недоступными в классе `Triangle`, как показано ниже.

```
// Доступ к закрытым членам класса не наследуется.
// Этот пример кода не подлежит компиляции.

using System;

// Класс для двумерных объектов.
class TwoDShape {
  double Width;  // теперь это закрытая переменная
  double Height; // теперь это закрытая переменная

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}

// Класс Triangle, производный от класса TwoDShape.
class Triangle : TwoDShape {
  public string Style; // тип треугольника

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2; // Ошибка, доступ к закрытому
                               // члену класса запрещен
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}
```

Класс `Triangle` не будет компилироваться, потому что обращаться к переменным `Width` и `Height` из метода `Area()` запрещено. А поскольку переменные `Width` и `Height` теперь являются закрытыми, то они доступны только для других членов своего класса, но не для членов производных классов.

ПРИМЕЧАНИЕ

Закрытый член класса остается закрытым в своем классе. Он не доступен из кода за пределами своего класса, включая и производные классы.

На первый взгляд, ограничение на доступ к частным членам базового класса из производного класса кажется трудно преодолимым, поскольку оно не дает во многих случаях возможности пользоваться частными членами этого класса. Но на самом деле это не так. Для преодоления данного ограничения в C# предусмотрены разные способы. Один из них состоит в использовании защищенных (`protected`) членов класса, рассматриваемых в следующем разделе, а второй — в применении открытых свойств для доступа к закрытым данным.

Как пояснялось в предыдущей главе, свойство позволяет управлять доступом к переменной экземпляра. Например, с помощью свойства можно ввести ограничения на доступ к значению переменной или же сделать ее доступной только для чтения. Так, если сделать свойство открытым, но объявить его базовую переменную закрытой, то этим свойством можно будет воспользоваться в производном классе, но нельзя будет получить непосредственный доступ к его базовой закрытой переменной.

Ниже приведен вариант класса `TwoDShape`, в котором переменные `Width` и `Height` превращены в свойства. По ходу дела в этом классе выполняется проверка: являются ли положительными значения свойств `Width` и `Height`. Это дает, например, возможность указывать свойства `Width` и `Height` в качестве координат формы в любом квадранте прямоугольной системы координат, не получая заранее их абсолютные значения.

```
// Использовать открытые свойства для установки и
// получения значений закрытых членов класса.

using System;

// Класс для двумерных объектов.
class TwoDShape {
  double pri_width;  // теперь это закрытая переменная
  double pri_height; // теперь это закрытая переменная

  // Свойства ширины и высоты двумерного объекта.
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}

// Класс для треугольников, производный от
// класса TwoDShape.
class Triangle : TwoDShape {
  public string Style; // тип треугольника
```

```
  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

class Shapes2 {
  static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.Width = 4.0;
    t1.Height = 4.0;
    t1.Style = "равнобедренный";

    t2.Width = 8.0;
    t2.Height = 12.0;
    t2.Style = "прямоугольный";

    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Площадь равна " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Площадь равна " + t2.Area());
  }
}
```

В этом варианте свойства `Width` и `Height` предоставляют доступ к закрытым членам `pri_width` и `pri_height` класса `TwoDShape`, в которых фактически хранятся значения ширины и высоты двумерного объекта. Следовательно, значения членов `pri_width` и `pri_height` класса `TwoDShape` могут быть установлены и получены с помощью соответствующих открытых свойств, несмотря на то, что сами эти члены по-прежнему остаются закрытыми.

Базовый и производный классы иногда еще называют *суперклассом* и *подклассом* соответственно. Эти термины происходят из практики программирования на Java. То, что в Java называется суперклассом, в C# обозначается как базовый класс. А то, что в Java называется подклассом, в C# обозначается как производный класс. Оба ряда терминов часто применяются к классу в обоих языках программирования, но в этой книге по-прежнему употребляются общепринятые в C# термины базового и производного классов, которые принято употреблять и в C++.

Организация защищенного доступа

Как пояснялось выше, открытый член базового класса недоступен для производного класса. Из этого можно предположить, что для доступа к некоторому члену базового класса из производного класса этот член необходимо сделать открытым. Но если сделать член класса открытым, то он станет доступным для всего кода, что далеко не всегда желательно. Правда, упомянутое предположение верно лишь отчасти, поскольку в C# допускается создание *защищенного* члена класса. Защищенный член является открытым в пределах иерархии классов, но закрытым за пределами этой иерархии.

Защищенный член создается с помощью модификатора доступа `protected`. Если член класса объявляется как `protected`, он становится закрытым, но за исключением одного случая, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а значит, доступным для производного класса. Таким образом, используя модификатор доступа `protected`, можно создать члены класса, являющиеся закрытыми для своего класса, но все же наследуемыми и доступными для производного класса.

Ниже приведен простой пример применения модификатора доступа `protected`.

```
// Продемонстрировать применение модификатора доступа protected.

using System;

class B {
  protected int i, j; // члены, закрытые для класса B,
                      // но доступные для класса D
  public void Set(int a, int b) {
    i = a;
    j = b;
  }

  public void Show() {
    Console.WriteLine(i + " " + j);
  }
}

class D : B {
  int k; // закрытый член

  // члены i и j класса B доступны для класса D
  public void Setk() {
    k = i * j;
  }

  public void Showk() {
    Console.WriteLine(k);
  }
}

class ProtectedDemo {
  static void Main() {
    D ob = new D();
    ob.Set(2, 3); // допустимо, поскольку доступно для класса D
```

```
    ob.Show(); // допустимо, поскольку доступно для класса D
    ob.Setk();  // допустимо, поскольку входит в класс D
    ob.Showk(); // допустимо, поскольку входит в класс D
  }
}
```

В данном примере класс `B` наследуется классом `D`, а его члены `i` и `j` объявлены как `protected`, и поэтому они доступны для метода `Setk()`. Если бы члены `i` и `j` класса `B` были объявлены как `private`, то они оказались бы недоступными для класса `D`, и приведенный выше код нельзя было бы скомпилировать.

Аналогично состоянию `public` и `private`, состояние `protected` сохраняется за членом класса независимо от количества уровней наследования. Поэтому когда производный класс используется в качестве базового для другого производного класса, любой защищенный член исходного базового класса, наследуемый первым производным классом, наследуется как защищенный и вторым производным классом.

Несмотря на всю свою полезность, защищенный доступ пригоден далеко не для всех ситуаций. Так, в классе `TwoDShape` из приведенного ранее примера требовалось, чтобы значения его членов `Width` и `Height` были доступными открыто, поскольку нужно было управлять значениями, которые им присваивались, что было бы невозможно, если бы они были объявлены как `protected`. В данном случае более подходящим решением оказалось применение свойств, чтобы управлять доступом, а не предотвращать его. Таким образом, модификатор доступа `protected` следует применять в том случае, если требуется создать член класса, доступный для всей иерархии классов, но для остального кода он должен быть закрытым. А для управления доступом к значению члена класса лучше воспользоваться свойством.

Конструкторы и наследование

В иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы. В связи с этим возникает следующий резонный вопрос: какой конструктор отвечает за построение объекта производного класса: конструктор базового класса, конструктор производного класса или же оба? На этот вопрос можно ответить так: конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта. И в этом есть своя логика, поскольку базовому классу неизвестны и недоступны любые элементы производного класса, а значит, их конструирование должно происходить раздельно. В приведенных выше примерах данный вопрос не возникал, поскольку они опирались на автоматическое создание конструкторов, используемых в C# по умолчанию. Но на практике конструкторы определяются в большинстве классов. Ниже будет показано, каким образом разрешается подобная ситуация.

Если конструктор определен только в производном классе, то все происходит очень просто: конструируется объект производного класса, а базовая часть объекта автоматически конструируется его конструктором, используемым по умолчанию. В качестве примера ниже приведен переработанный вариант класса `Triangle`, в котором определяется конструктор, а член `Style` делается закрытым, так как теперь он устанавливается конструктором.

```
// Добавить конструктор в класс Triangle.

using System;
```

```
// Класс для двумерных объектов.
class TwoDShape {
  double pri_width;
  double pri_height;

  // Свойства ширины и длины объекта.
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Ширина и длина равны " +
                      Width + " и " + Height);
  }
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style;

  // Конструктор.
  public Triangle(string s, double w, double h) {
    Width = w;  // инициализировать член базового класса
    Height = h; // инициализировать член базового класса
    Style = s; // инициализировать член производного класса
  }

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

class Shapes3 {
  static void Main() {
    Triangle t1 = new Triangle("равнобедренный", 4.0, 4.0);
    Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);

    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Площадь равна " + t1.Area());
```

```
    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Площадь равна " + t2.Area());
  }
}
```

В данном примере конструктор класса `Triangle` инициализирует наследуемые члены класса `TwoDShape` вместе с его собственным полем `Style`.

Когда конструкторы определяются как в базовом, так и в производном классе, процесс построения объекта несколько усложняется, поскольку должны выполняться конструкторы обоих классов. В данном случае приходится обращаться к еще одному ключевому слову языка C#: `base`, которое находит двоякое применение: во-первых, для вызова конструктора базового класса; и во-вторых, для доступа к члену базового класса, скрывающегося за членом производного класса. Ниже будет рассмотрено первое применение ключевого слова `base`.

Вызов конструкторов базового класса

С помощью формы расширенного объявления конструктора производного класса и ключевого слова `base` в производном классе может быть вызван конструктор, определенный в его базовом классе. Ниже приведена общая форма этого расширенного объявления:

```
конструктор_производного_класса(список_параметров) : base(список_аргументов) {
  // тело конструктора
}
```

где *список_аргументов* обозначает любые аргументы, необходимые конструктору в базовом классе. Обратите внимание на местоположение двоеточия.

Для того чтобы продемонстрировать применение ключевого слова `base` на конкретном примере, рассмотрим еще один вариант класса `TwoDShape` в приведенной ниже программе. В данном примере определяется конструктор, инициализирующий свойства `Width` и `Height`. Затем этот конструктор вызывается конструктором класса `Triangle`.

```
// Добавить конструктор в класс TwoDShape.

using System;

// Класс для двумерных объектов.
class TwoDShape {
  double pri_width;
  double pri_height;

  // Конструктор класса TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Свойства ширины и высоты объекта.
```

```
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style;

  // Вызвать конструктор базового класса.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

class Shapes4 {
  static void Main() {
    Triangle t1 = new Triangle("равнобедренный", 4.0, 4.0);
    Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Площадь равна " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Площадь равна " + t2.Area());
  }
}
```

Теперь конструктор класса `Triangle` объявляется следующим образом.

```
public Triangle(
       string s, double w, double h) : base(w, h) {
```

В данном варианте конструктор `Triangle()` вызывает метод `base` с параметрами `w` и `h`. Это, в свою очередь, приводит к вызову конструктора `TwoDShape()`, инициализирующего свойства `Width` и `Height` значениями параметров `w` и `h`. Они больше не инициализируются средствами самого класса `Triangle`, где теперь остается инициализировать только его собственный член `Style`, определяющий тип треугольника. Благодаря этому класс `TwoDShape` высвобождается для конструирования своего подобъекта любым избранным способом. Более того, в класс `TwoDShape` можно ввести функции, о которых даже не будут подозревать производные классы, что предотвращает нарушение существующего кода.

С помощью ключевого слова `base` можно вызвать конструктор любой формы, определяемой в базовом классе, причем выполняться будет лишь тот конструктор, параметры которого соответствуют переданным аргументам. В качестве примера ниже приведены расширенные варианты классов `TwoDShape` и `Triangle`, в которые включены как используемые по умолчанию конструкторы, так и конструкторы, принимающие один аргумент.

```
// Добавить дополнительные конструкторы в класс TwoDShape.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Конструктор, вызываемый по умолчанию.
  public TwoDShape() {
    Width = Height = 0.0;
  }

  // Конструктор класса TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Сконструировать объект равной ширины и высоты.
  public TwoDShape(double x) {
    Width = Height = x;
  }

  // Свойства ширины и высоты объекта.
  public double Width {
    get { return pri_width; }
      set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
```

```
    set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style;

  /* Конструктор, используемый по умолчанию.
     Автоматически вызывает конструктор, доступный по
     умолчанию в классе TwoDShape. */
  public Triangle() {
    Style = "null";
  }

  // Конструктор, принимающий три аргумента.
  public Triangle(
        string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Сконструировать равнобедренный треугольник.
  public Triangle(double x) : base(x) {
    Style = "равнобедренный";
  }

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

class Shapes5 {
  static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
    Triangle t3 = new Triangle(4.0);

    t1 = t2;

    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    Console.WriteLine("Площадь равна " + t1.Area());
```

```
    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Площадь равна " + t2.Area());

    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t3: ");
    t3.ShowStyle();
    t3.ShowDim();
    Console.WriteLine("Площадь равна " + t3.Area());

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Сведения об объекте t1:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48

Сведения об объекте t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48

Сведения об объекте t3:
Треугольник равнобедренный
Ширина и высота равны 4 и 4
Площадь равна 8
```

А теперь рассмотрим вкратце основные принципы действия ключевого слова `base`. Когда в производном классе указывается ключевое слово `base`, вызывается конструктор из его непосредственного базового класса. Следовательно, ключевое слово `base` всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии классов. Аргументы передаются базовому конструктору в качестве аргументов метода `base()`. Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

Наследование и сокрытие имен

В производном классе можно определить член с таким же именем, как и у члена его базового класса. В этом случае член базового класса скрывается в производном классе. И хотя формально в C# это не считается ошибкой, компилятор все же выдаст сообщение, предупреждающее о том, что имя скрывается. Если член базового класса требуется скрыть намеренно, то перед его именем следует указать ключевое слово `new`, чтобы избежать появления подобного предупреждающего сообщения. Следует,

однако, иметь в виду, что это совершенно отдельное применение ключевого слова new, не похожее на его применение при создании экземпляра объекта.

Ниже приведен пример сокрытия имени.

```
// Пример сокрытия имени с наследственной связью.

using System;

class A {
  public int i = 0;
}

// Создать производный класс.
class B : A {
  new int i; // этот член скрывает член i из класса A
  public B(int b) {
    i = b; // член i в классе B
  }

  public void Show() {
    Console.WriteLine("Член i в производном классе: " + i);
  }
}

class NameHiding {
  static void Main() {
  B ob = new B(2);

  ob.Show();
  }
}
```

Прежде всего обратите внимание на использование ключевого слова new в следующей строке кода.

```
new int i; // этот член скрывает член i из класса A
```

В этой строке компилятору, по существу, сообщается о том, что вновь создаваемая переменная i намеренно скрывает переменную i из базового класса A и что автору программы об этом известно. Если же опустить ключевое слово new в этой строке кода, то компилятор выдаст предупреждающее сообщение.

Вот к какому результату приводит выполнение приведенного выше кода.

```
Член i в производном классе: 2
```

В классе B определяется собственная переменная экземпляра i, которая скрывает переменную i из базового класса A. Поэтому при вызове метода Show() для объекта типа B выводится значение переменной i, определенной в классе B, а не той, что определена в классе A.

Применение ключевого слова base для доступа к скрытому имени

Имеется еще одна форма ключевого слова base, которая действует подобно ключевому слову this, за исключением того, что она всегда ссылается на базовый класс

в том производном классе, в котором она используется. Ниже эта форма приведена в общем виде:

```
base.член
```

где *член* может обозначать метод или переменную экземпляра. Эта форма ключевого слова `base` чаще всего применяется в тех случаях, когда под именами членов производного класса скрываются члены базового класса с теми же самыми именами. В качестве примера ниже приведен другой вариант иерархии классов из предыдущего примера.

```
// Применение ключевого слова base для преодоления
// препятствия, связанного с сокрытием имен.

using System;

class A {
  public int i = 0;
}

// Создать производный класс.
class B : A {
  new int i; // этот член скрывает член i из класса A

  public B(int a, int b) {
    base.i = a; // здесь обнаруживается скрытый член из класса A
    i = b; // член i из класса B
  }

  public void Show() {
    // Здесь выводится член i из класса A.
    Console.WriteLine("Член i в базовом классе: " + base.i);
    // А здесь выводится член i из класса B.
Console.WriteLine("Член i в производном классе: " + i);
  }
}

class UncoverName {
  static void Main() {
    B ob = new B(1, 2);
    ob.Show();
  }
}
```

Выполнение этого кода приводит к следующему результату.

```
Член i в базовом классе: 1
Член i в производном классе: 2
```

Несмотря на то что переменная экземпляра `i` в производном классе `B` скрывает переменную `i` из базового класса `A`, ключевое слово `base` разрешает доступ к переменной `i`, определенной в базовом классе.

С помощью ключевого слова `base` могут также вызываться скрытые методы. Например, в приведенном ниже коде класс `B` наследует класс `A` и в обоих классах объявляется метод `Show()`. А затем в методе `Show()` класса `B` с помощью ключевого слова `base` вызывается вариант метода `Show()`, определенный в классе `A`.

```
// Вызвать скрытый метод.

using System;

class A {
  public int i = 0;
  // Метод Show() в классе A
  public void Show() {
    Console.WriteLine("Член i в базовом классе: " + i);
  }
}

// Создать производный класс.
class B : A {
  new int i; // этот член скрывает член i из класса A

  public B(int a, int b) {
    base.i = a; // здесь обнаруживается скрытый член из класса A
    i = b; // член i из класса B
  }

  // Здесь скрывается метод Show() из класса A. Обратите
  // внимание на применение ключевого слова new.
  new public void Show() {
    base.Show(); // здесь вызывается метод Show() из класса A

    // далее выводится член i из класса B
    Console.WriteLine("Член i в производном классе: " + i);
  }
}

class UncoverName {
  static void Main() {
    B ob = new B(1, 2);
    ob.Show();
  }
}
```

Выполнение этого кода приводит к следующему результату.

```
Член i в базовом классе: 1
Член i в производном классе: 2
```

Как видите, в выражении `base.Show()` вызывается вариант метода `Show()` из базового класса.

Обратите также внимание на следующее: ключевое слово `new` используется в приведенном выше коде с целью сообщить компилятору о том, что метод `Show()`, вновь объявляемый в производном классе `B`, намеренно скрывает другой метод `Show()`, определенный в базовом классе `A`.

Создание многоуровневой иерархии классов

В представленных до сих пор примерах программ использовались простые иерархии классов, состоявшие только из базового и производного классов. Но в C# мож-

но также строить иерархии, состоящие из любого числа уровней наследования. Как упоминалось выше, многоуровневая иерархия идеально подходит для использования одного производного класса в качестве базового для другого производного класса. Так, если имеются три класса, A, B и C, то класс C может наследовать от класса B, а тот, в свою очередь, от класса A. В таком случае каждый производный класс наследует характерные особенности всех своих базовых классов. В частности, класс C наследует все члены классов B и A.

Для того чтобы показать, насколько полезной может оказаться многоуровневая иерархия классов, рассмотрим следующий пример программы. В ней производный класс `Triangle` служит в качестве базового для создания другого производного класса — `ColorTriangle`. При этом класс `ColorTriangle` наследует все характерные особенности, а по существу, члены классов `Triangle` и `TwoDShape`, к которым добавляется поле `color`, содержащее цвет треугольника.

```
// Пример построения многоуровневой иерархии классов.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Конструктор, используемый по умолчанию.
  public TwoDShape() {
    Width = Height = 0.0;
  }

  // Конструктор для класса TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Сконструировать объект равной ширины и высоты.
  public TwoDShape(double x) {
    Width = Height = x;
  }

  // Свойства ширины и высоты объекта.
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
```

```
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style; // закрытый член класса

  /* Конструктор, используемый по умолчанию.
     Автоматически вызывает конструктор, доступный по
     умолчанию в классе TwoDShape. */
  public Triangle() {
    Style = "null";
  }

  // Конструктор.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Сконструировать равнобедренный треугольник.
  public Triangle(double x) : base(x) {
    Style = "равнобедренный";
  }

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

// Расширить класс Triangle.
class ColorTriangle : Triangle {
  string color;

  public ColorTriangle(string c, string s,
                       double w, double h) : base(s, w, h) {
    color = c;
  }

  // Показать цвет треугольника.
  public void ShowColor() {
    Console.WriteLine("Цвет " + color);
  }
}

class Shapes6 {
  static void Main() {
    ColorTriangle t1 =
         new ColorTriangle("синий", "прямоугольный", 8.0, 12.0);
    ColorTriangle t2 =
```

```
        new ColorTriangle("красный", "равнобедренный", 2.0, 2.0);

    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle();
    t1.ShowDim();
    t1.ShowColor();
    Console.WriteLine("Площадь равна " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    t2.ShowColor();
    Console.WriteLine("Площадь равна " + t2.Area());
  }
}
```

При выполнении этой программы получается следующий результат.

```
Сведения об объекте t1:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Цвет синий
Площадь равна 48

Сведения об объекте t2:
Треугольник равнобедренный
Ширина и высота равны 2 и 2
Цвет красный
Площадь равна 2
```

Благодаря наследованию в классе `ColorTriangle` могут использоваться определенные ранее классы `Triangle` и `TwoDShape`, к элементам которых добавляется лишь та информация, которая требуется для конкретного применения данного класса. В этом отчасти и состоит ценность наследования, поскольку оно допускает повторное использование кода.

Приведенный выше пример демонстрирует еще одно важное положение: ключевое слово `base` всегда обозначает ссылку на конструктор ближайшего по иерархии базового класса. Так, ключевое слово `base` в классе `ColorTriangle` обозначает вызов конструктора из класса `Triangle`, а ключевое слово `base` в классе `Triangle` — вызов конструктора из класса `TwoDShape`. Если же в иерархии классов конструктору базового класса требуются параметры, то все производные классы должны предоставлять эти параметры вверх по иерархии, независимо от того, требуются они самому производному классу или нет.

Порядок вызова конструкторов

В связи с изложенными выше в отношении наследования и иерархии классов может возникнуть следующий резонный вопрос: когда создается объект производного класса и какой конструктор выполняется первым — тот, что определен в производном классе, или же тот, что определен в базовом классе? Так, если имеется базовый класс `A`

и производный класс B, то вызывается ли конструктор класса A раньше конструктора класса B? Ответ на этот вопрос состоит в том, что в иерархии классов конструкторы вызываются по порядку выведения классов: от базового к производному. Более того, этот порядок остается неизменным независимо от использования ключевого слова base. Так, если ключевое слово base не используется, то выполняется конструктор по умолчанию, т.е. конструктор без параметров. В приведенном ниже примере программы демонстрируется порядок вызова и выполнения конструкторов.

```
// Продемонстрировать порядок вызова конструкторов.

using System;

// Создать базовый класс.
class A {
  public A() {
    Console.WriteLine("Конструирование класса A.");
  }
}

// Создать класс, производный от класса A.
class B : A {
  public B() {
    Console.WriteLine("Конструирование класса B.");
  }
}

// Создать класс, производный от класса B.
class C : B {
  public C() {
    Console.WriteLine("Конструирование класса C.");
  }
}

class OrderOfConstruction {
  static void Main() {
    C c = new C();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Конструирование класса A.
Конструирование класса B.
Конструирование класса C.
```

Как видите, конструкторы вызываются по порядку выведения их классов.

Если хорошенько подумать, то в вызове конструкторов по порядку выведения их классов можно обнаружить определенный смысл. Ведь базовому классу ничего не известно ни об одном из производных от него классов, и поэтому любая инициализация, которая требуется его членам, осуществляется совершенно отдельно от инициализации членов производного класса, а возможно, это и необходимое условие. Следовательно, она должна выполняться первой.

Ссылки на базовый класс и объекты производных классов

Как вам должно быть уже известно, C# является строго типизированным языком программирования. Помимо стандартных преобразований и автоматического продвижения простых типов значений, в этом языке строго соблюдается принцип совместимости типов. Это означает, что переменная ссылки на объект класса одного типа, как правило, не может ссылаться на объект класса другого типа. В качестве примера рассмотрим следующую программу, в которой объявляются два класса одинаковой структуры.

```
// Эта программа не подлежит компиляции.

class X {
  int a;

  public X(int i) { a = i; }
}

class Y {
  int a;
  public Y(int i) { a = i; }
}

class IncompatibleRef {
  static void Main() {
    X x = new X(10);
    X x2;
    Y y = new Y(5);

    x2 = x; // верно, поскольку оба объекта относятся к одному и тому же типу
    x2 = y; // ошибка, поскольку это разнотипные объекты
  }
}
```

Несмотря на то что классы `X` и `Y` в данном примере совершенно одинаковы по своей структуре, ссылку на объект типа `Y` нельзя присвоить переменной ссылки на объект типа `X`, поскольку типы у них разные. Поэтому следующая строка кода оказывается неверной и может привести к ошибке из-за несовместимости типов во время компиляции.

```
x2 = y; // неверно, поскольку это разнотипные объекты
```

Вообще говоря, переменная ссылки на объект может ссылаться только на объект своего типа.

Но из этого принципа строгого соблюдения типов в C# имеется одно важное исключение: переменной ссылки на объект базового класса может быть присвоена ссылка на объект *любого* производного от него класса. Такое присваивание считается вполне допустимым, поскольку экземпляр объекта производного типа инкапсулирует экземпляр объекта базового типа. Следовательно, по ссылке на объект базового класса можно обращаться к объекту производного класса. Ниже приведен соответствующий пример.

```
// По ссылке на объект базового класса можно обращаться
// к объекту производного класса.

using System;

class X {
  public int a;

  public X(int i) {
    a = i;
  }
}

class Y : X {
  public int b;

  public Y(int i, int j) : base(j) {
    b = i;
  }
}

class BaseRef {
  static void Main() {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // верно, поскольку оба объекта относятся к одному и тому же типу
    Console.WriteLine("x2.a: " + x2.a);

    x2 = y; // тоже верно, поскольку класс Y является производным от класса X
    Console.WriteLine("x2.a: " + x2.a);

    // ссылкам на объекты класса X известно только о членах класса X
    x2.a = 19; // верно
//    x2.b = 27; // неверно, поскольку член b отсутствует у класса X
  }
}
```

В данном примере класс `Y` является производным от класса `X`. Поэтому следующая операция присваивания:

```
x2 = y; // тоже верно, поскольку класс Y является производным от класса X
```

считается вполне допустимой. Ведь по ссылке на объект базового класса (в данном случае — это переменная `x2` ссылки на объект класса `X`) можно обращаться к объекту производного класса, т.е. к объекту, на который ссылается переменная `y`.

Следует особо подчеркнуть, что доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается. Это означает, что если ссылка на объект производного класса присваивается переменной ссылки на объект базового класса, то доступ разрешается только к тем частям этого объекта, которые определяются базовым классом. Именно поэтому переменной `x2` недоступен член `b` класса `Y`, когда она ссылается на объект этого класса. И в этом есть своя логика, поскольку базовому классу ничего не известно о тех членах, которые до-

бавлены в производный от него класс. Именно поэтому последняя строка кода в приведенном выше примере была закомментирована.

Несмотря на кажущийся несколько отвлеченным характер приведенных выше рассуждений, им можно найти ряд важных применений на практике. Одно из них рассматривается ниже, а другое — далее в этой главе, когда речь пойдет о виртуальных методах.

Один из самых важных моментов для присваивания ссылок на объекты производного класса переменным базового класса наступает тогда, когда конструкторы вызываются в иерархии классов. Как вам должно быть уже известно, в классе нередко определяется конструктор, принимающий объект своего класса в качестве параметра. Благодаря этому в классе может быть сконструирована копия его объекта. Этой особенностью можно выгодно воспользоваться в классах, производных от такого класса. В качестве примера рассмотрим очередные варианты классов `TwoDShape` и `Triangle`. В оба класса добавлены конструкторы, принимающие объект в качестве параметра.

```
// Передать ссылку на объект производного класса
// переменной ссылки на объект базового класса.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Конструктор по умолчанию.
  public TwoDShape() {
    Width = Height = 0.0;
  }

  // Конструктор для класса TwoDShape.
  public TwoDShape(double w, double h) {
    Width = w;
    Height = h;
  }

  // Сконструировать объект равной ширины и высоты.
  public TwoDShape(double x) {
    Width = Height = x;
  }

  // Сконструировать копию объекта TwoDShape.
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
  }

  // Свойства ширины и высоты объекта.
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
```

```
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style;

  // Конструктор, используемый по умолчанию.
  public Triangle() {
    Style = "null";
  }

  // Конструктор для класса Triangle.
  public Triangle(string s, double w, double h) : base(w, h) {
    Style = s;
  }

  // Сконструировать равнобедренный треугольник.
  public Triangle(double x) : base(x) {
    Style = "равнобедренный";
  }

  // Сконструировать копию объекта типа Triangle.
  public Triangle(Triangle ob) : base(ob) {
  Style = ob.Style;
  }

  // Возвратить площадь треугольника.
  public double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

class Shapes7 {
  static void Main() {
    Triangle t1 = new Triangle("прямоугольный", 8.0, 12.0);

    // Сделать копию объекта t1.
    Triangle t2 = new Triangle(t1);

    Console.WriteLine("Сведения об объекте t1: ");
    t1.ShowStyle();
```

```
    t1.ShowDim();
    Console.WriteLine("Площадь равна " + t1.Area());

    Console.WriteLine();

    Console.WriteLine("Сведения об объекте t2: ");
    t2.ShowStyle();
    t2.ShowDim();
    Console.WriteLine("Площадь равна " + t2.Area());
  }
}
```

В представленном выше примере объект `t2` конструируется из объекта `t1` и поэтому подобен ему. Ниже приведен результат выполнения кода из данного примера.

```
Сведения об объекте t1:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48

Сведения об объекте t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48
```

Обратите особое внимание на следующий конструктор класса `Triangle`:

```
public Triangle(Triangle ob) : base(ob) {
  Style = ob.Style;
}
```

Он принимает объект типа `Triangle` в качестве своего параметра и передает его (с помощью ключевого слова `base`) следующему конструктору класса `TwoDShape`.

```
public TwoDShape(TwoDShape ob) {
  Width = ob.Width;
  Height = ob.Height;
}
```

Самое любопытное, что конструктор `TwoDShape()` предполагает получить объект класса `TwoDShape`, тогда как конструктор `Triangle()` передает ему объект класса `Triangle`. Как пояснялось выше, такое вполне допустимо, поскольку по ссылке на объект базового класса можно обращаться к объекту производного класса. Следовательно, конструктору `TwoDShape()` можно на совершенно законных основаниях передать ссылку на объект класса, производного от класса `TwoDShape`. А поскольку конструктор `TwoDShape()` инициализирует только те части объекта производного класса, которые являются членами класса `TwoDShape`, то для него не имеет никакого значения, содержит ли этот объект другие члены, добавленные в производном классе.

Виртуальные методы и их переопределение

Виртуальным называется такой метод, который объявляется как `virtual` в базовом классе. Виртуальный метод отличается тем, что он может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса

может быть свой вариант виртуального метода. Кроме того, виртуальные методы интересны тем, что именно происходит при их вызове по ссылке на базовый класс. В этом случае средствами языка C# определяется именно тот вариант виртуального метода, который следует вызывать, исходя из *типа* объекта, к которому происходит обращение *по ссылке*, причем это делается *во время выполнения*. Поэтому при ссылке на разные типы объектов выполняются разные варианты виртуального метода. Иными словами, вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу ссылки на этот объект. Так, если базовый класс содержит виртуальный метод и от него получены производные классы, то при обращении к разным типам объектов по ссылке на базовый класс выполняются разные варианты этого виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова `virtual`, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификатор `override`. А сам процесс повторного определения виртуального метода в производном классе называется *переопределением метода*. При переопределении имя, возвращаемый тип и сигнатура переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется. Кроме того, виртуальный метод не может быть объявлен как `static` или `abstract` (подробнее данный вопрос рассматривается далее в этой главе).

Переопределение метода служит основанием для воплощения одного из самых эффективных в C# принципов: *динамической диспетчеризации методов*, которая представляет собой механизм разрешения вызова во время выполнения, а не компиляции. Значение динамической диспетчеризации методов состоит в том, что именно благодаря ей в C# реализуется динамический полиморфизм.

Ниже приведен пример, демонстрирующий виртуальные методы и их переопределение.

```
// Продемонстрировать виртуальный метод.

using System;

class Base {
  // Создать виртуальный метод в базовом классе.
  public virtual void Who() {
    Console.WriteLine("Метод Who() в классе Base");
  }
}

class Derived1 : Base {
  // Переопределить метод Who() в производном классе.
  public override void Who() {
    Console.WriteLine("Метод Who() в классе Derived1");
  }
}

class Derived2 : Base {
  // Вновь переопределить метод Who() в еще одном производном классе.
  public override void Who() {
    Console.WriteLine("Метод Who() в классе Derived2");
  }
}
```

```
class OverrideDemo {
  static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derived2 dOb2 = new Derived2();

    Base baseRef; // ссылка на базовый класс

    baseRef = baseOb;
    baseRef.Who();

    baseRef = dOb1;
    baseRef.Who();

    baseRef = dOb2;
    baseRef.Who();
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Метод Who() в классе Base.
Метод Who() в классе Derived1
Метод Who() в классе Derived2
```

В коде из приведенного выше примера создаются базовый класс `Base` и два производных от него класса — `Derived1` и `Derived2`. В классе `Base` объявляется виртуальный метод `Who()`, который переопределяется в обоих производных классах. Затем в методе `Main()` объявляются объекты типа `Base`, `Derived1` и `Derived2`. Кроме того, объявляется переменная `baseRef` ссылочного типа `Base`. Далее ссылка на каждый тип объекта присваивается переменной `baseRef` и затем используется для вызова метода `Who()`. Как следует из результата выполнения приведенного выше кода, вариант выполняемого метода `Who()` определяется по типу объекта, к которому происходит обращение по ссылке во время вызова этого метода, а не по типу класса переменной `baseRef`.

Но переопределять виртуальный метод совсем не обязательно. Ведь если в производном классе не предоставляется собственный вариант виртуального метода, то используется его вариант из базового класса, как в приведенном ниже примере.

```
/* Если виртуальный метод не переопределяется, то
   используется его вариант из базового класса. */

using System;

class Base {
  // Создать виртуальный метод в базовом классе.
  public virtual void Who() {
    Console.WriteLine("Метод Who() в классе Base");
  }
}

class Derived1 : Base {
  // Переопределить метод Who() в производном классе.
```

```
  public override void Who() {
    Console.WriteLine("Метод Who() в классе Derived1");
  }
}

class Derived2 : Base {
  // В этом классе метод Who() не переопределяется.
}

class NoOverrideDemo {
  static void Main() {
    Base baseOb = new Base();
    Derived1 dOb1 = new Derived1();
    Derived2 dOb2 = new Derived2();

    Base baseRef; // ссылка на базовый класс

    baseRef = baseOb;
    baseRef.Who();

    baseRef = dOb1;
    baseRef.Who();

    baseRef = dOb2;
    baseRef.Who(); // вызывается метод Who() из класса Base
  }
}
```

Выполнение этого кода приводит к следующему результату.

```
Метод Who() в классе Base.
Метод Who() в классе Derived1
Метод Who() в классе Base
```

В данном примере метод `Who()` не переопределяется в классе `Derived2`. Поэтому для объекта класса `Derived2` вызывается метод `Who()` из класса `Base`.

Если при наличии многоуровневой иерархии виртуальный метод не переопределяется в производном классе, то выполняется ближайший его вариант, обнаруживаемый вверх по иерархии, как в приведенном ниже примере.

```
/* В многоуровневой иерархии классов выполняется тот
   переопределенный вариант виртуального метода,
   который обнаруживается первым при продвижении
   вверх по иерархии. */

using System;

class Base {
  // Создать виртуальный метод в базовом классе.
  public virtual void Who() {
    Console.WriteLine("Метод Who() в классе Base");
  }
}

class Derived1 : Base {
```

```
  // Переопределить метод Who() в производном классе.
  public override void Who() {
    Console.WriteLine("Метод Who() в классе Derived1");
  }
}

class Derived2 : Derived1 {
  // В этом классе метод Who() не переопределяется.
}

class Derived3 : Derived2 {
  // И в этом классе метод Who() не переопределяется.
}

class NoOverrideDemo2 {
  static void Main() {
    Derived3 dOb = new Derived3();
    Base baseRef; // ссылка на базовый класс

    baseRef = dOb;
    baseRef.Who(); // вызов метода Who() из класса Derived1
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Метод Who() в классе Derived1
```

В данном примере класс `Derived3` наследует класс `Derived2`, который наследует класс `Derived1`, а тот, в свою очередь, — класс `Base`. Как показывает приведенный выше результат, выполняется метод `Who()`, переопределяемый в классе `Derived1`, поскольку это первый вариант виртуального метода, обнаруживаемый при продвижении вверх по иерархии от классов `Derived3` и `Derived2`, где метод `Who()` не переопределяется, к классу `Derived1`.

И еще одно замечание: свойства также подлежат модификации ключевым словом `virtual` и переопределению ключевым словом `override`. Это же относится и к индексаторам.

Что дает переопределение методов

Благодаря переопределению методов в C# поддерживается динамический полиморфизм. В объектно-ориентированном программировании полиморфизм играет очень важную роль, потому что он позволяет определить в общем классе методы, которые становятся общими для всех производных от него классов, а в производных классах — определить конкретную реализацию некоторых или же всех этих методов. Переопределение методов — это еще один способ воплотить в C# главный принцип полиморфизма: один интерфейс — множество методов.

Удачное применение полиморфизма отчасти зависит от правильного понимания той особенности, что базовые и производные классы образуют иерархию, которая продвигается от меньшей к большей специализации. При надлежащем применении базовый класс предоставляет все необходимые элементы, которые могут использоваться в производном классе непосредственно. А с помощью виртуальных методов в базовом

классе определяются те методы, которые могут быть самостоятельно реализованы в производном классе. Таким образом, сочетая наследование с виртуальными методами, можно определить в базовом классе общую форму методов, которые будут использоваться во всех его производных классах.

Применение виртуальных методов

Для того чтобы стали понятнее преимущества виртуальных методов, применим их в классе `TwoDShape`. В предыдущих примерах в каждом классе, производном от класса `TwoDShape`, определялся метод `Area()`. Но, по-видимому, метод `Area()` лучше было бы сделать виртуальным в классе `TwoDShape` и тем самым предоставить возможность переопределить его в каждом производном классе с учетом особенностей расчета площади той двумерной формы, которую инкапсулирует этот класс. Именно это и сделано в приведенном ниже примере программы. Ради удобства демонстрации классов в этой программе введено также свойство `name` в классе `TwoDShape`.

```
// Применить виртуальные методы и полиморфизм.

using System;

class TwoDShape {
  double pri_width;
  double pri_height;

  // Конструктор по умолчанию.
  public TwoDShape() {
    Width = Height = 0.0;
    name = "null";
}

  // Параметризированный конструктор.
  public TwoDShape(double w, double h, string n) {
    Width = w;
    Height = h;
    name = n;
  }

  // Сконструировать объект равной ширины и высоты.
  public TwoDShape(double x, string n) {
    Width = Height = x;
    name = n;
  }

  // Сконструировать копию объекта TwoDShape.
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
    name = ob.name;
  }

  // Свойства ширины и высоты объекта.
  public double Width {
    get { return pri_width; }
```

```
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public string name { get; set; }

  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }

  public virtual double Area() {
    Console.WriteLine("Метод Area() должен быть переопределен");
    return 0.0;
  }
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style;

  // Конструктор, используемый по умолчанию.
  public Triangle() {
    Style = "null";
  }

  // Конструктор для класса Triangle.
  public Triangle(string s, double w, double h) :
         base(w, h, "треугольник") {
    Style = s;
  }

  // Сконструировать равнобедренный треугольник.
  public Triangle(double x) : base(x, "треугольник") {
    Style = "равнобедренный";
  }

  // Сконструировать копию объекта типа Triangle.
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }

  // Переопределить метод Area() для класса Triangle.
  public override double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
```

```
}

// Класс для прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape {
  // Конструктор для класса Rectangle.
  public Rectangle(double w, double h) :
    base(w, h, "прямоугольник"){ }
  // Сконструировать квадрат.
  public Rectangle(double x) :
    base(x, "прямоугольник") { }

  // Сконструировать копию объекта типа Rectangle.
  public Rectangle(Rectangle ob) : base(ob) { }

  // Возвратить логическое значение true, если
  // прямоугольник окажется квадратом.
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }

  // Переопределить метод Area() для класса Rectangle.
  public override double Area() {
    return Width * Height;
  }
}

class DynShapes {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[5];

    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);
    shapes[4] = new TwoDShape(10, 20, "общая форма");

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объект - " + shapes[i].name);
      Console.WriteLine("Площадь равна " + shapes[i].Area());

      Console.WriteLine();
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
Объект - треугольник
Площадь равна 48

Объект - прямоугольник
Площадь равна 100

Объект - прямоугольник
```

```
Площадь равна 40

Объект — треугольник
Площадь равна 24.5

Объект — общая форма
Метод Area() должен быть переопределен
Площадь равна 0
```

Рассмотрим данный пример программы более подробно. Прежде всего, метод `Area()` объявляется как `virtual` в классе `TwoDShape` и переопределяется в классах `Triangle` и `Rectangle` по объяснявшимся ранее причинам. В классе `TwoDShape` метод `Area()` реализован в виде заполнителя, который сообщает о том, что пользователь данного метода должен переопределить его в производном классе. Каждое переопределение метода `Area()` предоставляет конкретную его реализацию, соответствующую типу объекта, инкапсулируемого в производном классе. Так, если реализовать класс для эллипсов, то метод `Area()` должен вычислять площадь эллипса.

У программы из рассматриваемого здесь примера имеется еще одна примечательная особенность. Обратите внимание на то, что в методе `Main()` двумерные формы объявляются в виде массива объектов типа `TwoDShape`, но элементам этого массива присваиваются ссылки на объекты классов `Triangle`, `Rectangle` и `TwoDShape`. И это вполне допустимо, поскольку по ссылке на базовый класс можно обращаться к объекту производного класса. Далее в программе происходит циклическое обращения к элементам данного массива для вывода сведений о каждом объекте. Несмотря на всю свою простоту, данный пример наглядно демонстрирует преимущества наследования и переопределения методов. Тип объекта, хранящийся в переменной ссылки на базовый класс, определяется во время выполнения и соответственно обусловливает дальнейшие действия. Так, если объект является производным от класса `TwoDShape`, то для получения его площади вызывается метод `Area()`. Но интерфейс для выполнения этой операции остается тем же самым независимо от типа используемой двумерной формы.

Применение абстрактных классов

Иногда требуется создать базовый класс, в котором определяется лишь самая общая форма для всех его производных классов, а наполнение ее деталями предоставляется каждому из этих классов. В таком классе определяется лишь характер методов, которые должны быть конкретно реализованы в производных классах, а не в самом базовом классе. Подобная ситуация возникает, например, в связи с невозможностью получить содержательную реализацию метода в базовом классе. Именно такая ситуация была продемонстрирована в варианте класса `TwoDShape` из предыдущего примера, где метод `Area()` был просто определен как заполнитель. Такой метод не вычисляет и не выводит площадь двумерного объекта любого типа.

Создавая собственные библиотеки классов, вы можете сами убедиться в том, что у метода зачастую отсутствует содержательное определение в контексте его базового класса. Подобная ситуация разрешается двумя способами. Один из них, как показано в предыдущем примере, состоит в том, чтобы просто выдать предупреждающее сообщение. Такой способ может пригодиться в определенных ситуациях, например при отладке, но в практике программирования он обычно не применяется. Ведь в базовом

классе могут быть объявлены методы, которые должны быть переопределены в производном классе, чтобы этот класс стал содержательным. Рассмотрим для примера класс `Triangle`. Он был бы неполным, если бы в нем не был переопределен метод `Area()`. В подобных случаях требуется какой-то способ, гарантирующий, что в производном классе действительно будут переопределены все необходимые методы. И такой способ в C# имеется. Он состоит в использовании абстрактного метода.

Абстрактный метод создается с помощью указываемого модификатора типа `abstract`. У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом классе. Это означает, что он должен быть переопределен в производном классе, поскольку его вариант из базового класса просто непригоден для использования. Нетрудно догадаться, что абстрактный метод автоматически становится виртуальным и не требует указания модификатора `virtual`. В действительности совместное использование модификаторов `virtual` и `abstract` считается ошибкой.

Для определения абстрактного метода служит приведенная ниже общая форма.

```
abstract тип имя(список_параметров);
```

Как видите, у абстрактного метода отсутствует тело. Модификатор `abstract` может применяться только в методах экземпляра, но не в статических методах (`static`). Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением `class` указывается модификатор `abstract`. А поскольку реализация абстрактного класса не определяется полностью, то у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора `new` приведет к ошибке во время компиляции.

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как `abstract`. Таким образом, атрибут `abstract` наследуется до тех пор, пока не будет достигнута полная реализация класса.

Используя абстрактный класс, мы можем усовершенствовать рассматривавшийся ранее класс `TwoDShape`. Для неопределенной двухмерной фигуры понятие площади не имеет никакого смысла, поэтому в приведенном ниже варианте класса `TwoDShape` метод `Area()` и сам класс `TwoDShape` объявляются как `abstract`. Это, конечно, означает, что во всех классах, производных от класса `TwoDShape`, должен быть переопределен метод `Area()`.

```
// Создать абстрактный класс.

using System;

abstract class TwoDShape {
  double pri_width;
  double pri_height;

  // Конструктор, используемый по умолчанию.
  public TwoDShape() {
    Width = Height = 0.0;
    name = "null";
  }

  // Параметризированный конструктор.
```

```
  public TwoDShape(double w, double h, string n) {
    Width = w;
    Height = h;
    name = n;
  }

  // Сконструировать объект равной ширины и высоты.
  public TwoDShape(double x, string n) {
    Width = Height = x;
    name = n;
  }

  // Сконструировать копию объекта TwoDShape.
  public TwoDShape(TwoDShape ob) {
    Width = ob.Width;
    Height = ob.Height;
    name = ob.name;
  }

  // Свойства ширины и высоты объекта.
  public double Width {
    get { return pri_width; }
    set { pri_width = value < 0 ? -value : value; }
  }

  public double Height {
    get { return pri_height; }
    set { pri_height = value < 0 ? -value : value; }
  }

  public string name { get; set; }
  public void ShowDim() {
    Console.WriteLine("Ширина и высота равны " +
                      Width + " и " + Height);
  }

  // Теперь метод Area() является абстрактным.
  public abstract double Area();
}

// Класс для треугольников, производный от класса TwoDShape.
class Triangle : TwoDShape {
  string Style;

  // Конструктор, используемый по умолчанию.
  public Triangle() {
    Style = "null";
  }

  // Конструктор для класса Triangle.
  public Triangle(string s, double w, double h) :
         base(w, h, "треугольник") {
    Style = s;
  }
```

```
  // Сконструировать равнобедренный треугольник.
  public Triangle(double x) : base(x, "треугольник") {
    Style = "равнобедренный";
  }

  // Сконструировать копию объекта типа Triangle.
  public Triangle(Triangle ob) : base(ob) {
    Style = ob.Style;
  }

  // Переопределить метод Area() для класса Triangle.
  public override double Area() {
    return Width * Height / 2;
  }

  // Показать тип треугольника.
  public void ShowStyle() {
    Console.WriteLine("Треугольник " + Style);
  }
}

// Класс для прямоугольников, производный от класса TwoDShape
class Rectangle : TwoDShape {
  // Конструктор для класса Rectangle.
  public Rectangle(double w, double h) :
         base(w, h, "прямоугольник"){ }

  // Сконструировать квадрат.
  public Rectangle(double x) :
    base(x, "прямоугольник") { }

  // Сконструировать копию объекта типа Rectangle.
  public Rectangle(Rectangle ob) : base(ob) { }

  // Возвратить логическое значение true, если
  // прямоугольник окажется квадратом.
  public bool IsSquare() {
    if(Width == Height) return true;
    return false;
  }

  // Переопределить метод Area() для класса Rectangle.
  public override double Area() {
    return Width * Height;
  }
}

class AbsShape {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[4];

    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
```

```
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(7.0);

    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объект - " + shapes[i].name);
      Console.WriteLine("Площадь равна " + shapes[i].Area());

      Console.WriteLine();
    }
  }
}
```

Как показывает представленный выше пример программы, во всех производных классах метод `Area()` должен быть непременно переопределен, а также объявлен абстрактным. Убедитесь в этом сами, попробовав создать производный класс, в котором не переопределен метод `Area()`. В итоге вы получите сообщение об ошибке во время компиляции. Конечно, возможность создавать ссылки на объекты типа `TwoDShape` по-прежнему существует, и это было сделано в приведенном выше примере программы, но объявлять объекты типа `TwoDShape` уже нельзя. Именно поэтому массив `shapes` сокращен в методе `Main()` до 4 элементов, а объект типа `TwoDShape` для общей двухмерной формы больше не создается.

Обратите также внимание на то, что в класс `TwoDShape` по-прежнему входит метод `ShowDim()` и что он не объявляется с модификатором `abstract`. В абстрактные классы вполне допускается (и часто практикуется) включать конкретные методы, которые могут быть использованы в своем исходном виде в производном классе. А переопределению в производных классах подлежат только те методы, которые объявлены как `abstract`.

Предотвращение наследования с помощью ключевого слова `sealed`

Несмотря на всю эффективность и полезность наследования, иногда возникает потребность предотвратить его. Допустим, что имеется класс, инкапсулирующий последовательность инициализации некоторого специального оборудования, например медицинского монитора. В этом случае требуется, чтобы пользователи данного класса не могли изменять порядок инициализации монитора, чтобы исключить его неправильную настройку. Но независимо от конкретных причин в C# имеется возможность предотвратить наследование класса с помощью ключевого слова `sealed`.

Для того чтобы предотвратить наследование класса, достаточно указать ключевое слово `sealed` перед определением класса. Как и следовало ожидать, класс не допускается объявлять одновременно как `abstract` и `sealed`, поскольку сам абстрактный класс реализован не полностью и опирается в этом отношении на свои производные классы, обеспечивающие полную реализацию.

Ниже приведен пример объявления класса типа `sealed`.

```
sealed class A {
  // ...
}

// Следующий класс недопустим.
```

```
class B : A { // ОШИБКА! Наследовать класс A нельзя
  // ...
}
```

Как следует из комментариев в приведенном выше фрагменте кода, класс B не может наследовать класс A, потому что последний объявлен как sealed.

И еще одно замечание: ключевое слово sealed может быть также использовано в виртуальных методах для предотвращения их дальнейшего переопределения. Допустим, что имеется базовый класс B и производный класс D. Метод, объявленный в классе B как virtual, может быть объявлен в классе D как sealed. Благодаря этому в любом классе, наследующем от класса D, предотвращается переопределение данного метода. Подобная ситуация демонстрируется в приведенном ниже фрагменте кода.

```
class B {
  public virtual void MyMethod() { /* ... */ }
}

class D : B {
  // Здесь герметизируется метод MyMethod() и
  // предотвращается его дальнейшее переопределение.
  sealed public override void MyMethod() { /* ... */ }
  }

class X : D {
  // Ошибка! Метод MyMethod() герметизирован!
  public override void MyMethod() { /* ... */ }
}
```

Метод MyMethod() герметизирован в классе D, и поэтому не может быть переопределен в классе X.

Класс object

В C# предусмотрен специальный класс object, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от object. Это, в частности, означает, что переменная ссылочного типа object может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты. Формально имя object считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework.

В классе object определяются методы, приведенные в табл. 11.1. Это означает, что они доступны для каждого объекта.

Некоторые из этих методов требуют дополнительных пояснений. По умолчанию метод Equals(object) определяет, ссылается ли вызывающий объект на тот же самый объект, что и объект, указываемый в качества аргумента этого метода, т.е. он определяет, являются ли обе ссылки одинаковыми. Метод Equals(object) возвращает логическое значение true, если сравниваемые объекты одинаковы, в противном случае — логическое значение false. Он может быть также переопределен в создаваемых классах. Это позволяет выяснить, что же означает равенство объектов для создаваемого класса. Например, метод Equals(object) можно определить таким образом, чтобы в нем сравнивалось содержимое двух объектов.

Таблица 11.1. Методы класса `object`

Метод	Назначение
`public virtual bool Equals(object ob)`	Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке *ob*
`public static bool Equals(object objA, object objB)`	Определяет, является ли объект, доступный по ссылке *objA*, таким же, как и объект, доступный по ссылке *objB*
`protected Finalize()`	Выполняет завершающие действия перед "сборкой мусора". В C# метод `Finalize()` доступен посредством деструктора
`public virtual int GetHashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(obj objA, object objB)`	Определяет, делаются ли ссылки *objA* и *objB* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Метод `GetHashCode()` возвращает хеш-код, связанный с вызывающим объектом. Этот хеш-код можно затем использовать в любом алгоритме, где хеширование применяется в качестве средства доступа к хранимым объектам. Следует, однако, иметь в виду, что стандартная реализация метода `GetHashCode()` не пригодна на все случаи применения.

Как упоминалось в главе 9, если перегружается оператор ==, то обычно приходится переопределять методы `Equals(object)` и `GetHashCode()`, поскольку чаще всего требуется, чтобы метод `Equals(object)` и оператор == функционировали одинаково. Когда же переопределяется метод `Equals(object)`, то следует переопределить и метод `GetHashCode()`, чтобы оба метода оказались совместимыми.

Метод `ToString()` возвращает символьную строку, содержащую описание того объекта, для которого он вызывается. Кроме того, метод `ToString()` автоматически вызывается при выводе содержимого объекта с помощью метода `WriteLine()`. Этот метод переопределяется во многих классах, что позволяет приспосабливать описание к конкретным типам объектов, создаваемых в этих классах. Ниже приведен пример применения данного метода.

```
// Продемонстрировать применение метода ToString()

using System;

class MyClass {
  static int count = 0;
    int id;

  public MyClass() {
```

```
    id = count;
    count++;
  }

  public override string ToString() {
    return "Объект #" + id + " типа MyClass";
  }
}

class Test {
  static void Main() {
    MyClass ob1 = new MyClass();
    MyClass ob2 = new MyClass();
    MyClass ob3 = new MyClass();

    Console.WriteLine(ob1);
    Console.WriteLine(ob2);
    Console.WriteLine(ob3);
  }
}
```

При выполнении этого кода получается следующий результат.

```
Объект #0 типа MyClass
Объект #1 типа MyClass
Объект #2 типа MyClass
```

Упаковка и распаковка

Как пояснялось выше, все типы в C#, включая и простые типы значений, являются производными от класса `object`. Следовательно, ссылкой типа `object` можно воспользоваться для обращения к любому другому типу, в том числе и к типам значений. Когда ссылка на объект класса `object` используется для обращения к типу значения, то такой процесс называется *упаковкой*. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект. Но в любом случае упаковка происходит автоматически. Для этого достаточно присвоить значение переменной ссылочного типа `object`, а об остальном позаботится компилятор C#.

Распаковка представляет собой процесс извлечения упакованного значения из объекта. Это делается с помощью явного приведения типа ссылки на объект класса `object` к соответствующему типу значения. Попытка распаковать объект в другой тип может привести к ошибке во время выполнения.

Ниже приведен простой пример, демонстрирующий упаковку и распаковку.

```
// Простой пример упаковки и распаковки.

using System;

class BoxingDemo {
  static void Main() {
    int x;
    object obj;
```

```
    x = 10;
    obj = x; // упаковать значение переменной x в объект

    int y = (int)obj; // распаковать значение из объекта, доступного по
                      // ссылке obj, в переменную типа int
    Console.WriteLine(y);
  }
}
```

В этом примере кода выводится значение 10. Обратите внимание на то, что значение переменной x упаковывается в объект простым его присваиванием переменной obj, ссылающейся на этот объект. А затем это значение извлекается из объекта, доступного по его ссылке obj, и далее приводится к типу int.

Ниже приведен еще один, более интересный пример упаковки. В данном случае значение типа int передается в качестве аргумента методу Sqr(), который, в свою очередь, принимает параметр типа object.

```
// Пример упаковки при передаче значения методу.

using System;

class BoxingDemo {
  static void Main() {
    int x;
    x = 10;

    Console.WriteLine("Значение x равно: " + x);

    // значение переменной x автоматически упаковывается
    // когда оно передается методу Sqr().
    x = BoxingDemo.Sqr(x);
    Console.WriteLine("Значение x в квадрате равно: " + x);
  }

  static int Sqr(object o) {
    return (int)o * (int)o;
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Значение x равно: 10
Значение x в квадрате равно: 100
```

В данном примере значение переменной x автоматически упаковывается при передаче методу Sqr().

Упаковка и распаковка позволяют полностью унифицировать систему типов в C#. Благодаря тому что все типы являются производными от класса object, ссылка на значение любого типа может быть просто присвоена переменной ссылочного типа object, а все остальное возьмут на себя упаковка и распаковка. Более того, методы класса object оказываются доступными всем типам, поскольку они являются производными от этого класса. В качестве примера рассмотрим довольно любопытную программу.

```
// Благодаря упаковке становится возможным вызов методов по значению!

using System;

class MethOnValue {
  static void Main() {
    Console.WriteLine(10.ToString());
  }
}
```

В результате выполнения этой программы выводится значение 10. Дело в том, что метод `ToString()` возвращает строковое представление объекта, для которого он вызывается. В данном случае строковым представлением значения 10 как вызывающего объекта является само значение 10!

Класс `object` как универсальный тип данных

Если `object` является базовым классом для всех остальных типов и упаковка значений простых типов происходит автоматически, то класс `object` можно вполне использовать в качестве "универсального" типа данных. Для примера рассмотрим программу, в которой сначала создается массив типа `object`, элементам которого затем присваиваются значения различных типов данных.

```
// Использовать класс object для создания массива "обобщенного" типа.

using System;

class GenericDemo {
  static void Main() {
    object[] ga = new object[10];

    // Сохранить целые значения.
    for(int i=0; i < 3; i++)
      ga[i] = i;

    // Сохранить значения типа double.
    for(int i=3; i < 6; i++)
    ga[i] = (double) i / 2;

    // Сохранить две строки, а также значения типа bool и char.
    ga[6] = "Привет";
    ga[7] = true;
    ga[8] = 'X';
    ga[9] = "Конец";

    for(int i = 0; i < ga.Length; i++)
    Console.WriteLine("ga[" + i + "]: " + ga[i] + " ");
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
ga[0]: 0
ga[1]: 1
```

```
ga[2]: 2
ga[3]: 1.5
ga[4]: 2
ga[5]: 2.5
ga[6]: Привет
ga[7]: True
ga[8]: X
ga[9]: Конец
```

Как показывает данный пример, по ссылке на объект класса `object` можно обращаться к данным любого типа, поскольку в переменной ссылочного типа `object` допускается хранить ссылку на данные всех остальных типов. Следовательно, в массиве типа `object` из рассматриваемого здесь примера можно сохранить данные практически любого типа. В развитие этой идеи можно было бы, например, без особого труда создать класс стека со ссылками на объекты класса `object`. Это позволило бы хранить в стеке данные любого типа.

Несмотря на то что универсальный характер класса `object` может быть довольно эффективно использован в некоторых ситуациях, было бы ошибкой думать, что с помощью этого класса стоит пытаться обойти строго соблюдаемый в C# контроль типов. Вообще говоря, целое значение следует хранить в переменной типа `int`, строку — в переменной ссылочного типа `string` и т.д.

А самое главное, что начиная с версии 2.0 для программирования на C# стали доступными подлинно обобщенные типы данных — обобщения (более подробно они рассматриваются в главе 18). Внедрение обобщений позволило без труда определять классы и алгоритмы, автоматически обрабатывающие данные разных типов, соблюдая типовую безопасность. Благодаря обобщениям отпала необходимость пользоваться классом `object` как универсальным типом данных при создании нового кода. Универсальный характер этого класса лучше теперь оставить для применения в особых случаях.

12 ГЛАВА

Интерфейсы, структуры и перечисления

В этой главе рассматривается одно из самых важных в C# средств: *интерфейс*, определяющий ряд методов для реализации в классе. Но поскольку в самом интерфейсе ни один из методов не реализуется, интерфейс представляет собой чисто логическую конструкцию, описывающую функциональные возможности без конкретной их реализации.

Кроме того, в этой главе представлены еще два типа данных C#: структуры и перечисления. *Структуры* подобны классам, за исключением того, что они трактуются как типы значений, а не ссылочные типы. А *перечисления* представляют собой перечни целочисленных констант. Структуры и перечисления расширяют богатый арсенал средств программирования на C#.

Интерфейсы

Иногда в объектно-ориентированном программировании полезно определить, что именно должен делать класс, но не как он должен это делать. Примером тому может служить упоминавшийся ранее абстрактный метод. В абстрактном методе определяются возвращаемый тип и сигнатура метода, но не предоставляется его реализация. А в производном классе должна быть обеспечена своя собственная реализация каждого абстрактного метода, определенного в его базовом классе. Таким образом, абстрактный метод определяет *интерфейс*, но не реализацию метода. Конечно, абстрактные классы и методы приносят известную пользу, но положенный в их основу принцип может быть

развит далее. В C# предусмотрено разделение интерфейса класса и его реализации с помощью ключевого слова `interface`.

С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Но в интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс — множество методов.

Интерфейсы объявляются с помощью ключевого слова `interface`. Ниже приведена упрощенная форма объявления интерфейса.

```
interface имя{
  возвращаемый_тип имя_метода1(список_параметров);
  возвращаемый_тип имя_метода2(список_параметров);
  // ...
  возвращаемый_тип имя_методаN(список_параметров);
}
```

где *имя* — это конкретное имя интерфейса. В объявлении методов интерфейса используются только их *возвращаемый_тип* и сигнатура. Они, по существу, являются абстрактными методами. Как пояснялось выше, в интерфейсе не может быть никакой реализации. Поэтому все методы интерфейса должны быть реализованы в каждом классе, включающем в себя этот интерфейс. В самом же интерфейсе методы неявно считаются открытыми, поэтому доступ к ним не нужно указывать явно.

Ниже приведен пример объявления интерфейса для класса, генерирующего последовательный ряд чисел.

```
public interface ISeries {
  int GetNext(); // возвратить следующее по порядку число
  void Reset(); // перезапустить
  void SetStart(int x); // задать начальное значение
}
```

Этому интерфейсу присваивается имя `ISeries`. Префикс `I` в имени интерфейса указывать необязательно, но это принято делать в практике программирования, чтобы как-то отличать интерфейсы от классов. Интерфейс `ISeries` объявляется как `public` и поэтому может быть реализован в любом классе какой угодно программы.

Помимо методов, в интерфейсах можно также указывать свойства, индексаторы и события. Подробнее о событиях речь пойдет в главе 15, а в этой главе основное внимание будет уделено методам, свойствам и индексаторам. Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы. Кроме того, ни один из членов интерфейса не может быть объявлен как `static`.

Реализация интерфейсов

Как только интерфейс будет определен, он может быть реализован в одном или нескольких классах. Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе.

```
class имя_класса : имя_интерфейса {
  // тело класса
}
```

где *имя_интерфейса* — это конкретное имя реализуемого интерфейса. Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя.

В классе допускается реализовывать несколько интерфейсов. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс. В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Методы, реализующие интерфейс, должны быть объявлены как `public`. Дело в том, что в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой. Кроме того, возвращаемый тип и сигнатура реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, указанным в определении интерфейса.

Ниже приведен пример программы, в которой реализуется представленный ранее интерфейс `ISeries`. В этой программе создается класс `ByTwos`, генерирующий последовательный ряд чисел, в котором каждое последующее число на два больше предыдущего.

```
// Реализовать интерфейс ISeries.
class ByTwos : ISeries {
  int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int GetNext() {
    val += 2;
    return val;
  }

  public void Reset() {
    val = start;
}

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}
```

Как видите, в классе ByTwos реализуются три метода, определяемых в интерфейсе ISeries. Как пояснялось выше, это приходится делать потому, что в классе нельзя реализовать интерфейс частично.

Ниже приведен код класса, в котором демонстрируется применение класса ByTwos, реализующего интерфейс ISeries.

```
// Продемонстрировать применение класса ByTwos, реализующего интерфейс.

using System;

class SeriesDemo {
  static void Main() {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.GetNext());

    Console.WriteLine("\nСбросить");
    ob.Reset();
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.GetNext());

    Console.WriteLine("\nНачать с числа 100");
    ob.SetStart(100);
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.GetNext());
  }
}
```

Для того чтобы скомпилировать код класса SeriesDemo, необходимо включить в компиляцию файлы, содержащие интерфейс ISeries, а также классы ByTwos и SeriesDemo. Компилятор автоматически скомпилирует все три файла и сформирует из них окончательный исполняемый файл. Так, если эти файлы называются ISeries.cs, ByTwos.cs и SeriesDemo.cs, то программа будет скомпилирована в следующей командной строке:

```
>csc SeriesDemo.cs ISeries.cs ByTwos.cs
```

В интегрированной среде разработки Visual Studio для этой цели достаточно ввести все три упомянутых выше файла в конкретный проект C#. Кроме того, все три компилируемых элемента (интерфейс и оба класса) допускается включать в единый файл.

Ниже приведен результат выполнения скомпилированного кода.

```
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
Следующее число равно 10

Сбросить.
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
Следующее число равно 10
```

```
Начать с числа 100.
Следующее число равно 102
Следующее число равно 104
Следующее число равно 106
Следующее число равно 108
Следующее число равно 110
```

В классах, реализующих интерфейсы, разрешается и часто практикуется определять их собственные дополнительные члены. В качестве примера ниже приведен другой вариант класса `ByTwos`, в который добавлен метод `GetPrevious()`, возвращающий предыдущее значение.

```
// Реализовать интерфейс ISeries и добавить в
// класс ByTwos метод GetPrevious().

class ByTwos : ISeries {
  int start;
  int val;
  int prev;

  public ByTwos() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public int GetNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void Reset() {
    val = start;
    prev = start - 2;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
    prev = val - 2;
  }

  // Метод, не указанный в интерфейсе ISeries.
  public int GetPrevious() {
    return prev;
  }
}
```

Как видите, для того чтобы добавить метод `GetPrevious()`, потребовалось внести изменения в реализацию методов, определяемых в интерфейсе `ISeries`. Но поскольку интерфейс для этих методов остается прежним, то такие изменения не вызывают никаких осложнений и не нарушают уже существующий код. В этом и заключается одно из преимуществ интерфейсов.

Как пояснялось выше, интерфейс может быть реализован в любом количестве классов. В качестве примера ниже приведен класс `Primes`, генерирующий ряд простых чисел. Обратите внимание на то, реализация интерфейса `ISeries` в этом классе коренным образом отличается от той, что предоставляется в классе `ByTwos`.

```
// Использовать интерфейс ISeries для реализации
// процесса генерирования простых чисел.
class Primes : ISeries {
  int start;
  int val;

  public Primes() {
    start = 2;
    val = 2;
  }

  public int GetNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j <= i/j; j++) {
        if((i%j)==0) {
          isprime = false;
          break;
        }
      }
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}
```

Самое любопытное, что в обоих классах, `ByTwos` и `Primes`, реализуется один и тот же интерфейс, несмотря на то, что в них генерируются совершенно разные ряды чисел. Как пояснялось выше, в интерфейсе вообще отсутствует какая-либо реализация, поэтому он может быть свободно реализован в каждом классе так, как это требуется для самого класса.

Применение интерфейсных ссылок

Как это ни покажется странным, но в C# допускается объявлять переменные ссылочного интерфейсного типа, т.е. переменные ссылки на интерфейс. Такая переменная может ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки выполняется его вариант, реализованный в классе данного объекта. Этот процесс аналогичен применению ссылки на базовый класс для доступа к объекту производного класса, как пояснялось в главе 11.

В приведенном ниже примере программы демонстрируется применение интерфейсной ссылки. В этой программе переменная ссылки на интерфейс используется с целью вызвать методы для объектов обоих классов — `ByTwos` и `Primes`. Для ясности в данном примере показаны все части программы, собранные в единый файл.

```
// Продемонстрировать интерфейсные ссылки.

using System;

// Определить интерфейс.
public interface ISeries {
  int GetNext(); // возвратить следующее по порядку число
   void Reset(); // перезапустить
   void SetStart(int x); // задать начальное значение
}

// Использовать интерфейс ISeries для реализации процесса
// генерирования последовательного ряда чисел, в котором каждое
// последующее число на два больше предыдущего.
class ByTwos : ISeries {
  int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int GetNext() {
    val += 2;
   return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}

// Использовать интерфейс ISeries для реализации
// процесса генерирования простых чисел.
```

```
class Primes : ISeries {
  int start;
  int val;

  public Primes() {
    start = 2;
    val = 2;
  }

  public int GetNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j <= i/j; j++) {
        if((i%j)==0) {
          isprime = false;
          break;
        }
      }
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(int x) {
    start = x;
    val = start;
  }
}

class SeriesDemo2 {
  static void Main() {
    ByTwos twoOb = new ByTwos();
    Primes primeOb = new Primes();
    ISeries ob;

    for(int i=0; i < 5; i++) {
      ob = twoOb;
      Console.WriteLine("Следующее четное число равно " + ob.GetNext());

      ob = primeOb;
      Console.WriteLine("Следующее простое число " + "равно " + ob.GetNext());
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы:

```
Следующее четное число равно 2
Следующее простое число равно 3
Следующее четное число равно 4
Следующее простое число равно 5
Следующее четное число равно 6
Следующее простое число равно 7
Следующее четное число равно 8
Следующее простое число равно 11
Следующее четное число равно 10
Следующее простое число равно 13
```

В методе `Main()` переменная `ob` объявляется для ссылки на интерфейс `ISeries`. Это означает, что в ней могут храниться ссылки на объект любого класса, реализующего интерфейс `ISeries`. В данном случае она служит для ссылки на объекты `twoOb` и `primeOb` классов `ByTwos` и `Primes` соответственно, в которых реализован интерфейс `ISeries`.

И еще одно замечание: переменной ссылки на интерфейс доступны только методы, объявленные в ее интерфейсе. Поэтому интерфейсную ссылку нельзя использовать для доступа к любым другим переменным и методам, которые не поддерживаются объектом класса, реализующего данный интерфейс.

Интерфейсные свойства

Аналогично методам, свойства указываются в интерфейсе вообще без тела. Ниже приведена общая форма объявления интерфейсного свойства.

```
// Интерфейсное свойство
тип имя{
  get;
  set;
}
```

Очевидно, что в определении интерфейсных свойств, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: `get` или `set` соответственно.

Несмотря на то что объявление свойства в интерфейсе очень похоже на объявление автоматически реализуемого свойства в классе, между ними все же имеется отличие. При объявлении в интерфейсе свойство не становится автоматически реализуемым. В этом случае указывается только имя и тип свойства, а его реализация предоставляется каждому реализующему классу. Кроме того, при объявлении свойства в интерфейсе не разрешается указывать модификаторы доступа для аксессоров. Например, аксессор `set` не может быть указан в интерфейсе как `private`.

Ниже в качестве примера приведен переделанный вариант интерфейса `ISeries` и класса `ByTwos`, в котором свойство `Next` используется для получения и установки следующего по порядку числа, которое больше предыдущего на два.

```
// Использовать свойство в интерфейсе.

using System;
```

```
public interface ISeries {
  // Интерфейсное свойство.
  int Next {
    get; // возвратить следующее по порядку число
    set; // установить следующее число
  }
}

// Реализовать интерфейс ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
}

  // Получить или установить значение.
  public int Next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }
}

// Продемонстрировать применение интерфейсного свойства.
class SeriesDemo3 {
  static void Main() {
    ByTwos ob = new ByTwos();

    // Получить доступ к последовательному ряду чисел с помощью свойства.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.Next);

    Console.WriteLine("\nНачать с числа 21");
    ob.Next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.Next);
  }
}
```

При выполнении этого кода получается следующий результат.

```
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
Следующее число равно 10

Начать с числа 21
Следующее число равно 23
Следующее число равно 25
```

```
Следующее число равно 27
Следующее число равно 29
Следующее число равно 31
```

Интерфейсные индексаторы

В интерфейсе можно также указывать индексаторы. Ниже приведена общая форма объявления интерфейсного индексатора.

```
// Интерфейсный индексатор
тип_элемента this[int индекс]{
  get;
  set;
}
```

Как и прежде, в объявлении интерфейсных индексаторов, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: `get` или `set` соответственно.

Ниже в качестве примера приведен еще один вариант реализации интерфейса `ISeries`, в котором добавлен индексатор только для чтения, возвращающий *i-й* элемент числового ряда.

```
// Добавить индексатор в интерфейс.

using System;

public interface ISeries {
  // Интерфейсное свойство.
  int Next {
    get; // возвратить следующее по порядку число
    set; // установить следующее число
  }

  // Интерфейсный индексатор.
  int this[int index] {
    get; // возвратить указанное в ряду число
  }
}

// Реализовать интерфейс ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
  }

  // Получить или установить значение с помощью свойства.
  public int Next {
    get {
      val += 2;
      return val;
    }
```

```
    set {
      val = value;
    }
  }

  // Получить значение по индексу.
  public int this[int index] {
    get {
      val = 0;
      for(int i=0; i < index; i++)
        val += 2;
      return val;
    }
  }
}

// Продемонстрировать применение интерфейсного индексатора.
class SeriesDemo4 {
  static void Main() {
    ByTwos ob = new ByTwos();

    // Получить доступ к последовательному ряду чисел с помощью свойства.
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.Next);

    Console.WriteLine("\nНачать с числа 21");
    ob.Next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob.Next);

    Console.WriteLine("\nСбросить в 0");
    ob.Next = 0;

    // Получить доступ к последовательному ряду чисел с помощью индексатора
    for(int i=0; i < 5; i++)
      Console.WriteLine("Следующее число равно " + ob[i]);
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
Следующее число равно 10

Начать с числа 21
Следующее число равно 23
Следующее число равно 25
Следующее число равно 27
Следующее число равно 29
Следующее число равно 31
```

```
Сбросить в 0
Следующее число равно 0
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
```

Наследование интерфейсов

Один интерфейс может наследовать другой. Синтаксис наследования интерфейсов такой же, как и у классов. Когда в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов, как в приведенном ниже примере.

```
// Пример наследования интерфейсов.

using System;

public interface IA {
  void Meth1();
  void Meth2();
}

// В базовый интерфейс включены методы Meth1() и Meth2(),
// а в производный интерфейс добавлен еще один метод — Meth3().
public interface IB : IA {
  void Meth3();
}

// В этом классе должны быть реализованы все методы интерфейсов IA и IB.
class MyClass : IB {
  public void Meth1() {
    Console.WriteLine("Реализовать метод Meth1().");
  }

  public void Meth2() {
    Console.WriteLine("Реализовать метод Meth2().");
  }

  public void Meth3() {
    Console.WriteLine("Реализовать метод Meth3().");
  }
}

class IFExtend {
  static void Main() {
    MyClass ob = new MyClass();

    ob.Meth1();
    ob.Meth2();
    ob.Meth3();
  }
}
```

Ради интереса попробуйте удалить реализацию метода `Meth1()` из класса `MyClass`. Это приведет к ошибке во время компиляции. Как пояснялось ранее, в любом классе, реализующем интерфейс, должны быть реализованы все методы, определенные в этом интерфейсе, в том числе и те, что наследуются из других интерфейсов.

Сокрытие имен при наследовании интерфейсов

Когда один интерфейс наследует другой, то в производном интерфейсе может быть объявлен член, скрывающий член с аналогичным именем в базовом интерфейсе. Такое сокрытие имен происходит в том случае, если член в производном интерфейсе объявляется таким же образом, как и в базовом интерфейсе. Но если не указать в объявлении члена производного интерфейса ключевое слово `new`, то компилятор выдаст соответствующее предупреждающее сообщение.

Явные реализации

При реализации члена интерфейса имеется возможность указать его имя *полностью* вместе с именем самого интерфейса. В этом случае получается *явная реализация члена интерфейса*, или просто *явная реализация*. Так, если объявлен интерфейс `IMyIF`

```
interface IMyIF {
  int MyMeth(int x);
}
```

то следующая его реализация считается вполне допустимой:

```
class MyClass : IMyIF {
  int IMyIF.MyMeth(int x) {
    return x / 3;
  }
}
```

Как видите, при реализации члена `MyMeth()` интерфейса `IMyIF` указывается его полное имя, включающее в себя имя его интерфейса.

Для явной реализации интерфейсного метода могут быть две причины. Во-первых, когда интерфейсный метод реализуется с указанием его полного имени, то такой метод оказывается доступным не посредством объектов класса, реализующего данный интерфейс, а по интерфейсной ссылке. Следовательно, явная реализация позволяет реализовать интерфейсный метод таким образом, чтобы он *не* стал открытым членом класса, предоставляющего его реализацию. И во-вторых, в одном классе могут быть реализованы два интерфейса с методами, объявленными с одинаковыми именами и сигнатурами. Но неоднозначность в данном случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов. Рассмотрим каждую из этих двух возможностей явной реализации на конкретных примерах.

В приведенном ниже примере программы демонстрируется интерфейс `IEven`, в котором объявляются два метода: `IsEven()` и `IsOdd()`. В первом из них определяется четность числа, а во втором — его нечетность. Интерфейс `IEven` затем реализуется в классе `MyClass`. При этом метод `IsOdd()` реализуется явно.

```
// Реализовать член интерфейса явно.

using System;

interface IEven {
  bool IsOdd(int x);
  bool IsEven(int x);
}

class MyClass : IEven {
  // Явная реализация. Обратите внимание на то, что
  // этот член является закрытым по умолчанию.
  bool IEven.IsOdd(int x) {
    if((x%2) != 0) return true;
    else return false;
  }

  // Обычная реализация.
  public bool IsEven(int x) {
    IEven o = this; // Интерфейсная ссылка на вызывающий объект.
    return !o.IsOdd(x);
  }
}

class Demo {
  static void Main() {
    MyClass ob = new MyClass();
    bool result;

    result = ob.IsEven(4);
    if(result) Console.WriteLine("4 — четное число.");

    // result = ob.IsOdd(4); // Ошибка, член IsOdd интерфейса IEven недоступен

    // Но следующий код написан верно, поскольку в нем сначала создается
    // интерфейсная ссылка типа IEven на объект класса MyClass, а затем по
    // этой ссылке вызывается метод IsOdd().
    IEven iRef = (IEven) ob;
    result = iRef.IsOdd(3);
    if(result) Console.WriteLine("3 — нечетное число.");
  }
}
```

В приведенном выше примере метод `IsOdd()` реализуется явно, а значит, он недоступен как открытый член класса `MyClass`. Напротив, он доступен только по интерфейсной ссылке. Именно поэтому он вызывается посредством переменной `o` ссылочного типа `IEven` в реализации метода `IsEven()`.

Ниже приведен пример программы, в которой реализуются два интерфейса, причем в обоих интерфейсах объявляется метод `Meth()`. Благодаря явной реализации исключается неоднозначность, характерная для подобной ситуации.

```
// Воспользоваться явной реализацией для устранения неоднозначности.

using System;
```

```
interface IMyIF_A {
  int Meth(int x);
}

interface IMyIF_B {
  int Meth(int x);
}

// Оба интерфейса реализуются в классе MyClass.
class MyClass : IMyIF_A, IMyIF_B {
  // Реализовать оба метода Meth() явно.
  int IMyIF_A.Meth(int x) {
    return x + x;
  }

  int IMyIF_B.Meth(int x) {
    return x * x;
  }

  // Вызывать метод Meth() по интерфейсной ссылке.
  public int MethA(int x){
    IMyIF_A a_ob;
    a_ob = this;
    return a_ob.Meth(x); // вызов интерфейсного метода IMyIF_A
  }

  public int MethB(int x){
    IMyIF_B b_ob;
    b_ob = this;
    return b_ob.Meth(x); // вызов интерфейсного метода IMyIF_B
  }
}

class FQIFNames {
  static void Main() {
    MyClass ob = new MyClass();

    Console.Write("Вызов метода IMyIF_A.Meth(): ");
    Console.WriteLine(ob.MethA(3));

    Console.Write("Вызов метода IMyIF_B.Meth(): ");
    Console.WriteLine(ob.MethB(3));
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Вызов метода IMyIF_A.Meth(): 6
Вызов метода IMyIF_B.Meth(): 9
```

Анализируя приведенный выше пример программы, обратим прежде всего внимание на одинаковую сигнатуру метода `Meth()` в обоих интерфейсах, `IMyIF_A` и `IMyIF_B`. Когда оба этих интерфейса реализуются в классе `MyClass`, для каждого из них в отдельности это делается явно, т.е. с указанием полного имени метода `Meth()`. А поскольку явно реализованный метод может вызываться только по интерфейсной

ссылке, то в классе `MyClass` создаются две такие ссылки: одна — для интерфейса `IMyIF_A`, а другая — для интерфейса `IMyIF_B`. Именно по этим ссылкам происходит обращение к объектам данного класса с целью вызвать методы соответствующих интерфейсов, благодаря чему и устраняется неоднозначность.

Выбор между интерфейсом и абстрактным классом

Одна из самых больших трудностей программирования на C# состоит в правильном выборе между интерфейсом и абстрактным классом в тех случаях, когда требуется описать функциональные возможности, но не реализацию. В подобных случаях рекомендуется придерживаться следующего общего правила: если какое-то понятие можно описать с точки зрения функционального назначения, не уточняя конкретные детали реализации, то следует использовать интерфейс. А если требуются некоторые детали реализации, то данное понятие следует представить абстрактным классом.

Стандартные интерфейсы для среды .NET Framework

Для среды .NET Framework определено немало стандартных интерфейсов, которыми можно пользоваться в программах на C#. Так, в интерфейсе `System.IComparable` определен метод `CompareTo()`, применяемый для сравнения объектов, когда требуется соблюдать отношение порядка. Стандартные интерфейсы являются также важной частью классов коллекций, предоставляющих различные средства, в том числе стеки и очереди, для хранения целых групп объектов. Так, в интерфейсе `System.Collections.ICollection` определяются функции для всей коллекции, а в интерфейсе `System.Collections.IEnumerator` — способ последовательного обращения к элементам коллекции. Эти и многие другие интерфейсы подробнее рассматриваются в части II данной книги.

Структуры

Как вам должно быть уже известно, классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно. Но иногда прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти. Для разрешения подобных затруднений в C# предусмотрена *структура*, которая подобна классу, но относится к типу значения, а не к ссылочному типу данных.

Структуры объявляются с помощью ключевого слова `struct` и с точки зрения синтаксиса подобны классам. Ниже приведена общая форма объявления структуры:

```
struct имя : интерфейсы {
  // объявления членов
}
```

где *имя* обозначает конкретное имя структуры.

Одни структуры не могут наследовать другие структуры и классы или служить в качестве базовых для других структур и классов. (Разумеется, структуры, как и все остальные типы данных в C#, наследуют класс `object`.) Тем не менее в структуре можно реализовать один или несколько интерфейсов, которые указываются после имени структуры списком через запятую. Как и у классов, у каждой структуры имеются свои члены: методы, поля, индексаторы, свойства, операторные методы и события. В структурах допускается также определять конструкторы, но не деструкторы. В то же время для структуры нельзя определить конструктор, используемый по умолчанию (т.е. конструктор без параметров). Дело в том, что конструктор, вызываемый по умолчанию, определяется для всех структур автоматически и не подлежит изменению. Такой конструктор инициализирует поля структуры значениями, задаваемыми по умолчанию. А поскольку структуры не поддерживают наследование, то их члены нельзя указывать как `abstract`, `virtual` или `protected`.

Объект структуры может быть создан с помощью оператора `new` таким же образом, как и объект класса, но в этом нет особой необходимости. Ведь когда используется оператор `new`, то вызывается конструктор, используемый по умолчанию. А когда этот оператор не используется, объект по-прежнему создается, хотя и не инициализируется. В этом случае инициализацию любых членов структуры придется выполнить вручную.

В приведенном ниже примере программы демонстрируется применение структуры для хранения информации о книге.

```
// Продемонстрировать применение структуры.

using System;

// Определить структуру.
struct Book {
  public string Author;
  public string Title;
  public int Copyright;

  public Book(string a, string t, int c) {
    Author = a;
    Title = t;
    Copyright = c;
  }
}

// Продемонстрировать применение структуры Book.
class StructDemo {
  static void Main() {
    Book book1 = new Book("Герберт Шилдт",
                          "Полный справочник по C# 4.0",
                          2010); // вызов явно заданного конструктора
    Book book2 = new Book(); // вызов конструктора по умолчанию
    Book book3; // конструктор не вызывается

    Console.WriteLine(book1.Author + ", " +
                      book1.Title + ", (c) " + book1.Copyright);
    Console.WriteLine();

    if(book2.Title == null)
```

```
      Console.WriteLine("Член book2.Title пуст.");

    // А теперь ввести информацию в структуру book2.
     book2.Title = "О дивный новый мир";
     book2.Author = "Олдос Хаксли";
     book2.Copyright = 1932;
     Console.Write("Структура book2 теперь содержит:\n");
     Console.WriteLine(book2.Author + ", " +
                       book2.Title + ", (c) " + book2.Copyright);
     Console.WriteLine();

   // Console.WriteLine(book3.Title); // неверно, этот член структуры
                                       // нужно сначала инициализировать
     book3.Title = "Красный шторм";

     Console.WriteLine(book3.Title); // теперь верно
   }
}
```

При выполнении этой программы получается следующий результат.

```
Герберт Шилдт, Полный справочник по C# 4.0, (c) 2010

Член book2.Title пуст.
Структура book2 теперь содержит:
Олдос Хаксли, О дивный новый мир, (c) 1932

Красный шторм
```

Как демонстрирует приведенный выше пример программы, структура может быть инициализирована с помощью оператора `new` для вызова конструктора или же путем простого объявления объекта. Так, если используется оператор `new`, то поля структуры инициализируются конструктором, вызываемым по умолчанию (в этом случае во всех полях устанавливается задаваемое по умолчанию значение), или же конструктором, определяемым пользователем. А если оператор `new` не используется, как это имеет место для структуры `book3`, то объект структуры не инициализируется, а его поля должны быть установлены вручную перед тем, как пользоваться данным объектом.

Когда одна структура присваивается другой, создается копия ее объекта. В этом заключается одно из главных отличий структуры от класса. Как пояснялось ранее в этой книге, когда ссылка на один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части. А когда переменная одной структуры присваивается переменной другой структуры, создается *копия* объекта структуры из правой части оператора присваивания. Рассмотрим в качестве примера следующую программу.

```
// Скопировать структуру.

using System;

// Определить структуру.
struct MyStruct {
  public int x;
}
```

```
// Продемонстрировать присваивание структуры.
class StructAssignment {
  static void Main() {
    MyStruct a;
    MyStruct b;

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
a.x 10, b.x 20
a.x 20, b.x 30
```

Как показывает приведенный выше результат, после присваивания

```
a = b;
```

переменные структуры a и b по-прежнему остаются совершенно обособленными, т.е. переменная a не указывает на переменную b и никак не связана с ней, помимо того, что она содержит копию значения переменной b. Ситуация была бы совсем иной, если бы переменные a и b были ссылочного типа, указывая на объекты определенного класса. В качестве примера ниже приведен вариант предыдущей программы, где демонстрируется присваивание переменных ссылки на объекты определенного класса.

```
// Использовать ссылки на объекты определенного класса.

using System;

// Создать класс.
class MyClass {
  public int x;
  }

// Показать присваивание разных объектов данного класса.
class ClassAssignment {
  static void Main() {
    MyClass a = new MyClass();
    MyClass b = new MyClass();

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;
```

```
    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
a.x 10, b.x 20
a.x 30, b.x 30
```

Как видите, после того как переменная b будет присвоена переменной a, обе переменные станут указывать на один и тот же объект, т.е. на тот объект, на который первоначально указывала переменная b.

О назначении структур

В связи с изложенным выше возникает резонный вопрос: зачем в C# включена структура, если она обладает более скромными возможностями, чем класс? Ответ на этот вопрос заключается в повышении эффективности и производительности программ. Структуры относятся к типам значений, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти. Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам — по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки. И вообще, если нужно просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке, то с точки зрения производительности для них лучше выбрать структуру.

Ниже приведен еще один пример, демонстрирующий применение структуры на практике. В этом примере из области электронной коммерции имитируется запись транзакции. Каждая такая транзакция включает в себя заголовок пакета, содержащий номер и длину пакета. После заголовка следует номер счета и сумма транзакции. Заголовок пакета представляет собой самостоятельную единицу информации, и поэтому он организуется в отдельную структуру, которая затем используется для создания записи транзакции или же информационного пакета любого другого типа.

```
// Структуры удобны для группирования небольших объемов данных.

using System;

// Определить структуру пакета.
struct PacketHeader {
  public uint PackNum; // номер пакета
  public ushort PackLen; // длина пакета
  }

// Использовать структуру PacketHeader для создания записи транзакции
// в сфере электронной коммерции.
class Transaction {
  static uint transacNum = 0;
```

```
  PacketHeader ph; // ввести структуру PacketHeader в класс Transaction
  string accountNum;
  double amount;

  public Transaction(string acc, double val) {
    // создать заголовок пакета
    ph.PackNum = transacNum++;
    ph.PackLen = 512; // произвольная длина

    accountNum = acc;
    amount = val;
  }

  // Сымитировать транзакцию.
  public void sendTransaction() {
    Console.WriteLine("Пакет #: " + ph.PackNum +
                      ", Длина: " + ph.PackLen +
                      ",\n Счет #: " + accountNum +
                      ", Сумма: {0:C}\n", amount);
  }
}

// Продемонстрировать применение структуры в виде пакета транзакции.
class PacketDemo {
  static void Main() {
    Transaction t = new Transaction("31243", -100.12);
    Transaction t2 = new Transaction("AB4655", 345.25);
    Transaction t3 = new Transaction("8475-09", 9800.00);

    t.sendTransaction();
    t2.sendTransaction();
    t3.sendTransaction();
  }
}
```

Вот к какому результату может привести выполнение этого кода.

```
Пакет #: 0, Длина: 512,
    Счет #: 31243, Сумма: ($100.12)

Пакет #: 1, Длина: 512,
    Счет #: AB4655, Сумма: $345.25

Пакет #: 2, Длина: 512,
    Счет #: 8475-09, Сумма: $9,800.00
```

Структура `PacketHeader` оказывается вполне пригодной для формирования заголовка пакета транзакции, поскольку в ней хранится очень небольшое количество данных, не используется наследование и даже не содержатся методы. Кроме того, работа со структурой `PacketHeader` не влечет за собой никаких дополнительных издержек, связанных со ссылками на объекты, что весьма характерно для класса. Следовательно, структуру `PacketHeader` можно использовать для записи любой транзакции, не снижая эффективность данного процесса.

Любопытно, что в C++ также имеются структуры и используется ключевое слово `struct`. Но эти структуры отличаются от тех, что имеются в C#. Так, в C++ структура относится к типу класса, а значит, структура и класс в этом языке практически равноценны и отличаются друг от друга лишь доступом по умолчанию к их членам, которые оказываются закрытыми для класса и открытыми для структуры. А в C# структура относится к типу значения, тогда как класс — к ссылочному типу.

Перечисления

Перечисление представляет собой множество именованных целочисленных констант. Перечислимый тип данных объявляется с помощью ключевого слова `enum`. Ниже приведена общая форма объявления перечисления:

```
enum имя {список_перечисления};
```

где *имя* — это имя типа перечисления, а *список_перечисления* — список идентификаторов, разделяемый запятыми.

В приведенном ниже примере объявляется перечисление `Apple` различных сортов яблок.

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap,
             Cortland, McIntosh };
```

Следует особо подчеркнуть, что каждая символически обозначаемая константа в перечислении имеет целое значение. Тем не менее неявные преобразования перечислимого типа во встроенные целочисленные типы и обратно в C# не определены, а значит, в подобных случаях требуется явное приведение типов. Кроме того, приведение типов требуется при преобразовании двух перечислимых типов. Но поскольку перечисления обозначают целые значения, то их можно, например, использовать для управления оператором выбора `switch` или же оператором цикла `for`.

Для каждой последующей символически обозначаемой константы в перечислении задается целое значение, которое на единицу больше, чем у предыдущей константы. По умолчанию значение первой символически обозначаемой константы в перечислении равно нулю. Следовательно, в приведенном выше примере перечисления `Apple` константа `Jonathan` равна нулю, константа `GoldenDel` — 1, константа `RedDel` — 2 и т.д.

Доступ к членам перечисления осуществляется по имени их типа, после которого следует оператор-точка. Например, при выполнении фрагмента кода

```
Console.WriteLine(Apple.RedDel + " имеет значение " +
                  (int)Apple.RedDel);
```

выводится следующий результат.

```
RedDel имеет значение 2
```

Как показывает результат выполнения приведенного выше фрагмента кода, для вывода перечислимого значения используется его имя. Но для получения этого значения требуется предварительно привести его к типу `int`.

Ниже приведен пример программы, демонстрирующий применение перечисления `Apple`.

```
// Продемонстрировать применение перечисления.

using System;

class EnumDemo {
  enum Apple { Jonathan, GoldenDel, RedDel, Winesap,
               Cortland, McIntosh };

  static void Main() {
    string[] color = {
      "красный",
      "желтый",
      "красный",
      "красный",
      "красный",
      "красновато-зеленый"
  };

  Apple i; // объявить переменную перечислимого типа

  // Использовать переменную i для циклического
  // обращения к членам перечисления.
  for(i = Apple.Jonathan; i <= Apple.McIntosh; i++)
    Console.WriteLine(i + " имеет значение " + (int)i);

  Console.WriteLine();

  // Использовать перечисление для индексирования массива.
  for(i = Apple.Jonathan; i <= Apple.McIntosh; i++)
    Console.WriteLine("Цвет сорта " + i + " — " +
                      color[(int)i]);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Jonathan имеет значение 0
GoldenDel имеет значение 1
RedDel имеет значение 2
Winsap имеет значение 3
Cortland имеет значение 4
McIntosh имеет значение 5

Цвет сорта Jonathan - красный
Цвет сорта GoldenDel - желтый
Цвет сорта RedDel - красный
Цвет сорта Winsap - красный
Цвет сорта Cortland - красный
Цвет сорта McIntosh - красновато-зеленый
```

Обратите внимание на то, как переменная типа `Apple` управляет циклами `for`. Значения символически обозначаемых констант в перечислении `Apple` начинаются с нуля, поэтому их можно использовать для индексирования массива, чтобы получить цвет каждого сорта яблок. Обратите также внимание на необходимость производить приведение типов, когда перечислимое значение используется для индексирования

массива. Как упоминалось выше, в C# не предусмотрены неявные преобразования перечислимых типов в целочисленные и обратно, поэтому для этой цели требуется явное приведение типов.

И еще одно замечание: все перечисления неявно наследуют от класса `System.Enum`, который наследует от класса `System.ValueType`, а тот, в свою очередь, — от класса `object`.

Инициализация перечисления

Значение одной или нескольких символически обозначаемых констант в перечислении можно задать с помощью инициализатора. Для этого достаточно указать после символического обозначения отдельной константы знак равенства и целое значение. Каждой последующей константе присваивается значение, которое на единицу больше значения предыдущей инициализированной константы. Например, в приведенном ниже фрагменте кода константе `RedDel` присваивается значение 10.

```
enum Apple { Jonathan, GoldenDel, RedDel = 10, Winesap,
             Cortland, McIntosh };
```

В итоге все константы в перечислении принимают приведенные ниже значения.

Jonathan	0
GoldenDel	1
RedDel	10
Winesap	11
Cortland	12
McIntosh	13

Указание базового типа перечисления

По умолчанию в качестве базового для перечислений выбирается тип `int`, тем не менее перечисление может быть создано любого целочисленного типа, за исключением `char`. Для того чтобы указать другой тип, кроме `int`, достаточно поместить этот тип после имени перечисления, отделив его двоеточием. В качестве примера ниже задается тип `byte` для перечисления `Apple`.

```
enum Apple : byte { Jonathan, GoldenDel, RedDel,
                    Winesap, Cortland, McIntosh };
```

Теперь константа `Apple.Winesap`, например, имеет количественное значение типа `byte`.

Применение перечислений

На первый взгляд перечисления могут показаться любопытным, но не очень нужным элементом C#, но на самом деле это не так. Перечисления очень полезны, когда в программе требуется одна или несколько специальных символически обозначаемых констант. Допустим, что требуется написать программу для управления лентой конвейера на фабрике. Для этой цели можно создать метод `Conveyor()`, принимающий в качестве параметров следующие команды: "старт", "стоп", "вперед" и "назад". Вместо того чтобы передавать методу `Conveyor()` целые значения, например, 1 — в качестве

команды "старт", 2 — в качестве команды "стоп" и так далее, что чревато ошибками, можно создать перечисление, чтобы присвоить этим значениям содержательные символические обозначения. Ниже приведен пример применения такого подхода.

```
// Сымитировать управление лентой конвейера.
using System;

class ConveyorControl {
  // Перечислить команды конвейера.
  public enum Action { Start, Stop, Forward, Reverse };

  public void Conveyor(Action com) {
    switch(com) {
      case Action.Start:
        Console.WriteLine("Запустить конвейер.");
        break;
      case Action.Stop:
        Console.WriteLine("Остановить конвейер.");
        break;
      case Action.Forward:
        Console.WriteLine("Переместить конвейер вперед.");
        break;
      case Action.Reverse:
        Console.WriteLine("Переместить конвейер назад.");
        break;
    }
  }
}

class ConveyorDemo {
  static void Main() {
    ConveyorControl c = new ConveyorControl();

    c.Conveyor(ConveyorControl.Action.Start);
    c.Conveyor(ConveyorControl.Action.Forward);
    c.Conveyor(ConveyorControl.Action.Reverse);
    c.Conveyor(ConveyorControl.Action.Stop);
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Запустить конвейер.
Переместить конвейер вперед.
Переместить конвейер назад.
Остановить конвейер.
```

Метод `Conveyor()` принимает аргумент типа `Action`, и поэтому ему могут быть переданы только значения, определяемые в перечислении `Action`. Например, ниже приведена попытка передать методу `Conveyor()` значение 22.

```
c.Conveyor(22); // Ошибка!
```

Эта строка кода не будет скомпилирована, поскольку отсутствует предварительно заданное преобразование типа `int` в перечислимый тип `Action`. Именно это и препятствует передаче неправильных команд методу `Conveyor()`. Конечно, такое

преобразование можно организовать принудительно с помощью приведения типов, но это было бы преднамеренным, а не случайным или неумышленным действием. Кроме того, вероятность неумышленной передачи пользователем неправильных команд методу `Conveyor()` сводится с минимуму благодаря тому, что эти команды обозначены символическими именами в перечислении.

В приведенном выше примере обращает на себя внимание еще одно интересное обстоятельство: перечислимый тип используется для управления оператором `switch`. Как упоминалось выше, перечисления относятся к целочисленным типам данных, и поэтому их вполне допустимо использовать в операторе `switch`.

13 ГЛАВА

Обработка исключительных ситуаций

Исключительная ситуация, или просто исключение, происходит во время выполнения. Используя подсистему обработки исключительных ситуаций в C#, можно обрабатывать структурированным и контролируемым образом ошибки, возникающие при выполнении программы. Главное преимущество обработки исключительных ситуаций заключается в том, что она позволяет автоматизировать получение большей части кода, который раньше приходилось вводить в любую крупную программу вручную для обработки ошибок. Так, если программа написана на языке программирования без обработки исключительных ситуаций, то при неудачном выполнении методов приходится возвращать коды ошибок, которые необходимо проверять вручную при каждом вызове метода. Это не только трудоемкий, но и чреватый ошибками процесс. Обработка исключительных ситуаций рационализирует весь процесс обработки ошибок, позволяя определить в программе блок кода, называемый *обработчиком исключений* и выполняющийся автоматически, когда возникает ошибка. Это избавляет от необходимости проверять вручную, насколько удачно или неудачно завершилась конкретная операция либо вызов метода. Если возникнет ошибка, она будет обработана соответствующим образом обработчиком ошибок.

Обработка исключительных ситуаций важна еще и потому, что в C# определены стандартные исключения для типичных программных ошибок, например деление на нуль или выход индекса за границы массива. Для реагирования на подобные ошибки в программе должно быть организовано отслеживание и обработка соответствующих

исключительных ситуаций. Ведь в конечном счете для успешного программирования на C# необходимо научиться умело пользоваться подсистемой обработки исключительных ситуаций.

Класс System.Exception

В C# исключения представлены в виде классов. Все классы исключений должны быть производными от встроенного в C# класса `Exception`, являющегося частью пространства имен `System`. Следовательно, все исключения являются подклассами класса `Exception`.

К числу самых важных подклассов `Exception` относится класс `SystemException`. Именно от этого класса являются производными все исключения, генерируемые исполняющей системой C# (т.е. системой CLR). Класс `SystemException` ничего не добавляет к классу `Exception`, а просто определяет вершину иерархии стандартных исключений.

В среде .NET Framework определено несколько встроенных исключений, являющихся производными от класса `SystemException`. Например, при попытке выполнить деление на нуль генерируется исключение `DivideByZeroException`. Как будет показано далее в этой главе, в C# можно создавать собственные классы исключений, производные от класса `Exception`.

Основы обработки исключительных ситуаций

Обработка исключительных ситуаций в C# организуется с помощью четырех ключевых слов: `try`, `catch`, `throw` и `finally`. Они образуют взаимосвязанную подсистему, в которой применение одного из ключевых слов подразумевает применение другого. На протяжении всей этой главы назначение и применение каждого из упомянутых выше ключевых слов будет рассмотрено во всех подробностях. Но прежде необходимо дать общее представление о роли каждого из них в обработке исключительных ситуаций. Поэтому ниже кратко описан принцип их действия.

Операторы программы, которые требуется контролировать на появление исключений, заключаются в блок `try`. Если внутри блока `try` возникает исключительная ситуация, *генерируется* исключение. Это исключение может быть перехвачено и обработано каким-нибудь рациональным способом в коде программы с помощью оператора, обозначаемого ключевым словом `catch`. Исключения, возникающие на уровне системы, генерируются исполняющей системой автоматически. А для генерирования исключений вручную служит ключевое слово `throw`. Любой код, который должен быть непременно выполнен после выхода из блока `try`, помещается в блок `finally`.

Применение пары ключевых слов try и catch

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов `try` и `catch`. Эти ключевые слова действуют совместно и не могут быть использованы порознь. Ниже приведена общая форма определения блоков `try/catch` для обработки исключительных ситуаций:

```
try {
// Блок кода, проверяемый на наличие ошибок.
}
```

```
catch (ExcepType1 exOb) {
// Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик исключения типа ExcepType2.
}
.
.
.
```

где *ExcepType* — это тип возникающей исключительной ситуации. Когда исключение генерируется оператором `try`, оно перехватывается составляющим ему пару оператором `catch`, который затем обрабатывает это исключение. В зависимости от типа исключения выполняется и соответствующий оператор `catch`. Так, если типы генерируемого исключения и того, что указывается в операторе `catch`, совпадают, то выполняется именно этот оператор, а все остальные пропускаются. Когда исключение перехватывается, переменная исключения *exOb* получает свое значение.

На самом деле указывать переменную *exOb* необязательно. Так, ее необязательно указывать, если обработчику исключений не требуется доступ к объекту исключения, что бывает довольно часто. Для обработки исключения достаточно и его типа. Именно поэтому во многих примерах программ, приведенных в этой главе, переменная *exOb* опускается.

Следует, однако, иметь в виду, что если исключение не генерируется, то блок оператора `try` завершается как обычно, и все его операторы `catch` пропускаются. Выполнение программы возобновляется с первого оператора, следующего после завершающего оператора `catch`. Таким образом, оператор `catch` выполняется лишь в том случае, если генерируется исключение.

Простой пример обработки исключительной ситуации

Рассмотрим простой пример, демонстрирующий отслеживание и перехватывание исключения. Как вам должно быть уже известно, попытка индексировать массив за его границами приводит к ошибке. Когда возникает подобная ошибка, система CLR генерирует исключение `IndexOutOfRangeException`, которое определено как стандартное для среды .NET Framework. В приведенной ниже программе такое исключение генерируется намеренно и затем перехватывается.

```
// Продемонстрировать обработку исключительной ситуации.

using System;

class ExcDemo1 {
  static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine("До генерирования исключения.");

      // Сгенерировать исключение в связи с выходом индекса за границы массива.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
```

```
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

      Console.WriteLine("Не подлежит выводу");
    }

    catch (IndexOutOfRangeException) {
      // Перехватить исключение.
      Console.WriteLine("Индекс вышел за границы массива!");
    }
    Console.WriteLine("После блока перехвата исключения.");
  }
}
```

При выполнении этой программы получается следующий результат.

```
До генерирования исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Индекс вышел за границы массива!
После блока перехвата исключения.
```

В данном примере массив `nums` типа `int` состоит из четырех элементов. Но в цикле `for` предпринимается попытка проиндексировать этот массив от 0 до 9, что и приводит к появлению исключения `IndexOutOfRangeException`, когда происходит обращение к элементу массива по индексу 4.

Несмотря на всю свою краткость, приведенный выше пример наглядно демонстрирует ряд основных моментов процесса обработки исключительных ситуаций. Во-первых, код, который требуется контролировать на наличие ошибок, содержится в блоке `try`. Во-вторых, когда возникает исключительная ситуация (в данном случае — при попытке проиндексировать массив `nums` за его границами в цикле `for`), в блоке `try` генерируется исключение, которое затем перехватывается в блоке `catch`. В этот момент выполнение кода в блоке `try` завершается и управление передается блоку `catch`. Это означает, что оператор `catch` *не* вызывается специально, а выполнение кода переходит к нему автоматически. Следовательно, оператор, содержащий метод `WriteLine()` и следующий непосредственно за циклом `for`, где происходит выход индекса за границы массива, вообще не выполняется. А в задачу обработчика исключений входит исправление ошибки, приведшей к исключительной ситуации, чтобы продолжить выполнение программы в нормальном режиме.

Обратите внимание на то, что в операторе `catch` указан только тип исключения (в данном случае — `IndexOutOfRangeException`), а переменная исключения отсутствует. Как упоминалось ранее, переменную исключения требуется указывать лишь в том случае, если требуется доступ к объекту исключения. В ряде случаев значение объекта исключения может быть использовано обработчиком исключений для получения дополнительной информации о самой ошибке, но зачастую для обработки исключительной ситуации достаточно просто знать, что она произошла. Поэтому переменная исключения нередко отсутствует в обработчиках исключений, как в рассматриваемом здесь примере.

Как пояснялось ранее, если исключение не генерируется в блоке `try`, то блок `catch` не выполняется, а управление программой передается оператору, следующему после

блока catch. Для того чтобы убедиться в этом, замените в предыдущем примере программы строку кода

```
for(int i=0; i < 10; i++) {
```

на строку

```
for(int i=0; i < nums.Length; i++) {
```

Теперь индексирование массива не выходит за его границы в цикле for. Следовательно, никакого исключения не генерируется и блок catch не выполняется.

Второй пример обработки исключительной ситуации

Следует особо подчеркнуть, что весь код, выполняемый в блоке try, контролируется на предмет исключительных ситуаций, в том числе и тех, которые могут возникнуть в результате вызова метода из самого блока try. Исключение, генерируемое методом в блоке try, может быть перехвачено в том же блоке, если, конечно, этого не будет сделано в самом методе.

В качестве еще одного примера рассмотрим следующую программу, где блок try помещается в методе Main(). Из этого блока вызывается метод GenException(), в котором и генерируется исключение IndexOutOfRangeException. Это исключение не перехватывается методом GenException(). Но поскольку метод GenException() вызывается из блока try в методе Main(), то исключение перехватывается в блоке catch, связанном непосредственно с этим блоком try.

```
/* Исключение может быть сгенерировано одним методом
   и перехвачено другим. */

using System;

class ExcTest {
  // Сгенерировать исключение.
  public static void GenException() {
    int[] nums = new int[4];

    Console.WriteLine("До генерирования исключения.");

    // Сгенерировать исключение в связи с выходом индекса за границы
массива.
    for(int i=0; i < 10; i++) {
      nums[i] = i;
      Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
    }

    Console.WriteLine("Не подлежит выводу");
  }
}

class ExcDemo2 {
  static void Main() {

    try {
      ExcTest.GenException();
```

```
    }
    catch (IndexOutOfRangeException) {
      // Перехватить исключение.
      Console.WriteLine("Индекс вышел за границы массива!");
    }
    Console.WriteLine("После блока перехвата исключения.");
  }
}
```

Выполнение этой программы дает такой же результат, как и в предыдущем примере.

```
До генерирования исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Индекс вышел за границы массива!
После блока перехвата исключения.
```

Как пояснялось выше, метод `GenException()` вызывается из блока `try`, и поэтому генерируемое им исключение перехватывается не в нем, а в блоке `catch` внутри метода `Main()`. А если бы исключение перехватывалось в методе `GenException()`, оно не было бы вообще передано обратно методу `Main()`.

Последствия неперехвата исключений

Перехват одного из стандартных исключений, как в приведенных выше примерах, дает еще одно преимущество: он исключает аварийное завершение программы. Как только исключение будет сгенерировано, оно должно быть перехвачено каким-то фрагментом кода в определенном месте программы. Вообще говоря, если исключение не перехватывается в программе, то оно будет перехвачено исполняющей системой. Но дело в том, что исполняющая система выдаст сообщение об ошибке и прервет выполнение программы. Так, в приведенном ниже примере программы исключение в связи с выходом индекса за границы массива не перехватывается.

```
// Предоставить исполняющей системе C# возможность самой обрабатывать ошибки.

using System;

class NotHandled {
  static void Main() {
    int[] nums = new int[4];

    Console.WriteLine("До генерирования исключения.");

    // Сгенерировать исключение в связи с выходом индекса за границы массива.
    for(int i=0; i < 10; i++) {
      nums[i] = i;
      Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
    }
  }
}
```

Когда возникает ошибка индексирования массива, выполнение программы прерывается и выдается следующее сообщение об ошибке.

```
Необработанное исключение: System.IndexOutOfRangeException:
        Индекс находился вне границ массива.
   в NotHandled.Main() в <имя_файла>:строка 16
```

Это сообщение уведомляет об обнаружении в методе `NotHandled.Main()` необработанного исключения типа `System.IndexOutOfRangeException`, которое связано с выходом индекса за границы массива.

Такие сообщения об ошибках полезны для отладки программы, но, по меньше мере, нежелательны при ее использовании на практике! Именно поэтому так важно организовать обработку исключительных ситуаций в самой программе.

Как упоминалось ранее, тип генерируемого исключения должен соответствовать типу, указанному в операторе `catch`. В противном случае исключение не будет перехвачено. Например, в приведенной ниже программе предпринимается попытка перехватить ошибку нарушения границ массива в блоке `catch`, реагирующем на исключение `DivideByZeroException`, связанное с делением на нуль и являющееся еще одним стандартным исключением. Когда индексирование массива выходит за его границы, генерируется исключение `IndexOutOfRangeException`, но оно не будет перехвачено блоком `catch`, что приведет к аварийному завершению программы.

```
// Не сработает!

using System;

class ExcTypeMismatch {
  static void Main() {
    int[] nums = new int[4];

    try {
      Console.WriteLine("До генерирования исключения.");

      // Сгенерировать исключение в связи с выходом индекса за границы массива.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

      Console.WriteLine("Не подлежит выводу");
    }
   /* Если перехват рассчитан на исключение DivideByZeroException,
      то перехватить ошибку нарушения границ массива не удастся. */
    catch (DivideByZeroException) {
      // Перехватить исключение.
      Console.WriteLine("Индекс вышел за границы массива!");
    }
    Console.WriteLine("После блока перехвата исключения.");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
До генерирования исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3

Необработанное исключение: System.IndexOutOfRangeException:
          Индекс находился вне границ массива
    в ExcTypeMismatch.Main() в <имя_файла>:строка 18
```

Как следует из приведенного выше результата, в блоке `catch`, реагирующем на исключение `DivideByZeroException`, не удалось перехватить исключение `IndexOutOfRangeException`.

Обработка исключительных ситуаций — "изящный" способ устранения программных ошибок

Одно из главных преимуществ обработки исключительных ситуаций заключается в том, что она позволяет вовремя отреагировать на ошибку в программе и затем продолжить ее выполнение. В качестве примера рассмотрим еще одну программу, в которой элементы одного массива делятся на элементы другого. Если при этом происходит деление на нуль, то генерируется исключение `DivideByZeroException`. Обработка подобной исключительной ситуации заключается в том, что программа уведомляет об ошибке и затем продолжает свое выполнение. Таким образом, попытка деления на нуль не приведет к аварийному завершению программы из-за ошибки при ее выполнении. Вместо этого ошибка обрабатывается "изящно", не прерывая выполнение программы.

```
// Изящно обработать исключительную ситуацию и продолжить выполнение программы.

using System;

class ExcDemo3 {
  static void Main() {
    int[] numer = { 4, 8, 16, 32, 64, 128 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        // Перехватить исключение.
        Console.WriteLine("Делить на нуль нельзя!");
      }
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
4 / 2 равно 2
Делить на нуль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на нуль нельзя!
128 / 8 равно 16
```

Из данного примера следует еще один важный вывод: как только исключение обработано, оно удаляется из системы. Поэтому в приведенной выше программе проверка ошибок в блоке `try` начинается снова на каждом шаге цикла `for`, при условии, что все предыдущие исключительные ситуации были обработаны. Это позволяет обрабатывать в программе повторяющиеся ошибки.

Применение нескольких операторов `catch`

С одним оператором `try` можно связать несколько операторов `catch`. И на практике это делается довольно часто. Но все операторы `catch` должны перехватывать исключения разного типа. В качестве примера ниже приведена программа, в которой перехватываются ошибки выхода за границы массива и деления на нуль.

```
// Использовать несколько операторов catch.

using System;

class ExcDemo4 {
  static void Main() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        Console.WriteLine("Делить на нуль нельзя!");
      }
      catch (IndexOutOfRangeException) {
        Console.WriteLine("Подходящий элемент не найден.");
      }
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
4 / 2 равно 2
Делить на нуль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на нуль нельзя!
```

```
128 / 8 равно 16
Подходящий элемент не найден.
Подходящий элемент не найден.
```

Как следует из приведенного выше результата, каждый оператор `catch` реагирует только на свой тип исключения.

Вообще говоря, операторы `catch` выполняются по порядку их следования в программе. Но при этом выполняется только один блок `catch`, в котором тип исключения совпадает с типом генерируемого исключения. А все остальные блоки `catch` пропускаются.

Перехват всех исключений

Время от времени возникает потребность в перехвате всех исключений независимо от их типа. Для этой цели служит оператор `catch`, в котором тип и переменная исключения не указываются. Ниже приведена общая форма такого оператора.

```
catch {
  // обработка исключений
}
```

С помощью такой формы создается "универсальный" обработчик всех исключений, перехватываемых в программе.

Ниже приведен пример такого "универсального" обработчика исключений. Обратите внимание на то, что он перехватывает и обрабатывает оба исключения, `IndexOutOfRangeException` и `DivideByZeroException`, генерируемых в программе.

```
// Использовать "универсальный" обработчик исключений.

using System;

class ExcDemo5 {
  static void Main() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i < numer.Length; i++) {
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch { // "Универсальный" перехват.
        Console.WriteLine("Возникла некоторая исключительная ситуация.");
      }
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
4 / 2 равно 2
Возникла некоторая исключительная ситуация.
```

```
16 / 4 равно 4
32 / 4 равно 8
Возникла некоторая исключительная ситуация.
128 / 8 равно 16
Возникла некоторая исключительная ситуация.
Возникла некоторая исключительная ситуация.
```

Применяя "универсальный" перехват, следует иметь в виду, что его блок должен располагаться последним по порядку среди всех блоков `catch`.

ПРИМЕЧАНИЕ

В подавляющем большинстве случаев "универсальный" обработчик исключений не применяется. Как правило, исключения, которые могут быть сгенерированы в коде, обрабатываются по отдельности. Неправильное использование "универсального" обработчика может привести к тому, что ошибки, перехватывавшиеся при тестировании программы, маскируются. Кроме того, организовать надлежащую обработку всех исключительных ситуаций в одном обработчике не так-то просто. Иными словами, "универсальный" обработчик исключений может оказаться пригодным лишь в особых случаях, например в инструментальном средстве анализа кода во время выполнения.

Вложение блоков `try`

Один блок `try` может быть вложен в другой. Исключение, генерируемое во внутреннем блоке `try` и не перехваченное в соответствующем блоке `catch`, передается во внешний блок `try`. В качестве примера ниже приведена программа, в которой исключение `IndexOutOfRangeException` перехватывается не во внутреннем, а во внешнем блоке `try`.

```
// Использовать вложенный блок try.

using System;

class NestTrys {
  static void Main() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    try { // внешний блок try
      for(int i=0; i < numer.Length; i++) {
        try { // вложенный блок try
          Console.WriteLine(numer[i] + " / " +
                            denom[i] + " равно " +
                            numer[i]/denom[i]);
        }
        catch (DivideByZeroException) {
          Console.WriteLine("Делить на нуль нельзя!");
        }
      }
    }
```

```
    catch (IndexOutOfRangeException) {
      Console.WriteLine("Подходящий элемент не найден.");
      Console.WriteLine("Неисправимая ошибка - программа прервана.");
    }
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
4 / 2 равно 2
Делить на нуль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на нуль нельзя!
128 / 8 равно 16
Подходящий элемент не найден.
Неисправимая ошибка - программа прервана.
```

В данном примере исключение, обрабатываемое во внутреннем блоке `try` и связанное с ошибкой из-за деления на нуль, не мешает дальнейшему выполнению программы. Но ошибка нарушения границ массива, обнаруживаемая во внешнем блоке `try`, приводит к прерыванию программы.

Безусловно, приведенный выше пример демонстрирует далеко не единственное основание для применения вложенных блоков `try`, тем не менее из него можно сделать важный общий вывод. Вложенные блоки `try` нередко применяются для обработки различных категорий ошибок разными способами. В частности, одни ошибки считаются неисправимыми и не подлежат исправлению, а другие ошибки незначительны и могут быть обработаны немедленно. Как правило, внешний блок `try` служит для обнаружения и обработки самых серьезных ошибок, а во внутренних блоках `try` обрабатываются менее серьезные ошибки. Кроме того, внешний блок `try` может стать "универсальным" для тех ошибок, которые не подлежат обработке во внутреннем блоке.

Генерирование исключений вручную

В приведенных выше примерах перехватывались исключения, генерировавшиеся исполняющей системой автоматически. Но исключение может быть сгенерировано и вручную с помощью оператора `throw`. Ниже приведена общая форма такого генерирования:

```
throw exceptOb;
```

где в качестве *exceptOb* должен быть обозначен объект класса исключений, производного от класса `Exception`.

Ниже приведен пример программы, в которой демонстрируется применение оператора `throw` для генерирования исключения `DivideByZeroException`.

```
// Сгенерировать исключение вручную.

using System;

class ThrowDemo {
  static void Main() {
    try {
```

```
      Console.WriteLine("До генерирования исключения.");
      throw new DivideByZeroException();
    }
    catch (DivideByZeroException) {
      Console.WriteLine("Исключение перехвачено.");
    }
    Console.WriteLine("После пары операторов try/catch.");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
До генерирования исключения.
Исключение перехвачено.
После пары операторов try/catch.
```

Обратите внимание на то, что исключение `DivideByZeroException` было сгенерировано с использованием ключевого слова `new` в операторе `throw`. Не следует забывать, что в данном случае генерируется конкретный объект, а следовательно, он должен быть создан перед генерированием исключения. Это означает, что сгенерировать исключение только по его типу нельзя. В данном примере для создания объекта `DivideByZeroException` был автоматически вызван конструктор, используемый по умолчанию, хотя для генерирования исключений доступны и другие конструкторы.

Повторное генерирование исключений

Исключение, перехваченное в одном блоке `catch`, может быть повторно сгенерировано в другом блоке, чтобы быть перехваченным во внешнем блоке `catch`. Наиболее вероятной причиной для повторного генерирования исключения служит предоставление доступа к исключению нескольким обработчикам. Допустим, что один обработчик оперирует каким-нибудь одним аспектом исключения, а другой обработчик — другим его аспектом. Для повторного генерирования исключения достаточно указать оператор `throw` без сопутствующего выражения, как в приведенной ниже форме.

```
throw ;
```

Не следует, однако, забывать, что когда исключение генерируется повторно, то оно не перехватывается снова тем же самым блоком `catch`, а передается во внешний блок `catch`.

В приведенном ниже примере программы демонстрируется повторное генерирование исключения. В данном случае генерируется исключение `IndexOutOfRangeException`.

```
// Сгенерировать исключение повторно.

using System;

class Rethrow {
  public static void GenException() {
    // Здесь массив numer длиннее массива denom.
    int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] denom = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.Length; i++) {
```

```
      try {
        Console.WriteLine(numer[i] + " / " +
                          denom[i] + " равно " +
                          numer[i]/denom[i]);
      }
      catch (DivideByZeroException) {
        Console.WriteLine("Делить на нуль нельзя!");
      }
      catch (IndexOutOfRangeException) {
        Console.WriteLine("Подходящий элемент не найден.");
        throw; // сгенерировать исключение повторно
      }
    }
  }
}

class RethrowDemo {
  static void Main() {
    try {
      Rethrow.GenException();
    }
    catch(IndexOutOfRangeException) {
      // перехватить исключение повторно
      Console.WriteLine("Неисправимая ошибка - программа прервана.");
    }
  }
}
```

В этом примере программы ошибки из-за деления на нуль обрабатываются локально в методе `GenException()`, но ошибка выхода за границы массива генерируется повторно. В данном случае исключение `IndexOutOfRangeException` обрабатывается в методе `Main()`.

Использование блока `finally`

Иногда требуется определить кодовый блок, который будет выполняться после выхода из блока `try/catch`. В частности, исключительная ситуация может возникнуть в связи с ошибкой, приводящей к преждевременному возврату из текущего метода. Но в этом методе мог быть открыт файл, который нужно закрыть, или же установлено сетевое соединение, требующее разрывания. Подобные ситуации нередки в программировании, и поэтому для их разрешения в C# предусмотрен удобный способ: воспользоваться блоком `finally`.

Для того чтобы указать кодовый блок, который должен выполняться после блока `try/catch`, достаточно вставить блок `finally` в конце последовательности операторов `try/catch`. Ниже приведена общая форма совместного использования блоков `try/catch` и `finally`.

```
try {
// Блок кода, предназначенный для обработки ошибок.
}
catch (ExcepType1 exOb) {
// Обработчик исключения типа ExcepType1.
```

```
}
catch (ExcepType2 exOb) {
// Обработчик исключения типа ExcepType2.
}
.
.
.
finally {
// Код завершения обработки исключений.
}
```

Блок `finally` будет выполняться всякий раз, когда происходит выход из блока `try/catch`, независимо от причин, которые к этому привели. Это означает, что если блок `try` завершается нормально или по причине исключения, то последним выполняется код, определяемый в блоке `finally`. Блок `finally` выполняется и в том случае, если любой код в блоке `try` или в связанных с ним блоках `catch` приводит к возврату из метода.

Ниже приведен пример применения блока `finally`.

```
// Использовать блок finally.

using System;

class UseFinally {
  public static void GenException(int what) {
    int t;
    int[] nums = new int[2];

    Console.WriteLine("Получить " + what);
    try {
      switch(what) {
        case 0:
          t = 10 / what; // сгенерировать ошибку из-за деления на нуль
          break;
        case 1:
          nums[4] = 4; // сгенерировать ошибку индексирования массива
          break;
        case 2:
          return; // возврат из блока try
      }
    }
    catch (DivideByZeroException) {
      Console.WriteLine("Делить на нуль нельзя!");
      return; // возврат из блока catch
    }
    catch (IndexOutOfRangeException) {
      Console.WriteLine("Совпадающий элемент не найден.");
    }
    finally {
      Console.WriteLine("После выхода из блока try.");
    }
  }
}
```

```
class FinallyDemo {
  static void Main() {

    for(int i=0; i < 3; i++) {
      UseFinally.GenException(i);
      Console.WriteLine();
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Получить 0
Делить на нуль нельзя
После выхода из блока try.

Получить 1
Совпадающий элемент не найден.
После выхода из блока try.

Получить 2
После выхода из блока try.
```

Как следует из приведенного выше результата, блок `finally` выполняется независимо от причины выхода из блока `try`.

И еще одно замечание: с точки зрения синтаксиса блок `finally` следует после блока `try`, и формально блоки `catch` для этого не требуются. Следовательно, блок `finally` можно ввести непосредственно после блока `try`, опустив блоки `catch`. В этом случае блок `finally` начнет выполняться сразу же после выхода из блока `try`, но исключения обрабатываться не будут.

Подробное рассмотрение класса `Exception`

В приведенных выше примерах исключения только перехватывались, но никакой существенной обработке они не подвергались. Как пояснялось выше, в операторе `catch` допускается указывать тип *и* переменную исключения. Переменная получает ссылку на объект исключения. Во всех исключениях поддерживаются члены, определенные в классе `Exception`, поскольку все исключения являются производными от этого класса. В этом разделе будет рассмотрен ряд наиболее полезных членов и конструкторов класса `Exception` и приведены конкретные примеры использования переменной исключения.

В классе `Exception` определяется ряд свойств. К числу самых интересных относятся три свойства: `Message`, `StackTrace` и `TargetSite`. Все эти свойства доступны только для чтения. Свойство `Message` содержит символьную строку, описывающую характер ошибки; свойство `StackTrace` — строку с вызовами стека, приведшими к исключительной ситуации, а свойство `TargetSite` получает объект, обозначающий метод, сгенерировавший исключение.

Кроме того, в классе `Exception` определяется ряд методов. Чаще всего приходится пользоваться методом `ToString()`, возвращающим символьную строку с описанием исключения. Этот метод автоматически вызывается, например, при отображении исключения с помощью метода `WriteLine()`.

Применение всех трех упомянутых выше свойств и метода из класса `Exception` демонстрируется в приведенном ниже примере программы.

```
// Использовать члены класса Exception.

using System;

class ExcTest {
  public static void GenException() {
    int[] nums = new int[4];

    Console.WriteLine("До генерирования исключения.");
    // Сгенерировать исключение в связи с выходом за границы массива.
      for(int i=0; i < 10; i++) {
        nums[i] = i;
        Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
      }

    Console.WriteLine("Не подлежит выводу");
  }
}

class UseExcept {
  static void Main() {
    try {
      ExcTest.GenException();
    }
    catch (IndexOutOfRangeException exc) {
      Console.WriteLine("Стандартное сообщение таково: ");
      Console.WriteLine(exc); // вызвать метод ToString()
      Console.WriteLine("Свойство StackTrace: " + exc.StackTrace);
      Console.WriteLine("Свойство Message: " + exc.Message);
      Console.WriteLine("Свойство TargetSite: " + exc.TargetSite);
    }
    Console.WriteLine("После блока перехвата исключения.");
  }
}
```

При выполнении этой программы получается следующий результат.

```
До генерирования исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Стандартное сообщение таково: System.IndexOutOfRangeException:  Индекс находился
вне границ массива.
    в ExcTest.genException() в <имя_файла>:строка 15
    в UseExcept.Main()в <имя_файла>:строка 29
Свойство StackTrace:  в ExcTest.genException()в <имя_файла>:строка 15
   в UseExcept.Main()в <имя_файла>:строка 29
Свойство Message:  Индекс находился вне границ массива.
Свойство TargetSite:  Void genException()
После блока перехвата исключения.
```

В классе Exception определяются четыре следующих конструктора.

```
public Exception()
public Exception(string сообщение)
public Exception(string сообщение, Exception внутреннее_исключение)
protected Exception(System.Runtime.Serialization.SerializationInfo информация,
                    System.Runtime.Serialization.StreamingContext контекст)
```

Первый конструктор используется по умолчанию. Во втором конструкторе указывается строка *сообщение*, связанная со свойством Message, которое имеет отношение к генерируемому исключению. В третьем конструкторе указывается так называемое *внутреннее исключение*. Этот конструктор используется в том случае, когда одно исключение порождает другое, причем *внутреннее_исключение* обозначает первое исключение, которое будет пустым, если внутреннее исключение отсутствует. (Если внутреннее исключение присутствует, то оно может быть получено из свойства InnerException, определяемого в классе Exception.) И последний конструктор обрабатывает исключения, происходящие дистанционно, и поэтому требует десериализации.

Следует также заметить, что в четвертом конструкторе класса Exception типы SerializationInfo и StreamingContext относятся к пространству имен System.Runtime.Serialization.

Наиболее часто используемые исключения

В пространстве имен System определено несколько стандартных, встроенных исключений. Все эти исключения являются производными от класса SystemException, поскольку они генерируются системой CLR при появлении ошибки во время выполнения. В табл. 13.1 перечислены некоторые наиболее часто используемые стандартные исключения.

Таблица 13.1. Наиболее часто используемые исключения, определенные в пространстве имен System

Исключение	Значение
ArrayTypeMismatchException	Тип сохраняемого значения несовместим с типом массива
DivideByZeroException	Попытка деления на нуль
IndexOutOfRangeException	Индекс оказался за границами массива
InvalidCastException	Неверно выполнено динамическое приведение типов
OutOfMemoryException	Недостаточно свободной памяти для дальнейшего выполнения программы. Это исключение может быть, например, сгенерировано, если для создания объекта с помощью оператора new не хватает памяти
OverflowException	Произошло арифметическое переполнение
NullReferenceException	Попытка использовать пустую ссылку, т.е. ссылку, которая не указывает ни на один из объектов

Большинство исключений, приведенных в табл. 13.1, не требует особых пояснений, кроме исключения NullReferenceException. Это исключение генерируется при по-

пытке использовать пустую ссылку на несуществующий объект, например, при вызове метода по пустой ссылке. *Пустой* называется такая ссылка, которая не указывает ни на один из объектов. Для того чтобы создать такую ссылку, достаточно, например, присвоить явным образом пустое значение переменной ссылочного типа, используя ключевое слово `null`. Пустые ссылки могут также появляться и другими, менее очевидными путями. Ниже приведен пример программы, демонстрирующий обработку исключения `NullReferenceException`.

```
// Продемонстрировать обработку исключения NullReferenceException.

using System;

class X {
  int x;
  public X(int a) {
  x = a;
  }

  public int Add(X o) {
    return x + o.x;
  }
}

// Продемонстрировать генерирование и обработку
// исключения NullReferenceException.
class NREDemo {
  static void Main() {
    X p = new X(10);
    X q = null; // присвоить явным образом пустое значение переменной q
    int val;

    try {
      val = p.Add(q); // эта операция приведет к исключительной ситуации
    } catch (NullReferenceException) {
      Console.WriteLine("Исключение NullReferenceException!");
      Console.WriteLine("Исправление ошибки...\n");

      // А теперь исправить ошибку.
      q = new X(9);
      val = p.Add(q);
    }
    Console.WriteLine("Значение val равно {0}", val);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Исключение NullReferenceException!
Исправление ошибки...

Значение val равно 19
```

В приведенном выше примере программы создается класс `X`, в котором определяются член `x` и метод `Add()`, складывающий значение члена `x` в вызывающем объекте

со значением члена x в объекте, передаваемом этому методу в качестве параметра. Оба объекта класса X создаются в методе Main(). Первый из них (переменная p) инициализируется, а второй (переменная q) — нет. Вместо этого переменной q присваивается пустое значение. Затем вызывается метод p.Add() с переменной q в качестве аргумента. Но поскольку переменная q не ссылается ни на один из объектов, то при попытке получить значение члена q.x генерируется исключение NullReferenceException.

Получение производных классов исключений

Несмотря на то что встроенные исключения охватывают наиболее распространенные программные ошибки, обработка исключительных ситуаций в C# не ограничивается только этими ошибками. В действительности одна из сильных сторон принятого в C# подхода к обработке исключительных ситуаций состоит в том, что в этом языке допускается использовать исключения, определяемые пользователем, т.е. тем, кто программирует на C#. В частности, такие специальные исключения можно использовать для обработки ошибок в собственном коде, а создаются они очень просто. Для этого достаточно определить класс, производный от класса Exception. В таких классах совсем не обязательно что-то реализовывать — одного только их существования в системе типов уже достаточно, чтобы использовать их в качестве исключений.

ПРИМЕЧАНИЕ

В прошлом специальные исключения создавались как производные от класса Application.Exception, поскольку эта иерархия классов была первоначально зарезервирована для исключений прикладного характера. Но теперь корпорация Microsoft не рекомендует этого делать, а вместо этого получать исключения, производные от класса Exception. Именно по этой причине данный подход и рассматривается в настоящей книге.

Создаваемые пользователем классы будут автоматически получать свойства и методы, определенные в классе Exception и доступные для них. Разумеется, любой из этих членов класса Exception можно переопределить в создаваемых классах исключений.

Когда создается собственный класс исключений, то, как правило, желательно, чтобы в нем поддерживались все конструкторы, определенные в классе Exception. В простых специальных классах исключений этого нетрудно добиться, поскольку для этого достаточно передать подходящие аргументы соответствующему конструктору класса Exception, используя ключевое слово base. Но формально нужно предоставить только те конструкторы, которые фактически используются в программе.

Рассмотрим пример программы, в которой используется исключение специального типа. Напомним, что в конце главы 10 был разработан класс RangeArray, поддерживающий одномерные массивы, в которых начальный и конечный индексы определяются пользователем. Так, например, вполне допустимым считается массив, индексируемый в пределах от –5 до 27. Если же индекс выходил за границы массива, то для обработки этой ошибки в классе RangeArray была определена специальная переменная. Такая переменная устанавливалась и проверялась после каждой операции обращения к массиву в коде, использовавшем класс RangeArray. Безусловно, такой подход к обработке ошибок "неуклюж" и чреват дополнительными ошибками. В приведенном ниже улучшенном варианте класса RangeArray обработка ошибок нарушения границ

массива выполняется более изящным и надежным способом с помощью специально генерируемого исключения.

```
// Использовать специальное исключение для обработки
// ошибок при обращении к массиву класса RangeArray.

using System;

// Создать исключение для класса RangeArray.
class RangeArrayException : Exception {
/* Реализовать все конструкторы класса Exception. Такие конструкторы просто
   реализуют конструктор базового класса. А поскольку класс исключения
   RangeArrayException ничего не добавляет к классу Exception, то никаких
   дополнительных действий не требуется. */
  public RangeArrayException() : base() { }
  public RangeArrayException(string str) : base(str) { }
  public RangeArrayException(
    string str, Exception inner) : base(str, inner) { }
  protected RangeArrayException(
     System.Runtime.Serialization.SerializationInfo si,
     System.Runtime.Serialization.StreamingContext sc) :
       base(si, sc) { }

  // Переопределить метод ToString() для класса исключения RangeArrayException.
  public override string ToString() {
    return Message;
  }
}

// Улучшенный вариант класса RangeArray.
class RangeArray {
  // Закрытые данные.
  int[] a; // ссылка на базовый массив
  int lowerBound; // наименьший индекс
  int upperBound; // наибольший индекс

  // Автоматически реализуемое и доступное только для чтения свойство Length.
  public int Length { get; private set; }

  // Построить массив по заданному размеру
  public RangeArray(int low, int high) {
    high++;
    if(high <= low) {
      throw new RangeArrayException("Нижний индекс не меньше верхнего.");
    }
    a = new int[high - low];
    Length = high - low;

    lowerBound = low;
    upperBound = --high;
  }

  // Это индексатор для класса RangeArray.
  public int this[int index] {
```

```
    // Это аксессор get.
    get {
      if(ok(index)) {
        return a[index - lowerBound];
      } else {
        throw new RangeArrayException("Ошибка нарушения границ.");
      }
    }

    // Это аксессор set.
    set {
      if(ok(index)) {
        a[index - lowerBound] = value;
      }
      else throw new RangeArrayException("Ошибка нарушения границ.");
      }
    }

  // Возвратить логическое значение true, если
  // индекс находится в установленных границах.
  private bool ok(int index) {
    if(index >= lowerBound & index <= upperBound) return true;
   return false;
  }
}

// Продемонстрировать применение массива с произвольно
// задаваемыми пределами индексирования.
class RangeArrayDemo {
  static void Main() {
    try {
      RangeArray ra = new RangeArray(-5, 5);
      RangeArray ra2 = new RangeArray(1, 10);

      // Использовать объект ra в качестве массива.
      Console.WriteLine("Длина массива ra: " + ra.Length);
      for(int i = -5; i <= 5; i++)
        ra[i] = i;

      Console.Write("Содержимое массива ra: ");
      for(int i = -5; i <= 5; i++)
        Console.Write(ra[i] + " ");

      Console.WriteLine("\n");

      // Использовать объект ra2 в качестве массива.
      Console.WriteLine("Длина массива ra2: " + ra2.Length);
      for(int i = 1; i <= 10; i++)
        ra2[i] = i;

      Console.Write("Длина массива ra2: ");
      for(int i = 1; i <= 10; i++)
        Console.Write(ra2[i] + " ");
```

```
      Console.WriteLine("\n");

    } catch (RangeArrayException exc) {
        Console.WriteLine(exc);
    }

    // А теперь продемонстрировать обработку некоторых ошибок.
    Console.WriteLine("Сгенерировать ошибки нарушения границ.");

    // Использовать неверно заданный конструктор.
    try {
      RangeArray ra3 = new RangeArray(100, -10); // Ошибка!
    } catch (RangeArrayException exc) {
        Console.WriteLine(exc);
    }

    // Использовать неверно заданный индекс.
    try {
      RangeArray ra3 = new RangeArray(-2, 2);

      for(int i = -2; i <= 2; i++)
        ra3[i] = i;

      Console.Write("Содержимое массива ra3: ");
      for(int i = -2; i <= 10; i++) // сгенерировать ошибку нарушения границ
        Console.Write(ra3[i] + " ");

    } catch (RangeArrayException exc) {
        Console.WriteLine(exc);
    }
  }
}
```

После выполнения этой программы получается следующий результат.

```
Длина массива ra: 11
Содержимое массива ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Длина массива ra2: 10
Содержимое массива ra2: 1 2 3 4 5 6 7 8 9 10

Сгенерировать ошибки нарушения границ.
Нижний индекс не меньше верхнего.
Содержимое массива ra3: -2 -1 0 1 2 Ошибка нарушения границ.
```

Когда возникает ошибка нарушения границ массива класса `RangeArray`, генерируется объект типа `RangeArrayException`. В классе `RangeArray` это может произойти в трех следующих местах: в аксессоре `get` индексатора, в аксессоре `set` индексатора и в конструкторе класса `RangeArray`. Для перехвата этих исключений подразумевается, что объекты типа `RangeArray` должны быть сконструированы и доступны из блока `try`, что и продемонстрировано в приведенной выше программе. Используя специальное исключение для сообщения об ошибках, класс `RangeArray` теперь действует как один из встроенных в C# типов данных, и поэтому он может быть полностью интегрирован в механизм обработки ошибок, обнаруживаемых в программе.

Обратите внимание на то, что в теле конструкторов класса исключения RangeArrayException отсутствуют какие-либо операторы, но вместо этого они просто передают свои аргументы классу Exception, используя ключевое слово base. Как пояснялось ранее, в тех случаях, когда производный класс исключений не дополняет функции базового класса, весь процесс создания исключений можно поручить конструкторам класса Exception. Ведь производный класс исключений совсем не обязательно должен чем-то дополнять функции, наследуемые от класса Exception.

Прежде чем переходить к дальнейшему чтению, попробуйте немного поэкспериментировать с приведенной выше программой. В частности, попробуйте закомментировать переопределение метода ToString() и понаблюдайте за результатами. Кроме того, попытайтесь создать исключение, используя конструктор, вызываемый по умолчанию, и посмотрите, какое сообщение при этом сформируется стандартными средствами C#.

Перехват исключений производных классов

При попытке перехватить типы исключений, относящихся как к базовым, так и к производным классам, следует особенно внимательно соблюдать порядок следования операторов catch, поскольку перехват исключения базового класса будет совпадать с перехватом исключений любых его производных классов. Например, класс Exception является базовым для всех исключений, и поэтому вместе с исключением типа Exception могут быть перехвачены и все остальные исключения производных от него классов. Конечно, для более четкого перехвата всех исключений можно воспользоваться упоминавшейся ранее формой оператора catch без указания конкретного типа исключения. Но вопрос перехвата исключений производных классов становится весьма актуальным и в других ситуациях, особенно при создании собственных исключений.

Если требуется перехватывать исключения базового и производного классов, то первым по порядку должен следовать оператор catch, перехватывающий исключение производного класса. Это правило необходимо соблюдать потому, что при перехвате исключения базового класса будут также перехвачены исключения всех производных от него классов. Правда, это правило соблюдается автоматически: если первым расположить в коде оператор catch, перехватывающий исключение базового класса, то во время компиляции этого кода будет выдано сообщение об ошибке.

В приведенном ниже примере программы создаются два класса исключений: ExceptA и ExceptB. Класс ExceptA является производным от класса Exception, а класс ExceptB — производным от класса ExceptA. Затем в программе генерируются исключения каждого типа. Ради краткости в классах специальных исключений предоставляется только один конструктор, принимающий символьную строку, описывающую исключение. Но при разработке программ коммерческого назначения в классах специальных исключений обычно требуется предоставлять все четыре конструктора, определяемых в классе Exception.

```
// Исключения производных классов должны появляться до
// исключений базового класса.

using System;

// Создать класс исключения.
class ExceptA : Exception {
  public ExceptA(string str) : base(str) { }
```

```
  public override string ToString() {
    return Message;
  }
}

// Создать класс исключения, производный от класса ExceptA.
class ExceptB : ExceptA {
  public ExceptB(string str) : base(str) { }
  public override string ToString() {
    return Message;
  }
}

class OrderMatters {
  static void Main() {
    for(int x = 0; x < 3; x++) {
      try {
        if(x==0) throw new ExceptA("Перехват исключения типа ExceptA");
        else if(x==1) throw new ExceptB("Перехват исключения типа
ExceptB");
        else throw new Exception();
      }
      catch (ExceptB exc) {
        Console.WriteLine(exc);
      }
      catch (ExceptA exc) {
        Console.WriteLine(exc);
      }
      catch (Exception exc) {
        Console.WriteLine(exc);
      }
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Перехват исключения типа ExceptA.
Перехват исключения типа ExceptB.
System.Exception:  Выдано исключение типа "System.Exception".
   в OrderMatters.Main() в <имя_файла>:строка 36
```

Обратите внимание на порядок следования операторов `catch`. Именно в таком порядке они и должны выполняться. Класс `ExceptB` является производным от класса `ExceptA`, поэтому исключение типа `ExceptB` должно перехватываться до исключения типа `ExceptA`. Аналогично, исключение типа `Exception` (т.е. базового класса для всех исключений) должно перехватываться последним. Для того чтобы убедиться в этом, измените порядок следования операторов `catch`. В итоге это приведет к ошибке во время компиляции.

Полезным примером использования оператора `catch`, перехватывающего исключения базового класса, служит перехват всей категории исключений. Допустим, что создается ряд исключений для управления некоторым устройством. Если сделать их классы производными от общего базового класса, то в тех приложениях, где необязательно выяснять конкретную причину возникшей ошибки, достаточно перехватывать исключение базового класса и тем самым исключить ненужное дублирование кода.

Применение ключевых слов checked и unchecked

В C# имеется специальное средство, связанное с генерированием исключений, возникающих при переполнении в арифметических вычислениях. Как вам должно быть уже известно, результаты некоторых видов арифметических вычислений могут превышать диапазон представления чисел для типа данных, используемого в вычислении. В этом случае происходит так называемое *переполнение*. Рассмотрим в качестве примера следующий фрагмент кода.

```
byte a, b, result;
a = 127;
b = 127;

result = (byte)(a * b);
```

В этом коде произведение значений переменных a и b превышает диапазон представления чисел для типа byte. Следовательно, результат вычисления данного выражения приводит к переполнению для типа данных, сохраняемого в переменной result.

В C# допускается указывать, будет ли в коде сгенерировано исключение при переполнении, с помощью ключевых слов checked и unchecked. Так, если требуется указать, что выражение будет проверяться на переполнение, следует использовать ключевое слово checked, а если требуется проигнорировать переполнение — ключевое слово unchecked. В последнем случае результат усекается, чтобы не выйти за пределы диапазона представления чисел для целевого типа выражения.

У ключевого слова checked имеются две общие формы. В одной форме проверяется конкретное выражение, и поэтому она называется *операторной*. А в другой форме проверяется блок операторов, и поэтому она называется *блочной*. Ниже приведены обе формы:

```
checked (выражение)

checked {
  // проверяемые операторы
}
```

где *выражение* обозначает проверяемое выражение. Если вычисление проверяемого выражения приводит к переполнению, то генерируется исключение OverflowException.

У ключевого слова unchecked также имеются две общие формы. В первой, операторной форме переполнение игнорируется при вычислении конкретного выражения. А во второй, блочной форме оно игнорируется при выполнении блока операторов:

```
unchecked (выражение)

unchecked {
  // операторы, для которых переполнение игнорируется
}
```

где *выражение* обозначает конкретное выражение, при вычислении которого переполнение игнорируется. Если же в непроверяемом выражении происходит переполнение, то результат его вычисления усекается.

Ниже приведен пример программы, в котором демонстрируется применение ключевых слов checked и unchecked.

```
// Продемонстрировать применение ключевых слов checked и unchecked.

using System;

class CheckedDemo {
  static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;

    try {
      result = unchecked((byte)(a * b));
      Console.WriteLine("Непроверенный на переполнение результат: " +
                        result);
      result = checked((byte)(a * b)); // эта операция приводит к
                                       // исключительной ситуации
      Console.WriteLine("Проверенный на переполнение результат: " +
                        result); // не подлежит выполнению
    }
    catch (OverflowException exc) {
      Console.WriteLine(exc);
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
Непроверенный на переполнение результат: 1
System.OverflowException: Переполнение в результате
выполнения арифметической операции.
    в CheckedDemo.Main() в <имя_файла>:строка 20
```

Как видите, результат вычисления непроверяемого выражения был усечен. А вычисление проверяемого выражения привело к исключительной ситуации.

В представленном выше примере программы было продемонстрировано применение ключевых слов checked и unchecked в одном выражении. А в следующем примере программы показывается, каким образом проверяется и не проверяется на переполнение целый блок операторов.

```
// Продемонстрировать применение ключевых слов checked
// и unchecked в блоке операторов.

using System;

class CheckedBlocks {
  static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;
```

```
    try {
      unchecked {
        a = 127;
        b = 127;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Непроверенный на переполнение результат: " +
                          result);

        a = 125;
        b = 5;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Непроверенный на переполнение результат: " +
                          result);
      }
      checked {
        a = 2;
        b = 7;
        result = checked((byte)(a * b)); // верно
        Console.WriteLine("Проверенный на переполнение результат: " +
                          result);

        a = 127;
        b = 127;
        result = checked((byte)(a * b)); // эта операция приводит к
                                         // исключительной ситуации
        Console.WriteLine("Проверенный на переполнение результат: " +
                          result); // не подлежит выполнению
      }
    }
    catch (OverflowException exc) {
      Console.WriteLine(exc);
    }
  }
}
```

Результат выполнения этой программы приведен ниже.

```
Непроверенный на переполнение результат: 1
Непроверенный на переполнение результат: 113
Проверенный на переполнение результат: 14
System.OverflowException: Переполнение в результате
выполнения арифметической операции.
    в CheckedDemo.Main() в <имя_файла>:строка 41
```

Как видите, результаты выполнения непроверяемого на переполнение блока операторов были усечены. Когда же в проверяемом блоке операторов произошло переполнение, то возникла исключительная ситуация.

Потребность в применении ключевого слова `checked` или `unchecked` может возникнуть, в частности, потому, что по умолчанию проверяемое или непроверяемое состояние переполнения определяется путем установки соответствующего параметра компилятора и настройки самой среды выполнения. Поэтому в некоторых программах состояние переполнения лучше проверять явным образом.

14 ГЛАВА

Применение средств ввода-вывода

В примерах программ, приводившихся в предыдущих главах, уже применялись отдельные части системы ввода-вывода в C#, например метод `Console.WriteLine()`, но делалось это без каких-либо формальных пояснений. Система ввода-вывода основана в C# на иерархии классов, поэтому ее функции и особенности нельзя было представлять до тех пор, пока не были рассмотрены классы, наследование и исключения. А теперь настал черед и для ввода-вывода. В C# применяется система ввода-вывода и классы, определенные в среде .NET Framework, и поэтому рассмотрение ввода-вывода в этом языке относится ко всей системе ввода-вывода среды .NET в целом.

В этой главе речь пойдет о средствах консольного и файлового ввода-вывода. Следует, однако, сразу же предупредить, что система ввода-вывода в C# довольно обширна. Поэтому в этой главе рассматриваются лишь самые важные и наиболее часто используемые ее средства.

Организация системы ввода-вывода в C# на потоках

Ввод-вывод в программах на C# осуществляется посредством потоков. *Поток* — это некая абстракция производства или потребления информации. С физическим устройством поток связывает система ввода-вывода. Все потоки действуют одинаково — даже если они связаны с разными физическими устройствами. Поэтому классы и методы ввода-вывода могут применяться к самым разным типам устройств. Например, методами вывода на консоль можно пользоваться и для вывода в файл на диске.

Байтовые и символьные потоки

На самом низком уровне ввод-вывод в C# осуществляется байтами. И делается это потому, что многие устройства ориентированы на операции ввода-вывода отдельными байтами. Но человеку больше свойственно общаться символами. Напомним, что в C# тип `char` является 16-разрядным, а тип `byte` — 8-разрядным. Так, если в целях ввода-вывода используется набор символов в коде ASCII, то для преобразования типа `char` в тип `byte` достаточно отбросить старший байт значения типа `char`. Но это не годится для набора символов в уникоде (Unicode), где символы требуется представлять двумя, а то и больше байтами. Следовательно, байтовые потоки не совсем подходят для организации ввода-вывода отдельными символами. С целью разрешить это затруднение в среде .NET Framework определено несколько классов, выполняющих превращение байтового потока в символьный с автоматическим преобразованием типа `byte` в тип `char` и обратно.

Встроенные потоки

Для всех программ, в которых используется пространство имен `System`, доступны встроенные потоки, открывающиеся с помощью свойств `Console.In`, `Console.Out` и `Console.Error`. В частности, свойство `Console.Out` связано со стандартным потоком вывода. По умолчанию это поток вывода на консоль. Так, если вызывается метод `Console.WriteLine()`, информация автоматически передается свойству `Console.Out`. Свойство `Console.In` связано со стандартным потоком ввода, который по умолчанию осуществляется с клавиатуры. А свойство `Console.Error` связано со стандартным потоком сообщений об ошибках, которые по умолчанию также выводятся на консоль. Но эти потоки могут быть переадресованы на любое другое совместимое устройство ввода-вывода. Стандартные потоки являются символьными. Поэтому в эти потоки выводятся и вводятся из них символы.

Классы потоков

В среде .NET Framework определены классы как для байтовых, так и для символьных потоков. Но на самом деле классы символьных потоков служат лишь оболочками для превращения заключенного в них байтового потока в символьный, автоматически выполняя любые требующиеся преобразования типов данных. Следовательно, символьные потоки основываются на байтовых, хотя они и разделены логически.

Основные классы потоков определены в пространстве имен `System.IO`. Для того чтобы воспользоваться этими классами, как правило, достаточно ввести приведенный ниже оператор в самом начале программы.

```
using System.IO;
```

Пространство имен `System.IO` не указывается для консольного ввода-вывода потому, что для него определен класс `Console` в пространстве имен `System`.

Класс `Stream`

Основным для потоков является класс `System.IO.Stream`. Он представляет байтовый поток и является базовым для всех остальных классов потоков. Кроме того, он

является абстрактным классом, а это означает, что получить экземпляр объекта класса `Stream` нельзя. В классе `Stream` определяется ряд операций со стандартными потоками, представленных соответствующими методами. В табл. 14.1 перечислен ряд наиболее часто используемых методов, определенных в классе `Stream`.

Таблица 14.1. Некоторые методы, определенные в классе `Stream`

Метод	Описание
`void Close()`	Закрывает поток
`void Flush()`	Выводит содержимое потока на физическое устройство
`int ReadByte()`	Возвращает целочисленное представление следующего байта, доступного для ввода из потока. При обнаружении конца файла возвращает значение -1
`int Read(byte[] buffer, int offset, int count)`	Делает попытку ввести `count` байтов в массив `buffer`, начиная с элемента `buffer[offset]`. Возвращает количество успешно введенных байтов
`long Seek(long offset, SeekOrigin origin)`	Устанавливает текущее положение в потоке по указанному смещению `offset` относительно заданного начала отсчета `origin`. Возвращает новое положение в потоке
`void WriteByte(byte value)`	Выводит один байт в поток вывода
`void Write(byte[]buffer, int offset, int count)`	Выводит подмножество `count` байтов из массива `buffer`, начиная с элемента `buffer[offset]`. Возвращает количество выведенных байтов

Некоторые из методов, перечисленных в табл. 14.1, генерируют исключение `IOException` при появлении ошибки ввода-вывода. Если же предпринимается попытка выполнить неверную операцию, например вывести данные в поток, предназначенный только для чтения, то генерируется исключение `NotSupportedException`. Кроме того, могут быть сгенерированы и другие исключения — все зависит от конкретного метода.

Следует заметить, что в классе `Stream` определены методы для ввода (или чтения) и вывода (или записи) данных. Но не все потоки поддерживают обе эти операции, поскольку поток можно открывать только для чтения или только для записи. Кроме того, не все потоки поддерживают запрос текущего положения в потоке с помощью метода `Seek()`. Для того чтобы определить возможности потока, придется воспользоваться одним, а то и несколькими свойствами класса `Stream`. Эти свойства перечислены в табл. 14.2 наряду со свойствами `Length` и `Position`, содержащими длину потока и текущее положение в нем.

Таблица 14.2. Свойства, определенные в классе `Stream`

Свойство	Описание
`bool CanRead`	Принимает значение `true`, если из потока можно ввести данные. Доступно только для чтения
`bool CanSeek`	Принимает значение `true`, если поток поддерживает запрос текущего положения в потоке. Доступно только для чтения

Окончание табл. 14.2

Свойство	Описание
`bool CanWrite`	Принимает значение `true`, если в поток можно вывести данные. Доступно только для чтения
`long Length`	Содержит длину потока. Доступно только для чтения
`long Position`	Представляет текущее положение в потоке. Доступно как для чтения, так и для записи
`int ReadTimeout`	Представляет продолжительность времени ожидания в операциях ввода. Доступно как для чтения, так и для записи
`int WriteTimeout`	Представляет продолжительность времени ожидания в операциях вывода. Доступно как для чтения, так и для записи

Классы байтовых потоков

Производными от класса `Stream` являются несколько конкретных классов байтовых потоков. Эти классы определены в пространстве имен `System.IO` и перечислены ниже.

Класс потока	Описание
`BufferedStream`	Заключает в оболочку байтовый поток и добавляет буферизацию. Буферизация, как правило, повышает производительность
`FileStream`	Байтовый поток, предназначенный для файлового ввода-вывода
`MemoryStream`	Байтовый поток, использующий память для хранения данных
`UnmanagedMemoryStream`	Байтовый поток, использующий неуправляемую память для хранения данных

В среде NET Framework поддерживается также ряд других конкретных классов потоков, в том числе для ввода-вывода в сжатые файлы, сокеты и каналы. Кроме того, можно создать свои собственные производные классы потоков, хотя для подавляющего числа приложений достаточно и встроенных потоков.

Классы-оболочки символьных потоков

Для создания символьного потока достаточно заключить байтовый поток в один из классов-оболочек символьных потоков. На вершине иерархии классов символьных потоков находятся абстрактные классы `TextReader` и `TextWriter`. Так, класс `TextReader` организует ввод, а класс `TextWriter` — вывод. Методы, определенные в обоих этих классах, доступны для всех их подклассов. Они образуют минимальный набор функций ввода-вывода, которыми должны обладать все символьные потоки.

В табл. 14.3 перечислены методы ввода, определенные в классе `TextReader`. В целом, эти методы способны генерировать исключение `IOException` при появлении ошибки ввода, а некоторые из них — исключения других типов. Особый интерес вызывает метод `ReadLine()`, предназначенный для ввода целой текстовой строки, возвращая ее в виде объекта типа `string`. Этот метод удобен для чтения входных данных, содержащих пробелы. В классе `TextReader` имеется также метод `Close()`, определяемый следующим образом.

```
void Close()
```

Этот метод закрывает считывающий поток и освобождает его ресурсы.

Таблица 14.3. Методы ввода, определенные в классе `TextReader`

Метод	Описание
`int Peek()`	Получает следующий символ из потока ввода, но не удаляет его. Возвращает значение -1, если ни один из символов не доступен
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца потока возвращает значение -1
`int Read(char[]`*`buffer`*`, int` *`index`*`, int` *`count`*`)`	Делает попытку ввести количество *`count`* символов в массив *`buffer`*, начиная с элемента *`buffer[index]`*, и возвращает количество успешно введенных символов

`int ReadBlock(char[]`*`buffer`*`,`
`int`
`index``,`
`int`
`count``)`
	Делает попытку ввести количество *`count`* символов в массив *`buffer`*, начиная с элемента *`buffer[index]`*, и возвращает количество успешно введенных символов

`string ReadLine()`	Вводит следующую текстовую строку и возвращает ее в виде объекта типа `string`. При попытке прочитать признак конца файла возвращает пустое значение
`string ReadToEnd()`	Вводит все символы, оставшиеся в потоке, и возвращает их в виде объекта типа `string`

В классе `TextWriter` определены также варианты методов `Write()` и `WriteLine()`, предназначенные для вывода данных всех встроенных типов. Ниже в качестве примера перечислены лишь некоторые из перегружаемых вариантов этих методов.

Метод	Описание
`void Write(int`*`value`*`)`	Выводит значение типа `int`
`void Write(double`*`value`*`)`	Выводит значение типа `double`
`void Write(bool`*`value`*`)`	Выводит значение типа `bool`
`void WriteLine(string`*`value`*`)`	Выводит значение типа `string` с последующим символом новой строки
`void WriteLine(uint`*`value`*`)`	Выводит значение типа `uint` с последующим символом новой строки
`void WriteLine(char`*`value`*`)`	Выводит символ с последующим символом новой строки

Все эти методы генерируют исключение `IOException` при появлении ошибки вывода.

Кроме того, в классе `TextWriter` определены методы `Close()` и `Flush()`, приведенные ниже.

```
virtual void Close()
virtual void Flush()
```

Метод `Flush()` организует вывод в физическую среду всех данных, оставшихся в выходном буфере. А метод `Close()` закрывает записывающий поток и освобождает его ресурсы.

Классы `TextReader` и `TextWriter` реализуются несколькими классами символьных потоков, включая и те, что перечислены ниже. Следовательно, в этих классах потоков предоставляются методы и свойства, определенные в классах `TextReader` и `TextWriter`.

Класс потока	Описание
`StreamReader`	Предназначен для ввода символов из байтового потока. Этот класс является оболочкой для байтового потока ввода
`StreamWriter`	Предназначен для вывода символов в байтовый поток. Этот класс является оболочкой для байтового потока вывода
`StringReader`	Предназначен для ввода символов из символьной строки
`StringWriter`	Предназначен для вывода символов в символьную строку

Двоичные потоки

Помимо классов байтовых и символьных потоков, имеются еще два класса двоичных потоков, которые могут служить для непосредственного ввода и вывода двоичных данных — `BinaryReader` и `BinaryWriter`. Подробнее о них речь пойдет далее в этой главе, когда дойдет черед до файлового ввода-вывода.

А теперь, когда представлена общая структура системы ввода-вывода в C#, отведем оставшуюся часть этой главы более подробному рассмотрению различных частей данной системы, начиная с консольного ввода-вывода.

Консольный ввод-вывод

Консольный ввод-вывод осуществляется с помощью стандартных потоков, представленных свойствами `Console.In`, `Console.Out` и `Console.Error`. Примеры консольного ввода-вывода были представлены еще в главе 2, поэтому он должен быть вам уже знаком. Как будет показано ниже, он обладает и рядом других дополнительных возможностей.

Но прежде следует еще раз подчеркнуть, что большинство реальных приложений C# ориентированы не на консольный ввод-вывод в текстовом виде, а на графический оконный интерфейс для взаимодействия с пользователем, или же они представляют собой программный код, используемый на стороне сервера. Поэтому часть системы ввода-вывода, связанная с консолью, не находит широкого практического применения. И хотя программы, ориентированные на текстовый ввод-вывод, отлично подходят в качестве учебных примеров, коротких сервисных программ или определенного рода программных компонентов, для большинства реальных приложений они не годятся.

Чтение данных из потока ввода с консоли

Поток `Console.In` является экземпляром объекта класса `TextReader`, и поэтому для доступа к нему могут быть использованы методы и свойства, определенные в

классе TextReader. Но для этой цели чаще все же используются методы, предоставляемые классом Console, в котором автоматически организуется чтение данных из потока Console.In. В классе Console определены три метода ввода. Два первых метода, Read() и ReadLine(), были доступны еще в версии .NET Framework 1.0. А третий метод, ReadKey(), был добавлен в версию 2.0 этой среды.

Для чтения одного символа служит приведенный ниже метод Read().

```
static int Read()
```

Метод Read() возвращает очередной символ, считанный с консоли. Он ожидает до тех пор, пока пользователь не нажмет клавишу, а затем возвращает результат. Возвращаемый символ относится к типу int и поэтому должен быть приведен к типу char. Если при вводе возникает ошибка, то метод Read() возвращает значение –1. Этот метод сгенерирует исключение IOException при неудачном исходе операции ввода. Ввод с консоли с помощью метода Read() буферизуется построчно, поэтому пользователь должен нажать клавишу <Enter>, прежде чем программа получит любой символ, введенный с консоли.

Ниже приведен пример программы, в которой метод Read() используется для считывания символа, введенного с клавиатуры.

```
// Считать символ, введенный с клавиатуры.

using System;

class KbIn {
  static void Main() {
    char ch;

    Console.Write("Нажмите клавишу, а затем — <ENTER>: ");

    ch = (char) Console.Read(); // получить значение типа char
    Console.WriteLine("Вы нажали клавишу: " + ch);
  }
}
```

Вот, например, к какому результату может привести выполнение этой программы.

```
Нажмите клавишу, а затем — <ENTER>: t
Вы нажали клавишу: t
```

Необходимость буферизировать построчно ввод, осуществляемый с консоли посредством метода Read(), иногда может быть досадным препятствием. Ведь при нажатии клавиши <Enter> в поток ввода передается последовательность символов перевода каретки и перевода строки. Более того, эти символы остаются во входном буфере до тех пор, пока они не будут считаны. Следовательно, в некоторых приложениях приходится удалять эти символы (путем их считывания), прежде чем приступать к следующей операции ввода. Впрочем, для чтения введенных с клавиатуры символов без построчной буферизации, можно воспользоваться рассматриваемым далее методом ReadKey().

Для считывания строки символов служит приведенный ниже метод ReadLine().

```
static string ReadLine()
```

Символы считываются методом ReadLine() до тех пор, пока пользователь не нажмет клавишу <Enter>, а затем этот метод возвращает введенные символы в виде

объекта типа `string`. Кроме того, он сгенерирует исключение `IOException` при неудачном исходе операции ввода.

Ниже приведен пример программы, в которой демонстрируется чтение строки из потока `Console.In` с помощью метода `ReadLine()`.

```
// Ввод с консоли с помощью метода ReadLine().

using System;

class ReadString {
  static void Main() {
    string str;

    Console.WriteLine("Введите несколько символов.");
    str = Console.ReadLine();
    Console.WriteLine("Вы ввели: " + str);
  }
}
```

Выполнение этой программы может привести, например, к следующему результату.

```
Введите несколько символов.
Это просто тест.
Вы ввели: Это просто тест.
```

Итак, для чтения данных из потока `Console.In` проще всего воспользоваться методами класса `Console`. Но для этой цели можно обратиться и к методам базового класса `TextReader`. В качестве примера ниже приведен переделанный вариант предыдущего примера программы, в котором используется метод `ReadLine()`, определенный в классе `TextReader`.

```
// Прочитать введенную с клавиатуры строку
// непосредственно из потока Console.In.

using System;

class ReadChars2 {
  static void Main() {
    string str;

    Console.WriteLine("Введите несколько символов.");
    str = Console.In.ReadLine(); // вызвать метод ReadLine() класса TextReader
    Console.WriteLine("Вы ввели: " + str);
  }
}
```

Обратите внимание на то, что метод `ReadLine()` теперь вызывается непосредственно для потока `Console.In`. Поэтому если требуется доступ к методам, определенным в классе `TextReader`, который является базовым для потока `Console.In`, то подобные методы вызываются так, как было показано в приведенном выше примере.

Применение метода `ReadKey()`

В состав среды .NET Framework включен метод, определяемый в классе `Console` и позволяющий непосредственно считывать отдельно введенные с клавиатуры символы без построчной буферизации. Этот метод называется `ReadKey()`. При нажа-

тии клавиши метод ReadKey() немедленно возвращает введенный с клавиатуры символ. И в этом случае пользователю уже не нужно нажимать дополнительно клавишу <Enter>. Таким образом, метод ReadKey() позволяет считывать и обрабатывать ввод с клавиатуры в реальном масштабе времени.

Ниже приведены две формы объявления метода ReadKey().

```
static ConsoleKeyInfo ReadKey()
static ConsoleKeyInfo ReadKey(bool intercept)
```

В первой форме данного метода ожидается нажатие клавиши. Когда оно происходит, метод возвращает введенный с клавиатуры символ и выводит его на экран. Во второй форме также ожидается нажатие клавиши, и затем возвращается введенный с клавиатуры символ. Но если значение параметра *intercept* равно true, то введенный символ не отображается. А если значение параметра *intercept* равно false, то введенный символ отображается.

Метод ReadKey() возвращает информацию о нажатии клавиши в объекте типа ConsoleKeyInfo, который представляет собой структуру, состоящую из приведенных ниже свойств, доступных только для чтения.

```
char KeyChar
ConsoleKey Key
ConsoleModifiers Modifiers
```

Свойство KeyChar содержит эквивалент char введенного с клавиатуры символа, свойство Key — значение из перечисления ConsoleKey всех клавиш на клавиатуре, а свойство Modifiers — описание одной из модифицирующих клавиш (<Alt>, <Ctrl> или <Shift>), которые были нажаты, если это действительно имело место, при формировании ввода с клавиатуры. Эти модифицирующие клавиши представлены в перечислении ConsoleModifiers следующими значениями: Control, Shift и Alt. В свойстве Modifiers может присутствовать несколько значений нажатых модифицирующих клавиш.

Главное преимущество метода ReadKey() заключается в том, что он предоставляет средства для организации ввода с клавиатуры в диалоговом режиме, поскольку этот ввод не буферизуется построчно. Для того чтобы продемонстрировать данный метод в действии, ниже приведен соответствующий пример программы.

```
// Считать символы, введенные с консоли, используя метод ReadKey().

using System;

class ReadKeys {
  static void Main() {
    ConsoleKeyInfo keypress;

    Console.WriteLine("Введите несколько символов, " +
                      "а по окончании - <Q>.");
    do {
      keypress = Console.ReadKey(); // считать данные о нажатых клавишах
      Console.WriteLine(" Вы нажали клавишу: " + keypress.KeyChar);

      // Проверить нажатие модифицирующих клавиш.
      if((ConsoleModifiers.Alt & keypress.Modifiers) != 0)
        Console.WriteLine("Нажата клавиша <Alt>.");
      if((ConsoleModifiers.Control & keypress.Modifiers) != 0)
```

```
        Console.WriteLine("Нажата клавиша <Control>.");
      if((ConsoleModifiers.Shift & keypress.Modifiers) != 0)
        Console.WriteLine("Нажата клавиша <Shift>.");
    } while(keypress.KeyChar != 'Q');
  }
}
```

Вот, например, к какому результату может привести выполнение этой программы.

```
Введите несколько символов, а по окончании - <Q>.
a Вы нажали клавишу: a
b Вы нажали клавишу: b
d Вы нажали клавишу: d
A Вы нажали клавишу: A
Нажата клавиша <Shift>.
B Вы нажали клавишу: B
Нажата клавиша <Shift>.
C Вы нажали клавишу: C
Нажата клавиша <Shift>.
• Вы нажали клавишу: •
Нажата клавиша <Control>.
Q Вы нажали клавишу: Q
Нажата клавиша <Shift>.
```

Как следует из приведенного выше результата, всякий раз, когда нажимается клавиша, метод `ReadKey()` немедленно возвращает введенный с клавиатуры символ. Этим он отличается от упоминавшегося ранее метода `Read()`, в котором ввод выполняется с построчной буферизацией. Поэтому если требуется добиться в программе реакции на ввод с клавиатуры, то рекомендуется выбрать метод `ReadKey()`.

Запись данных в поток вывода на консоль

Потоки `Console.Out` и `Console.Error` являются объектами типа `TextWriter`. Вывод на консоль проще всего осуществить с помощью методов `Write()` и `WriteLine()`, с которыми вы уже знакомы. Существуют варианты этих методов для вывода данных каждого из встроенных типов. В классе `Console` определяются его собственные варианты метода `Write()` и `WriteLine()`, и поэтому они могут вызываться непосредственно для класса `Console`, как это было уже не раз показано на страницах данной книги. Но при желании эти и другие методы могут быть вызваны и для класса `TextWriter`, который является базовым для потоков `Console.Out` и `Console.Error`.

Ниже приведен пример программы, в котором демонстрируется вывод в потоки `Console.Out` и `Console.Error`. По умолчанию данные в обоих случаях выводятся на консоль.

```
// Организовать вывод в потоки Console.Out и Console.Error.

using System;

class ErrOut {
  static void Main() {
    int a=10, b=0;
    int result;

    Console.Out.WriteLine("Деление на нуль приведет " +
                          "к исключительной ситуации.");
```

```
    try {
      result = a / b; // сгенерировать исключение при попытке деления на нуль
    } catch(DivideByZeroException exc) {
      Console.Error.WriteLine(exc.Message);
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
Деление на нуль приведет к исключительной ситуации.
Попытка деления на нуль.
```

Начинающие программисты порой испытывают затруднения при использовании потока `Console.Error`. Перед ними невольно встает вопрос: если оба потока, `Console.Out` и `Console.Error`, по умолчанию выводят результат на консоль, то зачем нужны два разных потока вывода? Ответ на этот вопрос заключается в том, что стандартные потоки могут быть переадресованы на другие устройства. Так, поток `Console.Error` можно переадресовать в выходной файл на диске, а не на экран. Это, например, означает, что сообщения об ошибках могут быть направлены в файл журнала регистрации, не мешая выводу на консоль. И наоборот, если вывод на консоль переадресуется, а вывод сообщений об ошибках остается прежним, то на консоли появятся сообщения об ошибках, а не выводимые на нее данные. Мы еще вернемся к вопросу переадресации после рассмотрения файлового ввода-вывода.

Класс `FileStream` и байтовый ввод-вывод в файл

В среде .NET Framework предусмотрены классы для организации ввода-вывода в файлы. Безусловно, это в основном файлы дискового типа. На уровне операционной системы файлы имеют байтовую организацию. И, как следовало ожидать, для ввода и вывода байтов в файлы имеются соответствующие методы. Поэтому ввод и вывод в файлы байтовыми потоками весьма распространен. Кроме того, байтовый поток ввода или вывода в файл может быть заключен в соответствующий объект символьного потока. Операции символьного ввода-вывода в файл находят применение при обработке текста. О символьных потоках речь пойдет далее в этой главе, а здесь рассматривается байтовый ввод-вывод.

Для создания байтового потока, привязанного к файлу, служит класс `FileStream`. Этот класс является производным от класса `Stream` и наследует всего его функции.

Напомним, что классы потоков, в том числе и `FileStream`, определены в пространстве имен `System.IO`. Поэтому в самом начале любой использующей их программы обычно вводится следующая строка кода.

```
using System.IO;
```

Открытие и закрытие файла

Для формирования байтового потока, привязанного к файлу, создается объект класса `FileStream`. В этом классе определено несколько конструкторов. Ниже приведен едва ли не самый распространенный среди них:

```
FileStream(string путь, FileMode режим)
```

где *путь* обозначает имя открываемого файла, включая полный путь к нему; а *режим* — порядок открытия файла. В последнем случае указывается одно из значений, определяемых в перечислении `FileMode` и приведенных в табл. 14.4. Как правило, этот конструктор открывает файл для доступа с целью чтения или записи. Исключением из этого правила служит открытие файла в режиме `FileMode.Append`, когда файл становится доступным только для записи.

Таблица 14.4. Значения из перечисления `FileMode`

Значение	Описание
`FileMode.Append`	Добавляет выводимые данные в конец файла
`FileMode.Create`	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
`FileMode.CreateNew`	Создает новый выходной файл. Файл с таким же именем не должен существовать
`FileMode.Open`	Открывает существующий файл
`FileMode.OpenOrCreate`	Открывает файл, если он существует. В противном случае создает новый файл
`FileMode.Truncate`	Открывает существующий файл, но сокращает его длину до нуля

Если попытка открыть файл оказывается неудачной, то генерируется исключение. Если же файл нельзя открыть из-за того что он не существует, генерируется исключение `FileNotFoundException`. А если файл нельзя открыть из-за какой-нибудь ошибки ввода-вывода, то генерируется исключение `IOException`. К числу других исключений, которые могут быть сгенерированы при открытии файла, относятся следующие: `ArgumentNullException` (указано пустое имя файла), `ArgumentException` (указано неверное имя файла), `ArgumentOutOfRangeException` (указан неверный режим), `SecurityException` (у пользователя нет прав доступа к файлу), `PathTooLongException` (слишком длинное имя файла или путь к нему), `NotSupportedException` (в имени файла указано устройство, которое не поддерживается), а также `DirectoryNotFoundException` (указан неверный каталог).

Исключения `PathTooLongException`, `DirectoryNotFoundException` и `FileNotFoundException` относятся к подклассам класса исключений `IOException`. Поэтому все они могут быть перехвачены, если перехватывается исключение `IOException`.

Ниже в качестве примера приведен один из способов открытия файла `test.dat` для ввода.

```
FileStream fin;

try {
  fin = new FileStream("test", FileMode.Open);
}
catch(IOException exc) { // перехватить все исключения, связанные с вводом-выводом
  Console.WriteLine(exc.Message);
  // Обработать ошибку.
}
catch(Exception exc { // перехватить любое другое исключение.
  Console.WriteLine(exc.Message);
  // Обработать ошибку, если это возможно.
```

```
  // Еще раз сгенерировать необрабатываемые исключения.
}
```

В первом блоке `catch` из данного примера обрабатываются ошибки, возникающие в том случае, если файл не найден, путь к нему слишком длинен, каталог не существует, а также другие ошибки ввода-вывода. Во втором блоке `catch`, который является "универсальным" для всех остальных типов исключений, обрабатываются другие вероятные ошибки (возможно, даже путем повторного генерирования исключения). Кроме того, каждую ошибку можно проверять отдельно, уведомляя более подробно о ней и принимая конкретные меры по ее исправлению.

Ради простоты в примерах, представленных в этой книге, перехватывается только исключение `IOException`, но в реальной программе, скорее всего, потребуется перехватывать и другие вероятные исключения, связанные с вводом-выводом, в зависимости от обстоятельств. Кроме того, в обработчиках исключений, приводимых в качестве примера в этой главе, просто уведомляется об ошибке, но зачастую в них должны быть запрограммированы конкретные меры по исправлению ошибок, если это вообще возможно. Например, можно предложить пользователю еще раз ввести имя файла, если указанный ранее файл не был найден. Возможно, также потребуется сгенерировать исключение повторно.

Как упоминалось выше, конструктор класса `FileStream` открывает файл, доступный для чтения или записи. Если же требуется ограничить доступ к файлу только для чтения или же только для записи, то в таком случае следует использовать такой конструктор.

```
FileStream(string путь, FileMode режим, FileAccess доступ)
```

Как и прежде, *путь* обозначает имя открываемого файла, включая и полный путь к нему, а *режим* — порядок открытия файла. В то же время *доступ* обозначает конкретный способ доступа к файлу. В последнем случае указывается одно из значений, определяемых в перечислении `FileAccess` и приведенных ниже.

```
FileAccess.Read        FileAccess.Write        FileAccess.ReadWrite
```

Например, в следующем примере кода файл `test.dat` открывается только для чтения.

```
FileStream fin = new FileStream("test.dat", FileMode.Open, FileAccess.Read);
```

По завершении работы с файлом его следует закрыть, вызвав метод `Close()`. Ниже приведена общая форма обращения к этому методу.

```
void Close()
```

При закрытии файла высвобождаются системные ресурсы, распределенные для этого файла, что дает возможность использовать их для другого файла. Любопытно, что метод `Close()` вызывает, в свою очередь, метод `Dispose()`, который, собственно, и высвобождает системные ресурсы.

ПРИМЕЧАНИЕ

Оператор `using`, рассматриваемый в главе 20, предоставляет еще один способ закрытия файла, который больше не нужен. Такой способ оказывается удобным во многих случаях обращения с файлами, поскольку гарантирует закрытие ненужного больше файла простыми средствами. Но исключительно в целях демонстрации основ обращения с файлами, в том числе и того момента, когда файл может быть закрыт, во всех примерах, представленных в этой главе, используются явные вызовы метода `Close()`.

Чтение байтов из потока файлового ввода-вывода

В классе `FileStream` определены два метода для чтения байтов из файла: `ReadByte()` и `Read()`. Так, для чтения одного байта из файла используется метод `ReadByte()`, общая форма которого приведена ниже.

```
int ReadByte()
```

Всякий раз, когда этот метод вызывается, из файла считывается один байт, который затем возвращается в виде целого значения. К числу вероятных исключений, которые генерируются при этом, относятся `NotSupportedException` (поток не открыт для ввода) и `ObjectDisposedException` (поток закрыт).

Для чтения блока байтов из файла служит метод `Read()`, общая форма которого выглядит так.

```
int Read(byte[ ] array, int offset, int count)
```

В методе `Read()` предпринимается попытка считать количество *`count`* байтов в массив *`array`*, начиная с элемента *`array[offset]`*. Он возвращает количество байтов, успешно считанных из файла. Если же возникает ошибка ввода-вывода, то генерируется исключение `IOException`. К числу других вероятных исключений, которые генерируются при этом, относится `NotSupportedException`. Это исключение генерируется в том случае, если чтение из файла не поддерживается в потоке.

В приведенном ниже примере программы метод `ReadByte()` используется для ввода и отображения содержимого текстового файла, имя которого указывается в качестве аргумента командной строки. Обратите внимание на то, что в этой программе проверяется, указано ли имя файла, прежде чем пытаться открыть его.

```
/* Отобразить содержимое текстового файла.

  Чтобы воспользоваться этой программой, укажите имя того файла,
  содержимое которого требуется отобразить. Например, для просмотра
  содержимого файла TEST.CS введите в командной строке следующее:

  ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
  static void Main(string[] args) {
    int i;
    FileStream fin;

    if(args.Length != 1) {
    Console.WriteLine("Применение: ShowFile Файл");
    return;
    }

    try {
      fin = new FileStream(args[0], FileMode.Open);
    } catch(IOException exc) {
      Console.WriteLine("Не удается открыть файл");
      Console.WriteLine(exc.Message);
      return; // Файл не открывается, завершить программу
```

```
    }

    // Читать байты до конца файла.
      try {

        do {
          i = fin.ReadByte();
          if(i != -1) Console.Write((char) i);
        } while(i != -1);

      } catch(IOException exc) {
        Console.WriteLine("Ошибка чтения файла");
        Console.WriteLine(exc.Message);
      } finally {
        fin.Close();
      }
  }
}
```

Обратите внимание на то, что в приведенной выше программе применяются два блока `try`. В первом из них перехватываются исключения, возникающие при вводе-выводе и способные воспрепятствовать открытию файла. Если произойдет ошибка ввода-вывода, выполнение программы завершится. В противном случае во втором блоке `try` будет продолжен контроль исключений, возникающих в операциях ввода-вывода. Следовательно, второй блок `try` выполняется только в том случае, если в переменной `fin` содержится ссылка на открытый файл. Обратите также внимание на то, что файл закрывается в блоке `finally`, связанном со вторым блоком `try`. Это означает, что независимо от того, как завершится цикл `do-while` (нормально или аварийно из-за ошибки), файл все равно будет закрыт. И хотя в данном конкретном примере это и так важно, поскольку программа все равно завершится в данной точке, преимущество такого подхода, вообще говоря, заключается в том, что файл закрывается в завершающем блоке `finally` в *любом* случае — даже если выполнение кода доступа к этому файлу завершается преждевременно из-за какого-нибудь исключения.

В некоторых случаях оказывается проще заключить те части программы, где осуществляется открытие и доступ к файлу, внутрь блока `try`, вместо того чтобы разделять обе эти операции. В качестве примера ниже приведен другой, более краткий вариант написания представленной выше программы `ShowFile`.

```
// Отобразить содержимое текстового файла.

using System;
using System.IO;

class ShowFile {
  static void Main(string[] args) {
    int i;
    FileStream fin = null;

    if(args.Length != 1) {
      Console.WriteLine("Применение: ShowFile File");
      return;
    }

    // Использовать один блок try для открытия файла и чтения из него
```

```
    try {
      fin = new FileStream(args[0], FileMode.Open);

      // Читать байты до конца файла.
      do {
        i = fin.ReadByte();
        if(i != -1) Console.Write((char) i);
      } while(i != -1);
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      if(fin != null) fin.Close();
    }
  }
}
```

Обратите внимание на то, что в данном варианте программы переменная `fin` ссылки на объект класса `FileStream` инициализируется пустым значением. Если файл удастся открыть в конструкторе класса `FileStream`, то значение переменной `fin` окажется непустым, а иначе — оно так и останется пустым. Это очень важно, поскольку метод `Close()` вызывается внутри блока `finally` только в том случае, если значение переменной `fin` оказывается непустым. Подобный механизм препятствует любой попытке вызвать метод `Close()` для переменной `fin`, когда она не ссылается на открытый файл. Благодаря своей компактности такой подход часто применяется во многих примерах организации ввода-вывода, приведенных далее в этой книге. Следует, однако, иметь в виду, что он не пригоден в тех случаях, когда ситуацию, возникающую в связи с невозможностью открыть файл, нужно обрабатывать отдельно. Так, если пользователь неправильно введет имя файла, то на экран, возможно, придется вывести приглашение правильно ввести имя файла, прежде чем входить в блок `try`, где осуществляется проверка правильности доступа к файлу.

В целом, порядок открытия, доступа и закрытия файла зависит от конкретного приложения. То, что хорошо в одном случае, может оказаться неприемлемым в другом. Поэтому данный процесс приходится приспосабливать к конкретным потребностям разрабатываемой программы.

Запись в файл

Для записи байта в файл служит метод `WriteByte()`. Ниже приведена его простейшая форма.

```
void WriteByte(byte value)
```

Этот метод выполняет запись в файл байта, обозначаемого параметром *value*. Если базовый поток не открывается для вывода, то генерируется исключение `NotSupportedException`. А если поток закрыт, то генерируется исключение `ObjectDisposedException`.

Для записи в файл целого массива байтов может быть вызван метод `Write()`. Ниже приведена его общая форма.

```
void Write(byte[] array, int offset, int count)
```

В методе `Write()` предпринимается попытка записать в файл количество *count* байтов из массива *array*, начиная с элемента *array*`[`*offset*`]`. Он возвращает количе-

ство байтов, успешно записанных в файл. Если во время записи возникает ошибка, то генерируется исключение `IOException`. А если базовый поток не открывается для вывода, то генерируется исключение `NotSupportedException`. Кроме того, может быть сгенерирован ряд других исключений.

Вам, вероятно, известно, что при выводе в файл выводимые данные зачастую записываются на конкретном физическом устройстве не сразу. Вместо этого они буферизуются на уровне операционной системы до тех пор, пока не накопится достаточный объем данных, чтобы записать их сразу одним блоком. Благодаря этому повышается эффективность системы. Так, на диске файлы организованы по секторам величиной от 128 байтов и более. Поэтому выводимые данные обычно буферизуются до тех пор, пока не появится возможность записать на диск сразу весь сектор.

Но если данные требуется записать на физическое устройство без предварительного накопления в буфере, то для этой цели можно вызвать метод `Flush`.

```
void Flush()
```

При неудачном исходе данной операции генерируется исключение `IOException`. Если же поток закрыт, то генерируется исключение `ObjectDisposedException`.

По завершении вывода в файл следует закрыть его с помощью метода `Close()`. Этим гарантируется, что любые выведенные данные, оставшиеся в дисковом буфере, будут записаны на диск. В этом случае отпадает необходимость вызывать метод `Flush()` перед закрытием файла.

Ниже приведен простой пример программы, в котором демонстрируется порядок записи данных в файл.

```
// Записать данные в файл.

using System;
using System.IO;

class WriteToFile {
  static void Main(string[] args) {
    FileStream fout = null;

    try {
      // Открыть выходной файл.
      fout = new FileStream("test.txt", FileMode.CreateNew);

      // Записать весь английский алфавит в файл.
      for(char c = 'A'; c <= 'Z'; c++)
        fout.WriteByte((byte) c);
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      if(fout != null) fout.Close();
    }
  }
}
```

В данной программе сначала создается выходной файл под названием `test.txt` с помощью перечисляемого значения `FileMode.CreateNew`. Это означает, что файл с таким же именем не должен уже существовать. (В противном случае генерируется исключение `IOException`.) После открытия выходного файла в него записываются

прописные буквы английского алфавита. По завершении данной программы содержимое файла `test.txt` оказывается следующим.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Использование класса `FileStream` для копирования файла

Преимущество байтового ввода-вывода средствами класса `FileStream` заключается, в частности, в том, что его можно применить к файлам практически любого типа, а не только к текстовым файлам. В качестве примера ниже приведена программа, позволяющая копировать файл любого типа, в том числе исполняемый. Имена исходного и выходного файлов указываются в командной строке.

```
/* Копировать файл.
   Чтобы воспользоваться этой программой, укажите имена исходного и выходного
   файлов. Например, для копирования файла FIRST.DAT в файл SECOND.DAT
   введите в командной строке следующее:

   CopyFile FIRST.DAT SECOND.DAT
*/

using System;
using System.IO;

class CopyFile {
  static void Main(string[] args) {
    int i;
    FileStream fin = null;
    FileStream fout = null;

    if(args.Length != 2) {
      Console.WriteLine("Применение: CopyFile Откуда Куда");
      return;
    }

    try {
       // Открыть файлы.
       fin = new FileStream(args[0], FileMode.Open);
       fout = new FileStream(args[1], FileMode.Create);

       // Скопировать файл.
       do {
         i = fin.ReadByte();
         if(i != -1) fout.WriteByte((byte)i);
       } while(i != -1);
    } catch(IOException exc) {
        Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      if(fin != null) fin.Close();
      if(fout != null) fout.Close();
    }
  }
}
```

Символьный ввод-вывод в файл

Несмотря на то что файлы часто обрабатываются побайтово, для этой цели можно воспользоваться также символьными потоками. Преимущество символьных потоков заключается в том, что они оперируют символами непосредственно в уникоде. Так, если требуется сохранить текст в уникоде, то для этого лучше всего подойдут именно символьные потоки. В целом, для выполнения операций символьного ввода-вывода в файлы объект класса `FileStream` заключается в оболочку класса `StreamReader` или `StreamWriter`. В этих классах выполняется автоматическое преобразование байтового потока в символьный и наоборот.

Не следует, однако, забывать, что на уровне операционной системы файл представляет собой набор байтов. И применение класса `StreamReader` или `StreamWriter` никак не может этого изменить.

Класс `StreamWriter` является производным от класса `TextWriter`, а класс `StreamReader` — производным от класса `TextReader`. Следовательно, в классах `StreamReader` и `StreamWriter` доступны методы и свойства, определенные в их базовых классах.

Применение класса `StreamWriter`

Для создания символьного потока вывода достаточно заключить объект класса `Stream`, например `FileStream`, в оболочку класса `StreamWriter`. В классе `StreamWriter` определено несколько конструкторов. Ниже приведен едва ли не самый распространенный среди них:

```
StreamWriter(Stream поток)
```

где *поток* обозначает имя открытого потока. Этот конструктор генерирует исключение `ArgumentException`, если *поток* не открыт для вывода, а также исключение `ArgumentNullException`, если *поток* оказывается пустым. После создания объекта класс `StreamWriter` выполняет автоматическое преобразование символов в байты.

Ниже приведен простой пример сервисной программы ввода с клавиатуры и вывода на диск набранных текстовых строк, сохраняемых в файле `test.txt`. Набираемый тест вводится до тех пор, пока в нем не встретится строка "стоп". Для символьного вывода в файл в этой программе используется объект класса `FileStream`, заключенный в оболочку класса `StreamWriter`.

```
// Простая сервисная программа ввода с клавиатуры и вывода на диск,
// демонстрирующая применение класса StreamWriter.

using System;
using System.IO;

class KtoD {
  static void Main() {
    string str;
    FileStream fout;

    // Открыть сначала поток файлового ввода-вывода.
    try {
      fout = new FileStream("test.txt", FileMode.Create);
    }
```

```
    catch(IOException exc) {
      Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
      return ;
    }

    // Заключить поток файлового ввода-вывода в оболочку класса StreamWriter.
    StreamWriter fstr_out = new StreamWriter(fout);

    try {
      Console.WriteLine("Введите текст, а по окончании — 'стоп'.");
      do {
        Console.Write(": ");
        str = Console.ReadLine();

        if(str != "стоп") {
          str = str + "\r\n"; // добавить новую строку
          fstr_out.Write(str);
        }
      } while(str != "стоп");
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      fstr_out.Close();
    }
  }
}
```

В некоторых случаях файл удобнее открывать средствами самого класса `StreamWriter`. Для этого служит один из следующих конструкторов:

```
StreamWriter(string путь)
StreamWriter(string путь, bool append)
```

где *путь* — это имя открываемого файла, включая полный путь к нему. Если во второй форме этого конструктора значение параметра *append* равно `true`, то выводимые данные присоединяются в конец существующего файла. В противном случае эти данные перезаписывают содержимое указанного файла. Но независимо от формы конструктора файл создается, если он не существует. При появлении ошибок ввода-вывода в обоих случаях генерируется исключение `IOException`. Кроме того, могут быть сгенерированы и другие исключения.

Ниже приведен вариант представленной ранее сервисной программы ввода с клавиатуры и вывода на диск, измененный таким образом, чтобы открывать выходной файл средствами самого класса `StreamWriter`.

```
// Открыть файл средствами класса StreamWriter.

using System;
using System.IO;

class KtoD {
  static void Main() {
    string str;
    StreamWriter fstr_out = null;

    try {
      // Открыть файл, заключенный в оболочку класса StreamWriter.
```

```
      fstr_out = new StreamWriter("test.txt");

      Console.WriteLine("Введите текст, а по окончании — 'стоп'.");

      do {
        Console.Write(": ");
        str = Console.ReadLine();

        if(str != "стоп") {
          str = str + "\r\n"; // добавить новую строку
          fstr_out.Write(str);
        }
      } while(str != "стоп");
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      if(fstr_out != null) fstr_out.Close();
    }
  }
}
```

Применение класса `StreamReader`

Для создания символьного потока ввода достаточно заключить байтовый поток в оболочку класса `StreamReader`. В классе `StreamReader` определено несколько конструкторов. Ниже приведен наиболее часто используемый конструктор:

```
StreamReader(Stream поток)
```

где *поток* обозначает имя открытого потока. Этот конструктор генерирует исключение `ArgumentNullException`, если *поток* оказывается пустым, а также исключение `ArgumentException`, если *поток* не открыт для ввода. После своего создания объект класса `StreamReader` выполняет автоматическое преобразование байтов в символы. По завершении ввода из потока типа `StreamReader` его нужно закрыть. При этом закрывается и базовый поток.

В приведенном ниже примере создается простая сервисная программа ввода с диска и вывода на экран содержимого текстового файла `test.txt`. Она служит дополнением к представленной ранее сервисной программе ввода с клавиатуры и вывода на диск.

```
// Простая сервисная программа ввода с диска и вывода на экран,
// демонстрирующая применение класса StreamReader.

using System;
using System.IO;

class DtoS {
  static void Main() {
    FileStream fin;
    string s;

    try {
      fin = new FileStream("test.txt", FileMode.Open);
    }
```

```
    catch(IOException exc) {
      Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
      return;
    }

    StreamReader fstr_in = new StreamReader(fin);

    try {
      while((s = fstr_in.ReadLine()) != null) {
        Console.WriteLine(s);
    }
    } catch(IOException exc) {
       Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      fstr_in.Close();
    }
  }
}
```

Обратите внимание на то, как в этой программе определяется конец файла. Когда метод `ReadLine()` возвращает пустую ссылку, это означает, что достигнут конец файла. Такой способ вполне работоспособен, но в классе `StreamReader` предоставляется еще одно средство для обнаружения конца потока — `EndOfStream`. Это доступное для чтения свойство имеет логическое значение `true`, когда достигается конец потока, в противном случае — логическое значение `false`. Следовательно, свойство `EndOfStream` можно использовать для отслеживания конца файла. В качестве примера ниже представлен другой способ организации цикла `while` для чтения из файла.

```
while(!fstr_in.EndOfStream) {
  s = fstr_in.ReadLine();
  Console.WriteLine(s);
}
```

В данном случае код немного упрощается благодаря свойству `EndOfStream`, хотя общий порядок выполнения операции ввода из файла не меняется. Иногда такое применение свойства `EndOfStream` позволяет несколько упростить сложную ситуацию, внося ясность и улучшая структуру кода.

Иногда файл проще открыть, используя непосредственно класс `StreamReader`, аналогично классу `StreamWriter`. Для этой цели служит следующий конструктор:

```
StreamReader(string путь)
```

где *путь* — это имя открываемого файла, включая полный путь к нему. Указываемый файл должен существовать. В противном случае генерируется исключение `FileNotFoundException`. Если *путь* оказывается пустым, то генерируется исключение `ArgumentNullException`. А если *путь* содержит пустую строку, то генерируется исключение `ArgumentException`. Кроме того, могут быть сгенерированы исключения `IOException` и `DirectoryNotFoundException`.

Переадресация стандартных потоков

Как упоминалось ранее, стандартные потоки, например `Console.In`, могут быть переадресованы. И чаще всего они переадресовываются в файл. Когда стандартный

поток переадресовывается, то вводимые или выводимые данные направляются в новый поток в обход устройств, используемых по умолчанию. Благодаря переадресации стандартных потоков в программе может быть организован ввод команд из дискового файла, создание файлов журнала регистрации и даже чтение входных данных из сетевого соединения.

Переадресация стандартных потоков достигается двумя способами. Прежде всего, это делается при выполнении программы из командной строки с помощью операторов < и >, переадресовывающих потоки `Console.In` и `Console.Out` соответственно. Допустим, что имеется следующая программа.

```
using System;

class Test {
  static void Main() {
    Console.WriteLine("Это тест.");
  }
}
```

Если выполнить эту программу из командной строки

```
Test > log
```

то символьная строка `"Это тест."` будет выведена в файл `log`. Аналогичным образом переадресуется ввод. Но для переадресации ввода указываемый источник входных данных должен удовлетворять требованиям программы, иначе она "зависнет".

Операторы < и >, выполняющие переадресацию из командной строки, не являются составной частью C#, а предоставляются операционной системой. Поэтому если в рабочей среде поддерживается переадресация ввода-вывода, как, например, в Windows, то стандартные потоки ввода и вывода можно переадресовать, не внося никаких изменений в программу. Тем не менее существует другой способ, позволяющий осуществлять переадресацию стандартных потоков под управлением самой программы. Для этого служат приведенные ниже методы `SetIn()`, `SetOut()` и `SetError()`, являющиеся членами класса `Console`.

```
static void SetIn(TextReader новый_поток_ввода)
static void SetOut(TextWriter новый_поток_вывода)
static void SetError(TextWriter новый_поток_сообщений_об_ошибках)
```

Таким образом, для переадресации ввода вызывается метод `SetIn()` с указанием требуемого потока. С этой целью может быть использован любой поток ввода, при условии, что он является производным от класса `TextReader`. А для переадресации вывода вызывается метод `SetOut()` с указанием требуемого потока вывода, который должен быть также производным от класса `TextReader`. Так, для переадресации вывода в файл достаточно указать объект класса `FileStream`, заключенный в оболочку класса `StreamWriter`. Соответствующий пример программы приведен ниже.

```
// Переадресовать поток Console.Out.

using System;
using System.IO;

class Redirect {
  static void Main() {
    StreamWriter log_out = null;
```

```
    try {
      log_out = new StreamWriter("logfile.txt");

      // Переадресовать стандартный вывод в файл logfile.txt.
      Console.SetOut(log_out);

      Console.WriteLine("Это начало файла журнала регистрации.");

      for(int i=0; i<10; i++) Console.WriteLine(i);

        Console.WriteLine("Это конец файла журнала регистрации.");
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
    } finally {
      if(log_out != null) log_out.Close();
    }
  }
}
```

При выполнении этой программы на экран ничего не выводится, но файл `logfile.txt` будет содержать следующее.

```
Это начало файла журнала регистрации.
0
1
2
3
4
5
6
7
8
9
Это конец файла журнала регистрации.
```

Попробуйте сами поупражняться в переадресации других встроенных потоков.

Чтение и запись двоичных данных

В приведенных ранее примерах демонстрировались возможности чтения и записи байтов или символов. Но ведь имеется также возможность (и ею пользуются часто) читать и записывать другие типы данных. Например, можно создать файл, содержащий данные типа `int`, `double` или `short`. Для чтения и записи двоичных значений встроенных в C# типов данных служат классы потоков `BinaryReader` и `BinaryWriter`. Используя эти потоки, следует иметь в виду, что данные считываются и записываются во внутреннем двоичном формате, а не в удобочитаемой текстовой форме.

Класс `BinaryWriter`

Класс `BinaryWriter` служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных. Ниже приведен наиболее часто употребляемый конструктор этого класса:

```
BinaryWriter(Stream output)
```

где *output* обозначает поток, в который выводятся записываемые данные. Для записи в выходной файл в качестве параметра *output* может быть указан объект, создаваемый средствами класса `FileStream`. Если же параметр *output* оказывается пустым, то генерируется исключение `ArgumentNullException`. А если поток, определяемый параметром *output*, не был открыт для записи данных, то генерируется исключение `ArgumentException`. По завершении вывода в поток типа `BinaryWriter` его нужно закрыть. При этом закрывается и базовый поток.

В классе `BinaryWriter` определены методы, предназначенные для записи данных всех встроенных в C# типов. Некоторые из этих методов перечислены в табл. 14.5. Обратите внимание на то, что строковые данные типа `string` записываются во внутреннем формате с указанием длины строки. Кроме того, в классе `BinaryWriter` определены стандартные методы `Close()` и `Flush()`, действующие аналогично описанному выше.

Таблица 14.5. Наиболее часто используемые методы, определенные в классе `BinaryWriter`

Метод	Описание
`void Write(sbyte value)`	Записывает значение типа `sbyte` со знаком
`void Write(byte value)`	Записывает значение типа `byte` без знака
`void Write(byte[] buffer)`	Записывает массив значений типа `byte`
`void Write(short value)`	Записывает целочисленное значение типа `short` (короткое целое)
`void Write(ushort value)`	Записывает целочисленное значение типа `ushort` (короткое целое без знака)
`void Write(int value)`	Записывает целочисленное значение типа `int`
`void Write(uint value)`	Записывает целочисленное значение типа `uint` (целое без знака)
`void Write(long value)`	Записывает целочисленное значение типа `long` (длинное целое)
`void Write(ulong value)`	Записывает целочисленное значение типа `ulong` (длинное целое без знака)
`void Write(float value)`	Записывает значение типа `float` (с плавающей точкой одинарной точности)
`void Write(double value)`	Записывает значение типа `double` (с плавающей точкой двойной точности)
`void Write(decimal value)`	Записывает значение типа `decimal` (с двумя десятичными разрядами после запятой)
`void Write(char ch)`	Записывает символ
`void Write(char[] buffer)`	Записывает массив символов
`void Write(string value)`	Записывает строковое значение типа `string`, представленное во внутреннем формате с указанием длины строки

Класс `BinaryReader`

Класс `BinaryReader` служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных. Ниже приведен наиболее часто употребляемый конструктор этого класса:

```
BinaryReader(Stream input)
```

где *input* обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра *input* может быть указан объект, создаваемый средствами класса `FileStream`. Если же поток, определяемый параметром *input*, не был открыт для чтения данных или оказался недоступным по иным причинам, то генерируется исключение `ArgumentException`. По завершении ввода из потока типа `BinaryReader` его нужно закрыть. При этом закрывается и базовый поток.

В классе `BinaryReader` определены методы, предназначенные для чтения данных всех встроенных в C# типов. Некоторые из этих методов перечислены в табл. 14.6. Следует, однако, иметь в виду, что в методе `ReadString()` считывается символьная строка, хранящаяся во внутреннем формате с указанием ее длины. Все методы данного класса генерируют исключение `IOException`, если возникает ошибка ввода. Кроме того, могут быть сгенерированы и другие исключения.

Таблица 14.6. Наиболее часто используемые методы, определенные в классе `BinaryReader`

Метод	Описание
`bool ReadBoolean()`	Считывает значение логического типа `bool`
`byte ReadByte()`	Считывает значение типа `byte`
`sbyte ReadSByte()`	Считывает значение типа `sbyte`
`byte[] ReadBytes(int count)`	Считывает количество *count* байтов и возвращает их в виде массива
`char ReadChar()`	Считывает значение типа `char`
`char[] ReadChars(int count)`	Считывает количество *count* символов и возвращает их в виде массива
`decimal ReadDecimal()`	Считывает значение типа `decimal`
`double ReadDouble()`	Считывает значение типа `double`
`float ReadSingle()`	Считывает значение типа `float`
`short ReadInt16()`	Считывает значение типа `short`
`int ReadInt32()`	Считывает значение типа `int`
`long ReadInt64()`	Считывает значение типа `long`
`ushort ReadUInt16()`	Считывает значение типа `ushort`
`uint ReadUInt32()`	Считывает значение типа `uint`
`ulong ReadUInt64()`	Считывает значение типа `ulong`
`string ReadString()`	Считывает значение типа `string`, представленное во внутреннем двоичном формате с указанием длины строки. Этот метод следует использовать для считывания строки, которая была записана средствами класса `BinaryWriter`

В классе `BinaryWriter` определены также три приведенных ниже варианта метода `Read()`.

При неудачном исходе операции чтения эти методы генерируют исключение `IOException`. Кроме того, в классе `BinaryReader` определен стандартный метод `Close()`.

Метод	Описание
`int Read()`	Возвращает целочисленное представление следующего доступного символа из вызывающего потока ввода. При обнаружении конца файла возвращает значение -1
`int Read(byte[] buffer, int offset, int count)`	Делает попытку прочитать количество `count` байтов в массив `buffer`, начиная с элемента `buffer[offset]`, и возвращает количество успешно считанных байтов
`int Read(char[]buffer, int offset, int count)`	Делает попытку прочитать количество `count` символов в массив `buffer`, начиная с элемента `buffer[offset]`, и возвращает количество успешно считанных символов

Демонстрирование двоичного ввода-вывода

Ниже приведен пример программы, в котором демонстрируется применение классов `BinaryReader` и `BinaryWriter` для двоичного ввода-вывода. В этой программе в файл записываются и считываются обратно данные самых разных типов.

```
// Записать двоичные данные, а затем считать их обратно.

using System;
using System.IO;

class RWData {
  static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    int i = 10;
    double d = 1023.56;
    bool b = true;
    string str = "Это тест";

    // Открыть файл для вывода.
    try {
      dataOut = new
      BinaryWriter(new FileStream("testdata", FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
      return;
    }

    // Записать данные в файл.
    try {
      Console.WriteLine("Запись " + i);
      dataOut.Write(i);

      Console.WriteLine("Запись " + d);
      dataOut.Write(d);

      Console.WriteLine("Запись " + b);
      dataOut.Write(b);
```

```
      Console.WriteLine("Запись " + 12.2 * 7.4);
      dataOut.Write(12.2 * 7.4);

      Console.WriteLine("Запись " + str);
      dataOut.Write(str);
    }
    catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      dataOut.Close();
    }

  Console.WriteLine();

    // А теперь прочитать данные из файла.
    try {
      dataIn = new
      BinaryReader(new FileStream("testdata", FileMode.Open));
    }
    catch(IOException exc) {
      Console.WriteLine("Ошибка открытия файла:\n" + exc.Message);
      return;
    }

    try {
      i = dataIn.ReadInt32();
      Console.WriteLine("Чтение " + i);
      d = dataIn.ReadDouble();

      Console.WriteLine("Чтение " + d);
      b = dataIn.ReadBoolean();

      Console.WriteLine("Чтение " + b);
      d = dataIn.ReadDouble();

      Console.WriteLine("Чтение " + d);
      str = dataIn.ReadString();

      Console.WriteLine("Чтение " + str);
    }
    catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода:\n" + exc.Message);
    } finally {
      dataIn.Close();
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Запись 10
Запись 1023.56
Запись True
Запись 90.28
Запись Это тест
```

```
Чтение 10
Чтение 1023.56
Чтение True
Чтение 90.28
Чтение Это тест
```

Если просмотреть содержимое файла `testdata`, который получается при выполнении этой программы, то можно обнаружить, что он содержит данные в двоичной, а не в удобочитаемой текстовой форме.

Далее следует более практический пример, демонстрирующий, насколько эффективным может быть двоичный ввод-вывод. Для учета каждого предмета хранения на складе в приведенной ниже программе сначала запоминается наименование предмета, имеющееся в наличии, количество и стоимость, а затем пользователю предлагается ввести наименование предмета, чтобы найти его в базе данных. Если предмет найден, отображаются сведения о его запасах на складе.

```
/* Использовать классы BinaryReader и BinaryWriter для
   реализации простой программы учета товарных запасов. */

using System;
using System.IO;

class Inventory {
  static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    string item; // наименование предмета
    int onhand;  // имеющееся в наличии количество
    double cost; // цена

    try {
      dataOut = new
      BinaryWriter(new FileStream("inventory.dat", FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine("Не удается открыть файл " +
                        "товарных запасов для вывода");
      Console.WriteLine("Причина: " + exc.Message);
      return;
    }

    // Записать данные о товарных запасах в файл.
    try {
      dataOut.Write("Молотки");
      dataOut.Write(10);
      dataOut.Write(3.95);

      dataOut.Write("Отвертки");
      dataOut.Write(18);
      dataOut.Write(1.50);

      dataOut.Write("Плоскогубцы");
      dataOut.Write(5);
```

```
    dataOut.Write(4.95);

    dataOut.Write("Пилы");
    dataOut.Write(8);
    dataOut.Write(8.95);
  }
  catch(IOException exc) {
    Console.WriteLine("Ошибка записи в файл товарных запасов");
    Console.WriteLine("Причина: " + exc.Message);
  } finally {
    dataOut.Close();
  }

  Console.WriteLine();

  // А теперь открыть файл товарных запасов для чтения.
  try {
    dataIn = new
    BinaryReader(new FileStream("inventory.dat", FileMode.Open));
  }
  catch(IOException exc) {
    Console.WriteLine("Не удается открыть файл " +
                      "товарных запасов для ввода");
    Console.WriteLine("Причина: " + exc.Message);
    return;
  }

  // Найти предмет, введенный пользователем.
  Console.Write("Введите наименование для поиска: ");
  string what = Console.ReadLine();
  Console.WriteLine();

  try {
    for(;;) {
      // Читать данные о предмете хранения.
      item = dataIn.ReadString();
      onhand = dataIn.ReadInt32();
      cost = dataIn.ReadDouble();

      // Проверить, совпадает ли он с запрашиваемым предметом.
      // Если совпадает, то отобразить сведения о нем.

      if(item.Equals(what, StringComparison.OrdinalIgnoreCase)) {
        Console.WriteLine(item + ": " + onhand + " штук в наличии. " +
                          "Цена: {0:C} за штуку", cost);
        Console.WriteLine("Общая стоимость по наименованию <{0}>: {1:C}.",
                          item, cost * onhand);
        break;
      }
    }
  }
  catch(EndOfStreamException) {
    Console.WriteLine("Предмет не найден.");
  }
```

```
    catch(IOException exc) {
      Console.WriteLine("Ошибка чтения из файла товарных запасов");
      Console.WriteLine("Причина: " + exc.Message);
    } finally {
      dataIn.Glose();
    }
  }
}
```

Выполнение этой программы может привести, например, к следующему результату.

```
Введите наименование для поиска: Отвертки

Отвертки: 18 штук в наличии. Цена: $1.50 за штуку.
Общая стоимость по наименованию <Отвертки>: $27.00.
```

Обратите внимание на то, что сведения о товарных запасах сохраняются в этой программе в двоичном формате, а не в удобной для чтения текстовой форме. Благодаря этому обработка числовых данных может выполняться без предварительного их преобразования из текстовой формы.

Обратите также внимание на то, как в этой программе обнаруживается конец файла. Методы двоичного ввода генерируют исключение `EndOfStreamException` по достижении конца потока, и поэтому файл читается до тех пор, пока не будет найден искомый предмет или сгенерировано данное исключение. Таким образом, для обнаружения конца файла никакого специального механизма не требуется.

Файлы с произвольным доступом

В предыдущих примерах использовались *последовательные файлы*, т.е. файлы со строго линейным доступом, байт за байтом. Но доступ к содержимому файла может быть и произвольным. Для этого служит, в частности, метод `Seek()`, определенный в классе `FileStream`. Этот метод позволяет установить *указатель положения в файле*, или так называемый *указатель файла*, на любое место в файле. Ниже приведена общая форма метода `Seek()`:

```
long Seek(long offset, SeekOrigin origin)
```

где *`offset`* обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (*`origin`*). В качестве *`origin`* может быть указано одно из приведенных ниже значений, определяемых в перечислении `SeekOrigin`.

Значение	Описание
`SeekOrigin.Begin`	Поиск от начала файла
`SeekOrigin.Current`	Поиск от текущего положения
`SeekOrigin.End`	Поиск от конца файла

Следующая операция чтения или записи после вызова метода `Seek()` будет выполняться, начиная с нового положения в файле, возвращаемого этим методом. Если во время поиска в файле возникает ошибка, то генерируется исключение `IOException`. Если же запрос положения в файле не поддерживается базовым потоком, то генерируется исключение `NotSupportedException`. Кроме того, могут быть сгенерированы и другие исключения.

В приведенном ниже примере программы демонстрируется ввод-вывод в файл с произвольным доступом. Сначала в файл записываются прописные буквы английского алфавита, а затем его содержимое считывается обратно в произвольном порядке.

```
// Продемонстрировать произвольный доступ к файлу.

using System;
using System.IO;

class RandomAccessDemo {
  static void Main() {
    FileStream f = null;
    char ch;

    try {
      f = new FileStream("random.dat", FileMode.Create);
      // Записать английский алфавит в файл.
      for(int i=0; i < 26; i++)
        f.WriteByte((byte)('A'+i));

      // А теперь считать отдельные буквы английского алфавита.
      f.Seek(0, SeekOrigin.Begin); // найти первый байт
      ch = (char) f.ReadByte();
      Console.WriteLine("Первая буква: " + ch);

      f.Seek(1, SeekOrigin.Begin); // найти второй байт
      ch = (char) f.ReadByte();
      Console.WriteLine("Вторая буква: " + ch);

      f.Seek(4, SeekOrigin.Begin); // найти пятый байт
      ch = (char) f.ReadByte();
      Console.WriteLine("Пятая буква: " + ch);

      Console.WriteLine();

      // А теперь прочитать буквы английского алфавита через одну.
      Console.WriteLine("Буквы алфавита через одну: ");
      for(int i=0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // найти i-й символ
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
      }
    }
    catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
    } finally {
      if(f != null) f.Close();
    }

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Первая буква: A
Вторая буква: B
Пятая буква: E

Буквы алфавита через одну:
A C E G I K M O Q S U W Y
```

Несмотря на то что метод `Seek()` имеет немало преимуществ при использовании с файлами, существует и другой способ установки текущего положения в файле с помощью свойства `Position`. Как следует из табл. 14.2, свойство `Position` доступно как для чтения, так и для записи. Поэтому с его помощью можно получить или же установить текущее положение в файле. В качестве примера ниже приведен фрагмент кода из предыдущей программы записи и чтения из файла с произвольным доступом `random.dat`, измененный с целью продемонстрировать применение свойства `Position`.

```
Console.WriteLine("Буквы алфавита через одну: ");
for(int i=0; i < 26; i += 2) {
    f.Position = i; // найти i-й символ посредством свойства Position
    ch = (char) f.ReadByte();
    Console.Write(ch + " ");
}
```

Применение класса `MemoryStream`

Иногда оказывается полезно читать вводимые данные из массива или записывать выводимые данные в массив, а не вводить их непосредственно из устройства или выводить прямо на него. Для этой цели служит класс `MemoryStream`. Он представляет собой реализацию класса `Stream`, в которой массив байтов используется для ввода и вывода. В классе `MemoryStream` определено несколько конструкторов. Ниже представлен один из них:

```
MemoryStream(byte[ ] buffer)
```

где *`buffer`* обозначает массив байтов, используемый в качестве источника или адресата в запросах ввода-вывода. Используя этот конструктор, следует иметь в виду, что массив *`buffer`* должен быть достаточно большим для хранения направляемых в него данных.

В качестве примера ниже приведена программа, демонстрирующая применение класса `MemoryStream` в операциях ввода-вывода.

```
// Продемонстрировать применение класса MemoryStream.

using System;
using System.IO;

class MemStrDemo {
  static void Main() {
    byte[] storage = new byte[255];

    // Создать запоминающий поток.
    MemoryStream memstrm = new MemoryStream(storage);

    // Заключить объект memstrm в оболочки классов
```

```
    // чтения и записи данных в потоки.
    StreamWriter memwtr = new StreamWriter(memstrm);
    StreamReader memrdr = new StreamReader(memstrm);

    try {
    // Записать данные в память, используя объект memwtr.
    for(int i=0; i < 10; i++)
      memwtr.WriteLine("byte [" + i + "]: " + i);

      // Поставить в конце точку.
      memwtr.WriteLine(".");

      memwtr.Flush();

      Console.WriteLine("Чтение прямо из массива storage: ");

      // Отобразить содержимое массива storage непосредственно.
      foreach(char ch in storage) {
        if (ch == '.') break;
        Console.Write(ch);
      }

      Console.WriteLine("\nЧтение из потока с помощью объекта memrdr: ");

      // Читать из объекта memstrm средствами ввода данных из потока.
      memstrm.Seek(0, SeekOrigin.Begin); // установить указатель файла
                                         // в исходное положение
      string str = memrdr.ReadLine();
      while(str != null) {
        str = memrdr.ReadLine();
        if(str[0] == '.') break;
        Console.WriteLine(str);
      }
    } catch(IOException exc) {
       Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
    } finally {
       // Освободить ресурсы считывающего и записывающего потоков.
       memwtr.Close();
       memrdr.Close();
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Чтение прямо из массива storage:
byte [0]: 0
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

```
Чтение из потока с помощью объекта memrdr:
byte [1]: 1
byte [2]: 2
byte [3]: 3
byte [4]: 4
byte [5]: 5
byte [6]: 6
byte [7]: 7
byte [8]: 8
byte [9]: 9
```

В этой программе сначала создается массив байтов, называемый `storage`. Затем этот массив используется в качестве основной памяти для объекта `memstrm` класса `MemoryStream`. Из объекта `memstrm`, в свою очередь, создаются объекты `memrdr` класса `StreamReader` и `memwtr` класса `StreamWriter`. С помощью объекта `memwtr` выводимые данные записываются в запоминающий поток. Обратите внимание на то, что после записи выводимых данных для объекта `memwtr` вызывается метод `Flush()`. Это необходимо для того, чтобы содержимое буфера этого объекта записывалось непосредственно в базовый массив. Далее содержимое базового массива байтов отображается вручную в цикле `foreach`. После этого указатель файла устанавливается с помощью метода `Seek()` в начало запоминающего потока, из которого затем вводятся данные с помощью объекта потока `memrdr`.

Запоминающие потоки очень полезны для программирования. С их помощью можно, например, организовать сложный вывод с предварительным накоплением данных в массиве до тех пор, пока они не понадобятся. Этот прием особенно полезен для программирования в такой среде с графическим пользовательским интерфейсом, как Windows. Кроме того, стандартный поток может быть переадресован из массива. Это может пригодиться, например, для подачи тестовой информации в программу.

Применение классов `StringReader` и `StringWriter`

Для выполнения операций ввода-вывода с запоминанием в некоторых приложениях в качестве базовой памяти иногда лучше использовать массив типа `string`, чем массив типа `byte`. Именно для таких случаев и предусмотрены классы `StringReader` и `StringWriter`. В частности, класс `StringReader` наследует от класса `TextReader`, а класс `StringWriter` — от класса `TextWriter`. Следовательно, они представляют собой потоки, имеющие доступ к методам, определенным в этих двух базовых классах, что позволяет, например, вызывать метод `ReadLine()` для объекта класса `StringReader`, а метод `WriteLine()` — для объекта класса `StringWriter`.

Ниже приведен конструктор класса `StringReader`:

```
StringReader(string s)
```

где *s* обозначает символьную строку, из которой производится чтение.

В классе `StringWriter` определено несколько конструкторов. Ниже представлен один из наиболее часто используемых.

```
StringWriter()
```

Этот конструктор создает записывающий поток, который помещает выводимые данные в строку. Для получения содержимого этой строки достаточно вызвать метод `ToString()`.

Ниже приведен пример, демонстрирующий применение классов `StringReader` и `StringWriter`.

```
// // Продемонстрировать применение классов StringReader и StringWriter.

using System;
using System.IO;

class StrRdrWtrDemo {
  static void Main() {
    StringWriter strwtr = null;
    StringReader strrdr = null;

    try {
      // Создать объект класса StringWriter.
      strwtr = new StringWriter();

      // Вывести данные в записывающий поток типа StringWriter.
      for(int i=0; i < 10; i++)
        strwtr.WriteLine("Значение i равно: " + i);

      // Создать объект класса StringReader.
      strrdr = new StringReader(strwtr.ToString());

      // А теперь ввести данные из считывающего потока типа StringReader.
      string str = strrdr.ReadLine();
      while(str != null) {
        str = strrdr.ReadLine();
        Console.WriteLine(str);
      }
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода\n" + exc.Message);
    } finally {
      // Освободить ресурсы считывающего и записывающего потоков.
      if(strrdr != null) strrdr.Close();
      if(strwtr != null) strwtr.Close();
    }
  }
}
```

Вот к каком результату приводит выполнение этого кода.

```
Значение i равно: 1
Значение i равно: 2
Значение i равно: 3
Значение i равно: 4
Значение i равно: 5
Значение i равно: 6
Значение i равно: 7
Значение i равно: 8
Значение i равно: 9
```

В данном примере сначала создается объект `strwtr` класса `StringWriter`, в который выводятся данные с помощью метода `WriteLine()`. Затем создается объект класса `StringReader` с использованием символьной строки, содержащейся в объекте `strwtr`. Эта строка получается в результате вызова метода `ToString()` для объекта `strwtr`. И наконец, содержимое данной строки считывается с помощью метода `ReadLine()`.

Класс `File`

В среде .NET Framework определен класс `File`, который может оказаться полезным для работы с файлами, поскольку он содержит несколько статических методов, выполняющих типичные операции над файлами. В частности, в классе `File` имеются методы для копирования и перемещения, шифрования и расшифровывания, удаления файлов, а также для получения и задания информации о файлах, включая сведения об их существовании, времени создания, последнего доступа и различные атрибуты файлов (только для чтения, скрытых и пр.). Кроме того, в классе `File` имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа `FileStream` на него. В классе `File` содержится слишком много методов для подробного их рассмотрения, поэтому мы уделим внимание только трем из них. Сначала будет представлен метод `Copy()`, а затем — методы `Exists()` и `GetLastAccessTime()`. На примере этих методов вы сможете получить ясное представление о том, насколько удобны методы, доступные в классе `File`. И тогда вам станет ясно, что класс `File` определенно заслуживает более тщательного изучения.

СОВЕТ

Ряд методов для работы с файлами определен также в классе `FileInfo`. Этот класс отличается от класса `File` одним, очень важным преимуществом: для операций над файлами он предоставляет методы экземпляра и свойства, а не статические методы. Поэтому для выполнения нескольких операций над одним и тем же файлом лучше воспользоваться классом `FileInfo`.

Копирование файлов с помощью метода `Copy()`

Ранее в этой главе демонстрировался пример программы, в которой файл копировался вручную путем чтения байтов из одного файла и записи в другой. И хотя задача копирования файлов не представляет особых трудностей, ее можно полностью автоматизировать с помощью метода `Copy()`, определенного в классе `File`. Ниже представлены две формы его объявления.

```
static void Copy(string имя_исходного_файла, string имя_целевого_файла)
static void Copy(string имя_исходного_файла, string имя_целевого_файла,
                 boolean overwrite)
```

Метод `Copy()` копирует файл, на который указывает *имя_исходного_файла*, в файл, на который указывает *имя_целевого_файла*. В первой форме данный метод копирует файл только в том случае, если файл, на который указывает *имя_целевого_файла*, еще не существует. А во второй форме копия заменяет и перезаписывает целевой файл, если он существует и если параметр *overwrite* принимает логическое значение `true`. Но в обоих случаях может быть сгенерировано несколько видов исключений, включая `IOException` и `FileNotFoundException`.

В приведенном ниже примере программы метод Copy() применяется для копирования файла. Имена исходного и целевого файлов указываются в командной строке. Обратите внимание, насколько эта программа короче демонстрировавшейся ранее. Кроме того, она более эффективна.

```
/* Скопировать файл, используя метод File.Copy().

   Чтобы воспользоваться этой программой, укажите имя исходного и целевого
   файлов. Например, чтобы скопировать файл FIRST.DAT в файл SECOND.DAT,
   введите в командной строке следующее:

   CopyFile FIRST.DAT SECOND.DAT
*/

using System;
using System.IO;

class CopyFile {
  static void Main(string[] args) {
    if(args.Length != 2) {
     Console.WriteLine("Применение: CopyFile Откуда Куда");
     return;
    }

    // Копировать файлы.
    try {
      File.Copy(args[0], args[1]);
    } catch(IOException exc) {
      Console.WriteLine("Ошибка копирования файла\n" + exc.Message);
    }
  }
}
```

Как видите, в этой программе не нужно создавать поток типа FileStream или освобождать его ресурсы. Все это делается в методе Copy() автоматически. Обратите также внимание на то, что в данной программе существующий файл не перезаписывается. Поэтому если целевой файл должен быть перезаписан, то для этой цели лучше воспользоваться второй из упоминавшихся ранее форм метода Copy().

Применение методов Exists() и GetLastAccessTime()

С помощью методов класса File очень легко получить нужные сведения о файле. Рассмотрим два таких метода: Exists() и GetLastAccessTime(). Метод Exists() определяет, существует ли файл, а метод GetLastAccessTime() возвращает дату и время последнего доступа к файлу. Ниже приведены формы объявления обоих методов.

```
static bool Exists(string путь)
static DateTime GetLastAccessTime(string путь)
```

В обоих методах *путь* обозначает файл, сведения о котором требуется получить. Метод Exists() возвращает логическое значение true, если файл существует и доступен для вызывающего процесса. А метод GetLastAccessTime() возвращает структуру DateTime, содержащую дату и время последнего доступа к файлу. (Структура

DateTime описывается далее в этой книге, но метод ToString() автоматически приводит дату и время к удобочитаемому виду.) С указанием недействительных аргументов или прав доступа при вызове обоих рассматриваемых здесь методов может быть связан целый ряд исключений, но в действительности генерируется только исключение IOException.

В приведенном ниже примере программы методы Exists() и GetLastAccessTime() демонстрируются в действии. В этой программе сначала определяется, существует ли файл под названием test.txt. Если он существует, то на экран выводит время последнего доступа к нему.

```
// Применить методы Exists() и GetLastAccessTime().

using System;
using System.IO;

class ExistsDemo {
  static void Main() {
    if(File.Exists("test.txt"))
      Console.WriteLine("Файл существует. В последний раз он был доступен " +
                        File.GetLastAccessTime("test.txt"));
     else
    Console.WriteLine("Файл не существует");
  }
}
```

Кроме того, время создания файла можно выяснить, вызвав метод GetCreationTime(), а время последней записи в файл, вызвав метод GetLastWriteTime(). Имеются также варианты этих методов для представления данных о файле в формате всеобщего скоординированного времени (UTC). Попробуйте поэкспериментировать с ними.

Преобразование числовых строк в их внутреннее представление

Прежде чем завершить обсуждение темы ввода-вывода, рассмотрим еще один способ, который может пригодиться при чтении числовых строк. Как вам должно быть уже известно, метод WriteLine() предоставляет удобные средства для вывода различных типов данных на консоль, включая и числовые значения встроенных типов, например int или double. При этом числовые значения автоматически преобразуются методом WriteLine() в удобную для чтения текстовую форму. В то же время аналогичный метод ввода для чтения и преобразования строк с числовыми значениями в двоичный формат их внутреннего представления не предоставляется. В частности, отсутствует вариант метода Read() специально для чтения строки "100", введенной с клавиатуры, и автоматического ее преобразования в соответствующее двоичное значение, которое может быть затем сохранено в переменной типа int. Поэтому данную задачу приходится решать другими способами. И самый простой из них — воспользоваться методом Parse(), определенным для всех встроенных числовых типов данных.

Прежде всего необходимо отметить следующий важный факт: все встроенные в C# типы данных, например int или double, на самом деле являются не более чем *псевдонимами* (т.е. другими именами) структур, определяемых в среде .NET Framework. В действительности тип в C# невозможно отличить от типа структуры в среде .NET Framework, поскольку один просто носит имя другого. В C# для поддержки значений

простых типов используются структуры, и поэтому для типов этих значений имеются специально определенные члены структур.

Ниже приведены имена структур .NET и их эквиваленты в виде ключевых слов C# для числовых типов данных.

Имя структуры в .NET	Имя типа данных в C#
Decimal	decimal
Double	double
Single	float
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
UInt64	ulong
Byte	byte
Sbyte	sbyte

Эти структуры определены в пространстве имен `System`. Следовательно, имя структуры `Int32` полностью определяется как `System.Int32`. Эти структуры предоставляют обширный ряд методов, помогающих полностью интегрировать значения простых типов в иерархию объектов C#. А кроме того, в числовых структурах определяется статический метод `Parse()`, преобразующий числовую строку в соответствующий двоичный эквивалент.

Существует несколько перегружаемых форм метода `Parse()`. Ниже приведены его простейшие варианты для каждой числовой структуры. Они выполняют преобразование с учетом местной специфики представления чисел. Следует иметь в виду, что каждый метод возвращает двоичное значение, соответствующее преобразуемой строке.

Структура	Метод преобразования
Decimal	`static decimal Parse(string s)`
Double	`static double Parse(string s)`
Single	`static float Parse(string s)`
Int64	`static long Parse(string s)`
Int32	`static int Parse(string s)`
Int16	`static short Parse(string s)`
UInt64	`static ulong Parse(string s)`
UInt32	`static uint Parse(string s)`
UInt16	`static ushort Parse(string s)`
Byte	`static byte Parse(string s)`
Sbyte	`static sbyte Parse(string s)`

Приведенные выше варианты метода `Parse()` генерируют исключение `FormatException`, если строка *s* не содержит допустимое число, определяемое вызывающим типом данных. А если она содержит пустое значение, то генерируется исключение `ArgumentNullException`. Когда же значение в строке *s* превышает допустимый диапазон чисел для вызывающего типа данных, то генерируется исключение `OverflowException`.

Методы синтаксического анализа позволяют без особого труда преобразовать числовое значение, введенное с клавиатуры или же считанное из текстового файла в виде строки, в соответствующий внутренний формат. В качестве примера ниже приведена программа, в которой усредняется ряд чисел, вводимых пользователем. Сначала пользователю предлагается указать количество усредняемых значений, а затем это количество считывается методом `ReadLine()` и преобразуется из строки в целое число методом `Int32.Parse()`. Далее вводятся отдельные значения, преобразуемые методом `Double.Parse()` из строки в их эквивалент типа `double`.

```
// Эта программа усредняет ряд чисел, вводимых пользователем.

using System;
using System.IO;

class AvgNums {
  static void Main() {
    string str;
    int n;
    double sum = 0.0;
    double avg, t;

    Console.Write("Сколько чисел вы собираетесь ввести: ");
    str = Console.ReadLine();
    try {
      n = Int32.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    } catch(OverflowException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.WriteLine("Введите " + n + " чисел.");
    for(int i=0; i < n ; i++) {
      Console.Write(": ");
      str = Console.ReadLine();
      try {
        t = Double.Parse(str);
      } catch(FormatException exc) {
        Console.WriteLine(exc.Message);
        t = 0.0;
      } catch(OverflowException exc) {
        Console.WriteLine(exc.Message);
        t = 0;
      }
      sum += t;
    }
    avg = sum / n;
    Console.WriteLine("Среднее равно " + avg);
  }
}
```

Выполнение этой программы может привести, например, к следующему результату.

```
Сколько чисел вы собираетесь ввести: 5
Введите 5 чисел.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Среднее равно 3.3
```

Следует особо подчеркнуть, что для каждого преобразуемого значения необходимо выбирать подходящий метод синтаксического анализа. Так, если попытаться преобразовать строку, содержащую значение с плавающей точкой, методом `Int32.Parse()`, то искомый результат, т.е. числовое значение с плавающей точкой, получить не удастся.

Как пояснялось выше, при неудачном исходе преобразования метод `Parse()` сгенерирует исключение. Для того чтобы избежать генерирования исключений при преобразовании числовых строк, можно воспользоваться методом `TryParse()`, определенным для всех числовых структур. В качестве примера ниже приведен один из вариантов метода `TryParse()`, определяемых в структуре `Int32`:

```
static bool TryParse(string s, out int результат)
```

где *s* обозначает числовую строку, передаваемую данному методу, который возвращает соответствующий *результат* после преобразования с учетом выбираемой по умолчанию местной специфики представления чисел. (Конкретную местную специфику представления чисел с учетом региональных стандартов можно указать в другом варианте данного метода.) При неудачном исходе преобразования, например, когда параметр `s` не содержит числовую строку в надлежащей форме, метод `TryParse()` возвращает логическое значение `false`. В противном случае он возвращает логическое значение `true`. Следовательно, значение, возвращаемое этим методом, обязательно следует проверить, чтобы убедиться в удачном (или неудачном) исходе преобразования.

15 ГЛАВА

Делегаты, события и лямбда-выражения

В этой главе рассматриваются три новых средства C#: делегаты, события и лямбда-выражения. *Делегат* предоставляет возможность инкапсулировать метод, а *событие* уведомляет о том, что произошло некоторое действие. Делегаты и события тесно связаны друг с другом, поскольку событие основывается на делегате. Оба средства расширяют круг прикладных задача, решаемых при программировании на C#. А *лямбда-выражение* представляет собой новое синтаксическое средство, обеспечивающее упрощенный, но в то же время эффективный способ определения того, что по сути является единицей исполняемого кода. Лямбда-выражения обычно служат для работы с делегатами и событиями, поскольку делегат может ссылаться на лямбда-выражение. (Кроме того, лямбда-выражения очень важны для языка LINQ, описываемого в главе 19.) В данной главе рассматриваются также анонимные методы, ковариантность, контравариантность и групповые преобразования методов.

Делегаты

Начнем с определения понятия делегата. Попросту говоря, *делегат* представляет собой объект, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается. Ниже будет показано, насколько действенным оказывается такой принцип.

Следует особо подчеркнуть, что один и тот же делегат может быть использован для вызова разных методов во время выполнения программы, для чего достаточно изменить метод, на который ссылается делегат. Таким образом, метод, вызываемый делегатом, определяется во время выполнения, а не в процессе компиляции. В этом, собственно, и заключается главное преимущество делегата.

ПРИМЕЧАНИЕ

Если у вас имеется опыт программирования на C/C++, то вам полезно будет знать, что делегат в C# подобен указателю на функцию в C/C++.

Тип делегата объявляется с помощью ключевого слова `delegate`. Ниже приведена общая форма объявления делегата:

```
delegate возвращаемый_тип имя(список_параметров);
```

где *возвращаемый_тип* обозначает тип значения, возвращаемого методами, которые будут вызываться делегатом; *имя* — конкретное имя делегата; *список_параметров* — параметры, необходимые для методов, вызываемых делегатом. Как только будет создан экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый тип и параметры которых соответствуют указанным в объявлении делегата.

Самое главное, что делегат может служить для вызова *любого* метода с соответствующей сигнатурой и возвращаемым типом. Более того, вызываемый метод может быть методом экземпляра, связанным с отдельным объектом, или же статическим методом, связанным с конкретным классом. Значение имеет лишь одно: возвращаемый тип и сигнатура метода должны быть согласованы с теми, которые указаны в объявлении делегата.

Для того чтобы показать делегат в действии, рассмотрим для начала простой пример его применения.

```
// Простой пример применения делегата.

using System;

// Объявить тип делегата.
delegate string StrMod(string str);

class DelegateTest {
  // Заменить пробелы дефисами.
  static string ReplaceSpaces(string s) {
    Console.WriteLine("Замена пробелов дефисами.");
    return s.Replace(' ', '-');
  }

  // Удалить пробелы.
  static string RemoveSpaces(string s) {
    string temp = "";
    int i;

    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < s.Length; i++)
      if(s[i] != ' ') temp += s[i];
```

```
    return temp;
  }

  // Обратить строку.
  static string Reverse(string s) {
    string temp = "";
    int i, j;

    Console.WriteLine("Обращение строки.");
    for(j=0, i=s.Length-1; i >= 0; i--, j++)
      temp += s[i];

    return temp;
  }

  static void Main() {
    // Сконструировать делегат.
    StrMod strOp = new StrMod(ReplaceSpaces);
    string str;

    // Вызвать методы с помощью делегата.
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = new StrMod(RemoveSpaces);
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = new StrMod(Reverse);
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Замена пробелов дефисами.
Результирующая строка: Это-простой-тест.

Удаление пробелов.
Результирующая строка: Этопростойтест.

Обращение строки.
Результирующая строка: .тсет йотсорп отЭ
```

Рассмотрим данный пример более подробно. В его коде сначала объявляется делегат `StrMod` типа `string`, как показано ниже.

```
delegate string StrMod(string str);
```

Как видите, делегат `StrMod` принимает один параметр типа `string` и возвращает одно значение того же типа.

Далее в классе DelegateTest объявляются три статических метода с одним параметром типа string и возвращаемым значением того же типа. Следовательно, они соответствуют делегату StrMod. Эти методы видоизменяют строку в той или иной форме. Обратите внимание на то, что в методе ReplaceSpaces() для замены пробелов дефисами используется один из методов типа string — Replace().

В методе Main() создается переменная экземпляра strOp ссылочного типа StrMod и затем ей присваивается ссылка на метод ReplaceSpaces(). Обратите особое внимание на следующую строку кода.

```
StrMod strOp = new StrMod(ReplaceSpaces);
```

В этой строке метод ReplaceSpaces() передается в качестве параметра. При этом указывается только его имя, но не параметры. Данный пример можно обобщить: при получении экземпляра делегата достаточно указать только имя метода, на который должен ссылаться делегат. Ясно, что сигнатура метода должна совпадать с той, что указана в объявлении делегата. В противном случае во время компиляции возникнет ошибка.

Далее метод ReplaceSpaces() вызывается с помощью экземпляра делегата strOp, как показано ниже.

```
str = strOp("Это простой тест.");
```

Экземпляр делегата strOp ссылается на метод ReplaceSpaces(), и поэтому вызывается именно этот метод.

Затем экземпляру делегата strOp присваивается ссылка на метод RemoveSpaces(), и с его помощью вновь вызывается указанный метод — на этот раз RemoveSpaces().

И наконец, экземпляру делегата strOp присваивается ссылка на метод Reverse(). А в итоге вызывается именно этот метод.

Главный вывод из данного примера заключается в следующем: в тот момент, когда происходит обращение к экземпляру делегата strOp, вызывается метод, на который он ссылается. Следовательно, вызов метода разрешается во время выполнения, а не в процессе компиляции.

Групповое преобразование делегируемых методов

Еще в версии C# 2.0 было внедрено специальное средство, существенно упрощающее синтаксис присваивания метода делегату. Это так называемое *групповое преобразование методов*, позволяющее присвоить имя метода делегату, не прибегая к оператору new или явному вызову конструктора делегата.

Ниже приведен метод Main() из предыдущего примера, измененный с целью продемонстрировать групповое преобразование методов.

```
static void Main() {
  // Сконструировать делегат, используя групповое преобразование методов.
  StrMod strOp = ReplaceSpaces; // использовать групповое преобразование методов
  string str;

  // Вызвать методы с помощью делегата.
  str = strOp("Это простой тест.");
  Console.WriteLine("Результирующая строка: " + str);
  Console.WriteLine();
```

```
  strOp = RemoveSpaces; // использовать групповое преобразование методов
  str = strOp("Это простой тест.");
  Console.WriteLine("Результирующая строка: " + str);
  Console.WriteLine();

  strOp = Reverse; // использовать групповое преобразование методов
str = strOp("Это простой тест.");
  Console.WriteLine("Результирующая строка: " + str);
  Console.WriteLine();
}
```

Обратите особое внимание на то, как создается экземпляр делегата `strOp` и как ему присваивается метод `ReplaceSpaces` в следующей строке кода.

```
strOp = RemoveSpaces; // использовать групповое преобразование методов
```

В этой строке кода имя метода присваивается непосредственно экземпляру делегата `strOp`, а все заботы по автоматическому преобразованию метода в тип делегата "возлагаются" на средства C#. Этот синтаксис может быть распространен на любую ситуацию, в которой метод присваивается или преобразуется в тип делегата.

Синтаксис группового преобразования методов существенно упрощен по сравнению с прежним подходом к делегированию, поэтому в остальной части книги используется именно он.

Применение методов экземпляра в качестве делегатов

В предыдущем примере использовались статические методы, но делегат может ссылаться и на методы экземпляра, хотя для этого требуется ссылка на объект. Так, ниже приведен измененный вариант предыдущего примера, в котором операции со строками инкапсулируются в классе `StringOps`. Следует заметить, что в данном случае может быть также использован синтаксис группового преобразования методов.

```
// Делегаты могут ссылаться и на методы экземпляра.

using System;

// Объявить тип делегата.
delegate string StrMod(string str);

class StringOps {
  // Заменить пробелы дефисами.
  public string ReplaceSpaces(string s) {
    Console.WriteLine("Замена пробелов дефисами.");
    return s.Replace(' ', '-');
  }

  // Удалить пробелы.
  public string RemoveSpaces(string s) {
    string temp = "";
    int i;
    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < s.Length; i++)
      if(s[i] != ' ') temp += s[i];
```

```
    return temp;
  }

  // Обратить строку.
  public string Reverse(string s) {
    string temp = "";
    int i, j;
    Console.WriteLine("Обращение строки.");
    for(j=0, i=s.Length-1; i >= 0; i--, j++)
      temp += s[i];

    return temp;
  }
}

class DelegateTest {
  static void Main() {
    StringOps so = new StringOps(); // создать экземпляр
                                    // объекта класса StringOps
    // Инициализировать делегат.
    StrMod strOp = so.ReplaceSpaces;
    string str;

    // Вызвать методы с помощью делегатов.
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = so.RemoveSpaces;
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = so.Reverse;
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
  }
}
```

Результат выполнения этого кода получается таким же, как и в предыдущем примере, но на этот раз делегат обращается к методам по ссылке на экземпляр объекта класса `StringOps`.

Групповая адресация

Одним из самых примечательных свойств делегата является поддержка групповой адресации. Попросту говоря, *групповая адресация* — это возможность создать *список*, или *цепочку вызовов*, для методов, которые вызываются автоматически при обращении к делегату. Создать такую цепочку нетрудно. Для этого достаточно получить экземпляр делегата, а затем добавить методы в цепочку с помощью оператора + или +=. Для удаления метода из цепочки служит оператор - или -=. Если делегат возвращает значение, то им становится значение, возвращаемое последним методом в списке вызовов. Поэтому делегат, в котором используется групповая адресация, обычно имеет возвращаемый тип `void`.

Ниже приведен пример групповой адресации. Это переработанный вариант предыдущих примеров, в котором тип значений, возвращаемых методами манипулирования строками, изменен на void, а для возврата измененной строки в вызывающую часть кода служит параметр типа ref. Благодаря этому методы оказываются более приспособленными для групповой адресации.

```
// Продемонстрировать групповую адресацию.

using System;

// Объявить тип делегата.
delegate void StrMod(ref string str);

class MultiCastDemo {
  // Заменить пробелы дефисами.
  static void ReplaceSpaces(ref string s) {
    Console.WriteLine("Замена пробелов дефисами.");
    s = s.Replace(' ', '-');
    }

  // Удалить пробелы.
  static void RemoveSpaces(ref string s) {
    string temp = "";
    int i;

    Console.WriteLine("Удаление пробелов.");
    for(i=0; i < s.Length; i++)
      if(s[i] != ' ') temp += s[i];

    s = temp;
  }

  // Обратить строку.
  static void Reverse(ref string s) {
    string temp = "";
    int i, j;

    Console.WriteLine("Обращение строки.");
    for(j=0, i=s.Length-1; i >= 0; i--, j++)
      temp += s[i];

    s = temp;
  }

  static void Main() {
    // Сконструировать делегаты.
    StrMod strOp;
    StrMod replaceSp = ReplaceSpaces;
    StrMod removeSp = RemoveSpaces;
    StrMod reverseStr = Reverse;
    string str = "Это простой тест.";

    // Организовать групповую адресацию.
    strOp = replaceSp;
```

```
    strOp += reverseStr;

    // Обратиться к делегату с групповой адресацией.
    strOp(ref str);
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    // Удалить метод замены пробелов и добавить метод удаления пробелов.
    strOp -= replaceSp;
    strOp += removeSp;
    str = "Это простой тест."; // восстановить исходную строку

    // Обратиться к делегату с групповой адресацией.
    strOp(ref str);
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();
  }
}
```

Выполнение этого кода приводит к следующему результату.

```
Замена пробелов дефисами.
Обращение строки.
Результирующая строка: .тсет-йотсорп-отЭ

Обращение строки.
Удаление пробелов.
Результирующая строка: .тсетйотсорпотЭ
```

В методе `Main()` из рассматриваемого здесь примера кода создаются четыре экземпляра делегата. Первый из них, `strOp`, является пустым, а три остальных ссылаются на конкретные методы видоизменения строки. Затем организуется групповая адресация для вызова методов `RemoveSpaces()` и `Reverse()`. Это делается в приведенных ниже строках кода.

```
strOp = replaceSp;
strOp += reverseStr
```

Сначала делегату `strOp` присваивается ссылка `replaceSp`, а затем с помощью оператора += добавляется ссылка `reverseStr`. При обращении к делегату `strOp` вызываются оба метода, заменяя пробелы дефисами и обращая строку, как и показывает приведенный выше результат.

Далее ссылка `replaceSp` удаляется из цепочки вызовов в следующей строке кода:

```
strOp -= replaceSp;
```

и добавляется ссылка `removeSp` в строке кода.

```
strOp += removeSp;
```

После этого вновь происходит обращение к делегату `strOp`. На этот раз обращается строка с удаленными пробелами.

Цепочки вызовов являются весьма эффективным механизмом, поскольку они позволяют определить ряд методов, выполняемых единым блоком. Благодаря этому улучшается структура некоторых видов кода. Кроме того, цепочки вызовов имеют особое значение для обработки событий, как станет ясно в дальнейшем.

Ковариантность и контравариантность

Делегаты становятся еще более гибкими средствами программирования благодаря двум свойствам: *ковариантности* и *контравариантности*. Как правило, метод, передаваемый делегату, должен иметь такой же возвращаемый тип и сигнатуру, как и делегат. Но в отношении производных типов это правило оказывается не таким строгим благодаря ковариантности и контравариантности. В частности, ковариантность позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата. А контравариантность позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

Ниже приведен пример, демонстрирующий ковариантность и контравариантность.

```
// Продемонстрировать ковариантность и контравариантность.

using System;

class X {
  public int Val;
}

// Класс Y, производный от класса X.
class Y : X { }

// Этот делегат возвращает объект класса X и
// принимает объект класса Y в качестве аргумента.
delegate X ChangeIt(Y obj);

class CoContraVariance {

  // Этот метод возвращает объект класса X и
  // имеет объект класса X в качестве параметра.
  static X IncrA(X obj) {
    X temp = new X();
    temp.Val = obj.Val + 1;
    return temp;
  }

  // Этот метод возвращает объект класса Y и
  // имеет объект класса Y в качестве параметра.
  static Y IncrB(Y obj) {
    Y temp = new Y();
    temp.Val = obj.Val + 1;
    return temp;
  }

  static void Main() {
    Y Yob = new Y();
    // В данном случае параметром метода IncrA является объект класса X,
    // а параметром делегата ChangeIt — объект класса Y. Но благодаря
    // контравариантности следующая строка кода вполне допустима.
    ChangeIt change = IncrA;
```

```
    X Xob = change(Yob);

    Console.WriteLine("Xob: " + Xob.Val);

    // В этом случае возвращаемым типом метода IncrB служит объект класса Y,
    // а возвращаемым типом делегата ChangeIt — объект класса X. Но благодаря
    // ковариантности следующая строка кода оказывается вполне допустимой.
    change = IncrB;

    Yob = (Y) change(Yob);

    Console.WriteLine("Yob: " + Yob.Val);
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Xob: 1
Yob: 1
```

В данном примере класс `Y` является производным от класса `X`. А делегат `ChangeIt` объявляется следующим образом.

```
delegate X ChangeIt(Y obj);
```

Делегат возвращает объект класса `X` и принимает в качестве параметра объект класса `Y`. А методы `IncrA()` и `IncrB()` объявляются следующим образом.

```
static X IncrA(X obj)
static Y IncrB(Y obj)
```

Метод `IncrA()` принимает объект класса `X` в качестве параметра и возвращает объект того же класса. А метод `IncrB()` принимает в качестве параметра объект класса `Y` и возвращает объект того же класса. Но благодаря ковариантности и контравариантности любой из этих методов может быть передан делегату `ChangeIt`, что и демонстрирует рассматриваемый здесь пример.

Таким образом, в строке

```
ChangeIt change = IncrA;
```

метод `IncrA()` может быть передан делегату благодаря контравариантности, так как объект класса `X` служит в качестве параметра метода `IncrA()`, а объект класса `Y` — в качестве параметра делегата `ChangeIt`. Но метод и делегат оказываются совместимыми в силу контравариантности, поскольку типом параметра метода, передаваемого делегату, служит класс, являющийся базовым для класса, указываемого в качестве типа параметра делегата.

Приведенная ниже строка кода также является вполне допустимой, но на этот раз благодаря ковариантности.

```
change = IncrB;
```

В данном случае возвращаемым типом для метода `IncrB()` служит класс `Y`, а для делегата — класс `X`. Но поскольку возвращаемый тип метода является производным классом от возвращаемого типа делегата, то оба оказываются совместимыми в силу ковариантности.

Класс System.Delegate

Все делегаты и классы оказываются производными неявным образом от класса `System.Delegate`. Как правило, членами этого класса не пользуются непосредственно, и это не делается явным образом в данной книге. Но члены класса `System.Delegate` могут оказаться полезными в ряде особых случаев.

Назначение делегатов

В предыдущих примерах был наглядно продемонстрирован внутренний механизм действия делегатов, но эти примеры не показывают их истинное назначение. Как правило, делегаты применяются по двум причинам. Во-первых, как упоминалось ранее в этой главе, делегаты поддерживают события. И во-вторых, делегаты позволяют вызывать методы во время выполнения программы, не зная о них ничего определенного в ходе компиляции. Это очень удобно для создания базовой конструкции, допускающей подключение отдельных программных компонентов. Рассмотрим в качестве примера графическую программу, аналогичную стандартной сервисной программе Windows Paint. С помощью делегата можно предоставить пользователю возможность подключать специальные цветные фильтры или анализаторы изображений. Кроме того, пользователь может составлять из этих фильтров или анализаторов целые последовательности. Подобные возможности программы нетрудно обеспечить, используя делегаты.

Анонимные функции

Метод, на который ссылается делегат, нередко используется только для этой цели. Иными словами, единственным основанием для существования метода служит то обстоятельство, что он может быть вызван посредством делегата, но сам он не вызывается вообще. В подобных случаях можно воспользоваться *анонимной функцией*, чтобы не создавать отдельный метод. Анонимная функция, по существу, представляет собой безымянный кодовый блок, передаваемый конструктору делегата. Преимущество анонимной функции состоит, в частности, в ее простоте. Благодаря ей отпадает необходимость объявлять отдельный метод, единственное назначение которого состоит в том, что он передается делегату.

Начиная с версии 3.0, в C# предусмотрены две разновидности анонимных функций: *анонимные методы* и *лямбда-выражения*. Анонимные методы были внедрены в C# еще в версии 2.0, а лямбда-выражения — в версии 3.0. В целом лямбда-выражение совершенствует принцип действия анонимного метода и в настоящее время считается более предпочтительным для создания анонимной функции. Но анонимные методы широко применяются в существующем коде C#.и поэтому по-прежнему являются важной составной частью C#. А поскольку анонимные методы предшествовали появлению лямбда-выражений, то ясное представление о них позволяет лучше понять особенности лямбда-выражений. К тому же анонимные методы могут быть использованы в целом ряде случаев, где применение лямбда-выражений оказывается невозможным. Именно поэтому в этой главе рассматриваются и анонимные методы, и лямбда-выражения.

Анонимные методы

Анонимный метод — один из способов создания безымянного блока кода, связанного с конкретным экземпляром делегата. Для создания анонимного метода достаточно указать кодовый блок после ключевого слова `delegate`. Покажем, как это делается, на конкретном примере. В приведенной ниже программе анонимный метод служит для подсчета от 0 до 5.

```
// Продемонстрировать применение анонимного метода.

using System;

// Объявить тип делегата.
delegate void CountIt();

class AnonMethDemo {
  static void Main() {

    // Далее следует код для подсчета чисел, передаваемый делегату
    // в качестве анонимного метода.
    CountIt count = delegate {
      // Этот кодовый блок передается делегату.
      for(int i=0; i <= 5; i++)
        Console.WriteLine(i);
    }; // обратите внимание на точку с запятой

    count();
  }
}
```

В данной программе сначала объявляется тип делегата `CountIt` без параметров и с возвращаемым типом `void`. Далее в методе `Main()` создается экземпляр `count` делегата `CountIt`, которому передается кодовый блок, следующий после ключевого слова `delegate`. Именно этот кодовый блок и является анонимным методом, который будет выполняться при обращении к делегату `count`. Обратите внимание на то, что после кодового блока следует точка с запятой, фактически завершающая оператор объявления. Ниже приведен результат выполнения данной программы.

```
0
1
2
3
4
5
```

Передача аргументов анонимному методу

Анонимному методу можно передать один или несколько аргументов. Для этого достаточно указать в скобках список параметров после ключевого слова `delegate`, а при обращении к экземпляру делегата — передать ему соответствующие аргументы. В качестве примера ниже приведен вариант предыдущей программы, измененный с целью передать в качестве аргумента конечное значение для подсчета.

```
// Продемонстрировать применение анонимного метода, принимающего аргумент.

using System;

// Обратите внимание на то, что теперь у делегата CountIt имеется параметр.
delegate void CountIt(int end);

class AnonMethDemo2 {

  static void Main() {

    // Здесь конечное значение для подсчета передается анонимному методу.
    CountIt count = delegate (int end) {
      for(int i=0; i <= end; i++)
        Console.WriteLine(i);
    };

    count(3);
    Console.WriteLine();
    count(5);
  }
}
```

В этом варианте программы делегат `CountIt` принимает целочисленный аргумент. Обратите внимание на то, что при создании анонимного метода список параметров указывается после ключевого слова `delegate`. Параметр `end` становится доступным для кода в анонимном методе таким же образом, как и при создании именованного метода. Ниже приведен результат выполнения данной программы.

```
0
1
2
3

0
1
2
3
4
5
```

Возврат значения из анонимного метода

Анонимный метод может возвращать значение. Для этой цели служит оператор `return`, действующий в анонимном методе таким же образом, как и в именованном методе. Как и следовало ожидать, тип возвращаемого значения должен быть совместим с возвращаемым типом, указываемым в объявлении делегата. В качестве примера ниже приведен код, выполняющий подсчет с суммированием и возвращающий результат.

```
// Продемонстрировать применение анонимного метода, возвращающего значение.

using System;
```

```
// Этот делегат возвращает значение.
delegate int CountIt(int end);

class AnonMethDemo3 {

  static void Main() {
    int result;

    // Здесь конечное значение для подсчета передается анонимному методу.
    // А возвращается сумма подсчитанных чисел.
    CountIt count = delegate (int end) {
      int sum = 0;

      for(int i=0; i <= end; i++) {
        Console.WriteLine(i);
        sum += i;
      }
      return sum; // возвратить значение из анонимного метода
    };

    result = count(3);
    Console.WriteLine("Сумма 3 равна " + result);
    Console.WriteLine();

    result = count(5);
    Console.WriteLine("Сумма 5 равна " + result);
  }
}
```

В этом варианте кода суммарное значение возвращается кодовым блоком, связанным с экземпляром делегата `count`. Обратите внимание на то, что оператор `return` применяется в анонимном методе таким же образом, как и в именованном методе. Ниже приведен результат выполнения данного кода.

```
0
1
2
3
Сумма 3 равна 6

0
1
2
3
4
5
Сумма 5 равна 15
```

Применение внешних переменных в анонимных методах

Локальная переменная, в область действия которой входит анонимный метод, называется *внешней переменной*. Такие переменные доступны для использования в анонимном методе. И в этом случае внешняя переменная считается *захваченной*. Захваченная переменная существует до тех пор, пока захвативший ее делегат не будет собран

в "мусор". Поэтому если локальная переменная, которая обычно прекращает свое существование после выхода из кодового блока, используется в анонимном методе, то она продолжает существовать до тех пор, пока не будет уничтожен делегат, ссылающийся на этот метод.

Захват локальной переменной может привести к неожиданным результатам. В качестве примера рассмотрим еще один вариант программы подсчета с суммированием чисел. В данном варианте объект `CountIt` конструируется и возвращается статическим методом `Counter()`. Этот объект использует переменную `sum`, объявленную в охватывающей области действия метода `Counter()`, а не самого анонимного метода. Поэтому переменная `sum` захватывается анонимным методом. Метод `Counter()` вызывается в методе `Main()` для получения объекта `CountIt`, а следовательно, переменная `sum` не уничтожается до самого конца программы.

```
// Продемонстрировать применение захваченной переменной.

using System;

// Этот делегат возвращает значение типа int и принимает аргумент типа int.
delegate int CountIt(int end);

class VarCapture {

  static CountIt Counter() {
    int sum = 0;

    // Здесь подсчитанная сумма сохраняется в переменной sum.
    CountIt ctObj = delegate (int end) {
      for(int i=0; i <= end; i++) {
        Console.WriteLine(i);
        sum += i;
      }
      return sum;
    };
    return ctObj;
  }

  static void Main() {
    // Получить результат подсчета.
    CountIt count = Counter();

    int result;

    result = count(3);
    Console.WriteLine("Сумма 3 равна " + result);
    Console.WriteLine();

    result = count(5);
    Console.WriteLine("Сумма 5 равна " + result);
  }
}
```

Ниже приведен результат выполнения этой программы. Обратите особое внимание на суммарное значение.

```
0
1
2
3
Сумма 3 равна 6

0
1
2
3
4
5
Сумма 5 равна 21
```

Как видите, подсчет по-прежнему выполняется как обычно. Но обратите внимание на то, что сумма 5 теперь равна 21, а не 15! Дело в том, что переменная `sum` захватывается объектом `ctObj` при его создании в методе `Counter()`. Это означает, что она продолжает существовать вплоть до уничтожения делегата `count` при "сборке мусора" в самом конце программы. Следовательно, ее значение не уничтожается после возврата из метода `Counter()` или при каждом вызове анонимного метода, когда происходит обращение к делегату `count` в методе `Main()`.

Несмотря на то что применение захваченных переменных может привести к довольно неожиданным результатам, как в приведенном выше примере, оно все же логически обоснованно. Ведь когда анонимный метод захватывает переменную, она продолжает существовать до тех пор, пока используется захватывающий ее делегат. В противном случае захваченная переменная оказалась бы неопределенной, когда она могла бы потребоваться делегату.

Лямбда-выражения

Несмотря на всю ценность анонимных методов, им на смену пришел более совершенный подход: *лямбда-выражение*. Не будет преувеличением сказать, что лямбда-выражение относится к одним из самых важных нововведений в C#, начиная с выпуска исходной версии 1.0 этого языка программирования. Лямбда-выражение основывается на совершенно новом синтаксическом элементе и служит более эффективной альтернативой анонимному методу. И хотя лямбда-выражения находят применение главным образом в работе с LINQ (подробнее об этом — в главе 19), они часто используются и вместе с делегатами и событиями. Именно об этом применении лямбда-выражений и пойдет речь в данном разделе.

Лямбда-выражение — это другой собой создания анонимной функции. (Первый ее способ, анонимный метод, был рассмотрен в предыдущем разделе.) Следовательно, лямбда-выражение может быть присвоено делегату. А поскольку лямбда-выражение считается более эффективным, чем эквивалентный ему анонимный метод, то в большинстве случаев рекомендуется отдавать предпочтение именно ему.

Лямбда-оператор

Во всех лямбда-выражениях применяется новый лямбда-оператор =>, который разделяет лямбда-выражение на две части. В левой его части указывается входной параметр (или несколько параметров), а в правой части — тело лямбда-выражения. Оператор => иногда описывается такими словами, как "переходит" или "становится".

В C# поддерживаются две разновидности лямбда-выражений в зависимости от тела самого лямбда-выражения. Так, если тело лямбда-выражения состоит из одного выражения, то образуется *одиночное лямбда-выражение*. В этом случае тело выражения не заключается в фигурные скобки. Если же тело лямбда-выражения состоит из блока операторов, заключенных в фигурные скобки, то образуется *блочное лямбда-выражение*. При этом блочное лямбда-выражение может содержать целый ряд операторов, в том числе циклы, вызовы методов и условные операторы `if`. Обе разновидности лямбда-выражений рассматриваются далее по отдельности.

Одиночные лямбда-выражения

В одиночном лямбда-выражении часть, находящаяся справа от оператора =>, воздействует на параметр (или ряд параметров), указываемый слева. Возвращаемым результатом вычисления такого выражения является результат выполнения лямбда-оператора.

Ниже приведена общая форма одиночного лямбда-выражения, принимающего единственный параметр.

```
параметр => выражение
```

Если же требуется указать несколько параметров, то используется следующая форма.

```
(список_параметров) => выражение
```

Таким образом, когда требуется указать два параметра или более, их следует заключить в скобки. Если же *выражение* не требует параметров, то следует использовать пустые скобки.

Ниже приведен простой пример одиночного лямбда-выражения.

```
count => count + 2
```

В этом выражении `count` служит параметром, на который воздействует выражение `count + 2`. В итоге значение параметра `count` увеличивается на 2. А вот еще один пример одиночного лямбда-выражения.

```
n => n % 2 == 0
```

В данном случае выражение возвращает логическое значение `true`, если числовое значение параметра `n` оказывается четным, а иначе — логическое значение `false`.

Лямбда-выражение применяется в два этапа. Сначала объявляется тип делегата, совместимый с лямбда-выражением, а затем экземпляр делегата, которому присваивается лямбда-выражение. После этого лямбда-выражение вычисляется при обращении к экземпляру делегата. Результатом его вычисления становится возвращаемое значение.

В приведенном ниже примере программы демонстрируется применение двух одиночных лямбда-выражений. Сначала в этой программе объявляются два типа делегатов. Первый из них, `Incr`, принимает аргумент типа `int` и возвращает результат того же типа. Второй делегат, `IsEven`, также принимает аргумент типа `int`, но возвращает результат типа `bool`. Затем экземплярам этих делегатов присваиваются одиночные лямбда-выражения. И наконец, лямбда-выражения вычисляются с помощью соответствующих экземпляров делегатов.

```
// Применить два одиночных лямбда-выражения.

using System;
```

```
// Объявить делегат, принимающий аргумент типа int и
// возвращающий результат типа int.
delegate int Incr(int v);

// Объявить делегат, принимающий аргумент типа int и
// возвращающий результат типа bool.
delegate bool IsEven(int v);

class SimpleLambdaDemo {

  static void Main() {

    // Создать делегат Incr, ссылающийся на лямбда-выражение,
    // увеличивающее свой параметр на 2.
    Incr incr = count => count + 2;

    // А теперь использовать лямбда-выражение incr.
    Console.WriteLine("Использование лямбда-выражения incr: ");
    int x = -10;
    while(x <= 0) {
      Console.Write(x + " ");
      x = incr(x); // увеличить значение x на 2
    }

    Console.WriteLine("\n");

    // Создать экземпляр делегата IsEven, ссылающийся на лямбда-выражение,
    // возвращающее логическое значение true, если его параметр имеет четное
    // значение, а иначе — логическое значение false.
    IsEven isEven = n => n % 2 == 0;

    // А теперь использовать лямбда-выражение isEven.
     Console.WriteLine("Использование лямбда-выражения isEven: ");
     for(int i=1; i <= 10; i++)
       if(isEven(i)) Console.WriteLine(i + " четное.");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Использование лямбда-выражения incr:
-10 -8 -6 -4 -2 0

Использование лямбда-выражения isEven:
2 четное.
4 четное.
6 четное.
8 четное.
10 четное.
```

Обратите в данной программе особое внимание на следующие строки объявлений.

```
Incr incr = count => count + 2;
IsEven isEven = n => n % 2 == 0;
```

В первой строке объявления экземпляру делегата `incr` присваивается одиночное лямбда-выражение, возвращающее результат увеличения на 2 значения параметра `count`. Это выражение может быть присвоено делегату `Incr`, поскольку оно совместимо с объявлением данного делегата. Аргумент, указываемый при обращении к экземпляру делегата `incr`, передается параметру `count`, который и возвращает результат вычисления лямбда-выражения. Во второй строке объявления делегату `isEven` присваивается выражение, возвращающее логическое значение `true`, если передаваемый ему аргумент оказывается четным, а иначе — логическое значение `false`. Следовательно, это лямбда-выражение совместимо с объявлением делегата `IsEven`.

В связи со всем изложенным выше возникает резонный вопрос: каким образом компилятору становится известно о типе данных, используемых в лямбда-выражении, например, о типе `int` параметра `count` в лямбда-выражении, присваиваемом экземпляру делегата `incr`? Ответить на этот вопрос можно так: компилятор делает заключение о типе параметра и типе результата вычисления выражения по типу делегата. Следовательно, параметры и возвращаемое значение лямбда-выражения должны быть совместимы по типу с параметрами и возвращаемым значением делегата.

Несмотря на всю полезность логического заключения о типе данных, в некоторых случаях приходится явно указывать тип параметра лямбда-выражения. Для этого достаточно ввести конкретное название типа данных. В качестве примера ниже приведен другой способ объявления экземпляра делегата `incr`.

```
Incr incr = (int count) => count + 2;
```

Как видите, `count` теперь явно объявлен как параметр типа `int`. Обратите также внимание на использование скобок. Теперь они необходимы. (Скобки могут быть опущены только в том случае, если задается лишь один параметр, а его тип явно не указывается.)

В предыдущем примере в обоих лямбда-выражениях использовался единственный параметр, но в целом у лямбда-выражений может быть любое количество параметров, в том числе и нулевое. Если в лямбда-выражении используется несколько параметров, их *необходимо* заключить в скобки. Ниже приведен пример использования лямбда-выражения с целью определить, находится ли значение в заданных пределах.

```
(low, high, val) => val >= low && val <= high;
```

А вот как объявляется тип делегата, совместимого с этим лямбда-выражением.

```
delegate bool InRange(int lower, int upper, int v);
```

Следовательно, экземпляр делегата `InRange` может быть создан следующим образом.

```
InRange rangeOK = (low, high, val) => val >= low && val <= high;
```

После этого одиночное лямбда-выражение может быть выполнено так, как показано ниже.

```
if(rangeOK(1, 5, 3)) Console.WriteLine(
                              "Число 3 находится в пределах от 1 до 5.");
```

И последнее замечание: внешние переменные могут использоваться и захватываться в лямбда-выражениях таким же образом, как и в анонимных методах.

Блочные лямбда-выражения

Как упоминалось выше, существуют две разновидности лямбда-выражений. Первая из них, одиночное лямбда-выражение, была рассмотрена в предыдущем разделе. Тело такого лямбда-выражения состоит только из одного выражения. Второй разновидностью является *блочное лямбда-выражение*. Для такого лямбда-выражения характерны расширенные возможности выполнения различных операций, поскольку в его теле допускается указывать несколько операторов. Например, в блочном лямбда-выражении можно использовать циклы и условные операторы `if`, объявлять переменные и т.д. Создать блочное лямбда-выражение нетрудно. Для этого достаточно заключить тело выражения в фигурные скобки. Помимо возможности использовать несколько операторов, в остальном блочное лямбда-выражение практически ничем не отличается от только что рассмотренного одиночного лямбда-выражения.

Ниже приведен пример использования блочного лямбда-выражения для вычисления и возврата факториала целого значения.

```
// Продемонстрировать применение блочного лямбда-выражения.

using System;

// Делегат IntOp принимает один аргумент типа int
// и возвращает результат типа int.
delegate int IntOp(int end);

class StatementLambdaDemo {

  static void Main() {

    // Блочное лямбда-выражение возвращает факториал
    // передаваемого ему значения.
    IntOp fact = n => {
                        int r = 1;
                        for(int i=1; i <= n; i++)
                          r = i * r;
                        return r;
                      };

    Console.WriteLine("Факториал 3 равен " + fact(3));
    Console.WriteLine("Факториал 5 равен " + fact(5));
  }
}
```

При выполнении этого кода получается следующий результат.

```
Факториал 3 равен 6
Факториал 5 равен 120
```

В приведенном выше примере обратите внимание на то, что в теле блочного лямбда-выражения объявляется переменная `r`, организуется цикл `for` и используется оператор `return`. Все эти элементы вполне допустимы в блочном лямбда-выражении. И в этом отношении оно очень похоже на анонимный метод. Следовательно, многие анонимные методы могут быть преобразованы в блочные лямбда-выражения при обновлении унаследованного кода. И еще одно замечание: когда в блочном лямбда-выражении встречается оператор `return`, он просто обусловливает возврат из лямбда-выражения, но не возврат из охватывающего метода.

И в заключение рассмотрим еще один пример, демонстрирующий блочное лямбда-выражение в действии. Ниже приведен вариант первого примера из этой главы, измененного с целью использовать блочные лямбда-выражения вместо автономных методов для выполнения различных операций со строками.

```
// Первый пример применения делегатов, переделанный с
// целью использовать блочные лямбда-выражения.

using System;

// Объявить тип делегата.
delegate string StrMod(string s);

class UseStatementLambdas {

  static void Main() {

    // Создать делегаты, ссылающиеся на лямбда- выражения,
    // выполняющие различные операции с символьными строками.

    // Заменить пробелы дефисами.
    StrMod ReplaceSpaces = s => {
         Console.WriteLine("Замена пробелов дефисами.");
         return s.Replace(' ', '-');
       };

    // Удалить пробелы.
    StrMod RemoveSpaces = s => {
               string temp = "";
               int i;

               Console.WriteLine("Удаление пробелов.");
               for(i=0; i < s.Length; i++)
                 if(s[i] != ' ') temp += s[i];

                return temp;
              };

    // Обратить строку.
    StrMod Reverse = s => {
               string temp = "";
               int i, j;

               Console.WriteLine("Обращение строки.");
               for(j=0, i=s.Length-1; i >= 0; i--, j++)
                 temp += s[i];

               return temp;
             };

    string str;

    // Обратиться к лямбда-выражениям с помощью делегатов.
    StrMod strOp = ReplaceSpaces;
```

```
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = RemoveSpaces;
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
    Console.WriteLine();

    strOp = Reverse;
    str = strOp("Это простой тест.");
    Console.WriteLine("Результирующая строка: " + str);
  }
}
```

Результат выполнения кода этого примера оказывается таким же, как и в первом примере применения делегатов.

```
Замена пробелов дефисами.
Результирующая строка: Это-простой-тест.

Удаление пробелов.
Результирующая строка: Этопростойтест.

Обращение строки.
Результирующая строка: .тсет йотсорп отЭ
```

События

Еще одним важным средством C#, основывающимся на делегатах, является *событие*. Событие, по существу, представляет собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу: объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события. Обработчики событий обычно представлены делегатами.

События являются членами класса и объявляются с помощью ключевого слова `event`. Чаще всего для этой цели используется следующая форма:

```
event делегат_события имя_события;
```

где *делегат_события* обозначает имя делегата, используемого для поддержки события, а *имя_события* — конкретный объект объявляемого события.

Рассмотрим для начала очень простой пример.

```
// Очень простой пример, демонстрирующий событие.

using System;

// Объявить тип делегата для события.
delegate void MyEventHandler();

// Объявить класс, содержащий событие.
class MyEvent {
  public event MyEventHandler SomeEvent;
```

```
  // Этот метод вызывается для запуска события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class EventDemo {
  // Обработчик события.
  static void Handler() {
    Console.WriteLine("Произошло событие");
  }

  static void Main() {
    MyEvent evt = new MyEvent();

    // Добавить метод Handler() в список событий.
    evt.SomeEvent += Handler;

   // Запустить событие.
   evt.OnSomeEvent();
  }
}
```

Вот какой результат получается при выполнении этого кода.

```
Произошло событие
```

Несмотря на всю свою простоту, данный пример кода содержит все основные элементы, необходимые для обработки событий. Он начинается с объявления типа делегата для обработчика событий, как показано ниже.

```
delegate void MyEventHandler();
```

Все события активизируются с помощью делегатов. Поэтому тип делегата события определяет возвращаемый тип и сигнатуру для события. В данном случае параметры события отсутствуют, но их разрешается указывать.

Далее создается класс события `MyEvent`. В этом классе объявляется событие `SomeEvent` в следующей строке кода.

```
public event MyEventHandler SomeEvent;
```

Обратите внимание на синтаксис этого объявления. Ключевое слово `event` уведомляет компилятор о том, что объявляется событие.

Кроме того, в классе `MyEvent` объявляется метод `OnSomeEvent()`, вызываемый для сигнализации о запуске события. Это означает, что он вызывается, когда происходит событие. В методе `OnSomeEvent()` вызывается обработчик событий с помощью делегата `SomeEvent`.

```
if(SomeEvent != null)
  SomeEvent();
```

Как видите, обработчик вызывается лишь в том случае, если событие `SomeEvent` не является пустым. А поскольку интерес к событию должен быть зарегистрирован в других частях программы, чтобы получать уведомления о нем, то метод `OnSomeEvent()` может быть вызван до регистрации любого обработчика события. Но во избежание

вызова по пустой ссылке делегат события должен быть проверен, чтобы убедиться в том, что он не является пустым.

В классе `EventDemo` создается обработчик событий `Handler()`. В данном простом примере обработчик событий просто выводит сообщение, но другие обработчики могут выполнять более содержательные функции. Далее в методе `Main()` создается объект класса события `MyEvent`, а `Handler()` регистрируется как обработчик этого события, добавляемый в список.

```
MyEvent evt = new MyEvent();

// Добавить метод Handler() в список событий.
evt.SomeEvent += Handler;
```

Обратите внимание на то, что обработчик добавляется в список с помощью оператора +=. События поддерживают только операторы += и -=. В данном случае метод `Handler()` является статическим, но в качестве обработчиков событий могут также служить методы экземпляра.

И наконец, событие запускается, как показано ниже.

```
// Запустить событие.
evt.OnSomeEvent();
```

Вызов метода `OnSomeEvent()` приводит к вызову всех событий, зарегистрированных обработчиком. В данном случае зарегистрирован только один такой обработчик, но их может быть больше, как поясняется в следующем разделе.

Пример групповой адресации события

Как и делегаты, события поддерживают групповую адресацию. Это дает возможность нескольким объектам реагировать на уведомление о событии. Ниже приведен пример групповой адресации события.

```
// Продемонстрировать групповую адресацию события.

using System;

// Объявить тип делегата для события.
delegate void MyEventHandler();

// Объявить делегат, содержащий событие.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для запуска события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {
  public void Xhandler() {
    Console.WriteLine("Событие получено объектом класса X");
  }
```

```
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Событие получено объектом класса Y");
  }
}

class EventDemo2 {
  static void Handler() {
    Console.WriteLine("Событие получено объектом класса EventDemo");
  }

  static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();
    Y yOb = new Y();

    // Добавить обработчики в список событий.
    evt.SomeEvent += Handler;
    evt.SomeEvent += xOb.Xhandler;
    evt.SomeEvent += yOb.Yhandler;

    // Запустить событие.
    evt.OnSomeEvent();
    Console.WriteLine();

    // Удалить обработчик.
    evt.SomeEvent -= xOb.Xhandler;
    evt.OnSomeEvent();
  }
}
```

При выполнении кода этого примера получается следующий результат.

```
Событие получено объектом класса EventDemo
Событие получено объектом класса X
Событие получено объектом класса Y

Событие получено объектом класса EventDemo
Событие получено объектом класса Y
```

В данном примере создаются два дополнительных класса, `X` и `Y`, в которых также определяются обработчики событий, совместимые с делегатом `MyEventHandler`. Поэтому эти обработчики могут быть также включены в цепочку событий. Обратите внимание на то, что обработчики в классах `X` и `Y` не являются статическими. Это означает, что сначала должны быть созданы объекты каждого из этих классов, а затем в цепочку событий должны быть введены обработчики, связанные с их экземплярами. Об отличиях между обработчиками экземпляра и статическими обработчиками речь пойдет в следующем разделе.

Методы экземпляра в сравнении со статическими методами в качестве обработчиков событий

Методы экземпляра и статические методы могут быть использованы в качестве обработчиков событий, но между ними имеется одно существенное отличие. Когда

статический метод используется в качестве обработчика, уведомление о событии распространяется на весь класс. А когда в качестве обработчика используется метод экземпляра, то события адресуются конкретным экземплярам объектов. Следовательно, каждый объект определенного класса, которому требуется получить уведомление о событии, должен быть зарегистрирован отдельно. На практике большинство обработчиков событий представляет собой методы экземпляра, хотя это, конечно, зависит от конкретного приложения. Рассмотрим применение каждой из этих двух разновидностей методов в качестве обработчиков событий на конкретных примерах.

В приведенной ниже программе создается класс X, в котором метод экземпляра определяется в качестве обработчика событий. Это означает, что каждый объект класса X должен быть зарегистрирован отдельно, чтобы получать уведомления о событиях. Для демонстрации этого факта в данной программе производится групповая адресация события трем отдельным объектам класса X.

```
/* Уведомления о событиях получают отдельные объекты, когда метод экземпляра
   используется в качестве обработчика событий. */

using System;

// Объявить тип делегата для события.
delegate void MyEventHandler();

// Объявить класс, содержащий событие.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для запуска события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {
  int id;
  public X(int x) { id = x; }

  // Этот метод экземпляра предназначен в качестве обработчика событий.
  public void Xhandler() {
    Console.WriteLine("Событие получено объектом " + id);
  }
}

class EventDemo3 {
  static void Main() {
    MyEvent evt = new MyEvent();
    X o1 = new X(1);
    X o2 = new X(2);
    X o3 = new X(3);
    evt.SomeEvent += o1.Xhandler;
    evt.SomeEvent += o2.Xhandler;
    evt.SomeEvent += o3.Xhandler;
```

```
    // Запустить событие.
    evt.OnSomeEvent();
  }
}
```

Выполнение кода из этого примера приводит к следующему результату.

```
Событие получено объектом 1
Событие получено объектом 2
Событие получено объектом 3
```

Как следует из результата выполнения кода из приведенного выше примера, каждый объект должен зарегистрировать свой интерес в событии отдельно, и тогда он будет получать отдельное уведомление о событии.

С другой стороны, когда в качестве обработчика событий используется статический метод, события обрабатываются независимо от какого-либо объекта, как демонстрируется в приведенном ниже примере программы.

```
/* Уведомления о событии получает класс, когда статический метод
   используется в качестве обработчика событий. */

using System;

// Объявить тип делегата для события.
delegate void MyEventHandler();

// Объявить класс, содержащий событие.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для запуска события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent();
  }
}

class X {

  /* Этот статический метод предназначен в качестве
    обработчика событий. */
    public static void Xhandler() {
      Console.WriteLine("Событие получено классом.");
  }
}

class EventDemo4 {
  static void Main() {
    MyEvent evt = new MyEvent();

    evt.SomeEvent += X.Xhandler;

    // Запустить событие.
    evt.OnSomeEvent();
  }
}
```

При выполнение кода этого примера получается следующий результат.

```
Событие получено классом.
```

Обратите в данном примере внимание на то, что объекты класса `X` вообще не создаются. Но поскольку `Xhandler()` является статическим методом класса `X`, то он может быть привязан к событию `SomeEvent` и выполнен при вызове метода `OnSomeEvent()`.

Применение аксессоров событий

В приведенных выше примерах события формировались в форме, допускавшей автоматическое управление списком вызовов обработчиков событий, включая добавление и удаление обработчиков событий из списка. Поэтому управление этим списком не нужно было организовывать вручную. Благодаря именно этому свойству такие события используются чаще всего. Тем не менее организовать управление списком вызовов обработчиков событий можно и вручную, чтобы, например, реализовать специальный механизм сохранения событий.

Для управления списком обработчиков событий служит расширенная форма оператора `event`, позволяющая использовать *аксессоры событий*. Эти аксессоры предоставляют средства для управления реализацией подобного списка в приведенной ниже форме.

```
event делегат_события имя_события {
  add {
    // Код добавления события в цепочку событий.
  }

  remove {
    // Код удаления события из цепочки событий.
  }
}
```

В эту форму входят два аксессора событий: `add` и `remove`. Аксессор `add` вызывается, когда обработчик событий добавляется в цепочку событий с помощью оператора +=. В то же время аксессор `remove` вызывается, когда обработчик событий удаляется из цепочки событий с помощью оператора -=.

Когда вызывается аксессор `add` или `remove`, он принимает в качестве параметра добавляемый или удаляемый обработчик. Как и в других разновидностях аксессоров, этот неявный параметр называется `value`. Реализовав аксессоры `add` или `remove`, можно организовать специальную схему хранения обработчиков событий. Например, обработчики событий можно хранить в массиве, стеке или очереди.

Ниже приведен пример программы, демонстрирующей аксессорную форму события. В ней для хранения обработчиков событий используется массив. Этот массив состоит всего из трех элементов, поэтому в цепочке событий можно хранить одновременно только три обработчика.

```
// Создать специальные средства для управления списками
// вызова обработчиков событий.

using System;

// Объявить тип делегата для события.
delegate void MyEventHandler();
```

```
// Объявить класс для хранения максимум трех событий.
class MyEvent {
  MyEventHandler[] evnt = new MyEventHandler[3];

  public event MyEventHandler SomeEvent {
    // Добавить событие в список.
    add {
      int i;
      for(i=0; i < 3; i++)
        if(evnt[i] == null) {
          evnt[i] = value;
          break;
        }
      if (i == 3) Console.WriteLine("Список событий заполнен.");
    }

    // Удалить событие из списка.
    remove {
      int i;

      for(i=0; i < 3; i++)
        if(evnt[i] == value) {
          evnt[i] = null;
          break;
        }
      if (i == 3) Console.WriteLine("Обработчик событий не найден.");
    }
  }

  // Этот метод вызывается для запуска событий.
  public void OnSomeEvent() {
    for(int i=0; i < 3; i++)
      if(evnt[i] != null) evnt[i]();
  }
}

// Создать ряд классов, использующих делегат MyEventHandler.
class W {
  public void Whandler() {
    Console.WriteLine("Событие получено объектом W");
  }
}

class X {
  public void Xhandler() {
    Console.WriteLine("Событие получено объектом X");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Событие получено объектом Y");
  }
}
```

```
class Z {
  public void Zhandler() {
    Console.WriteLine("Событие получено объектом Z");
  }
}

class EventDemo5 {
  static void Main() {
    MyEvent evt = new MyEvent();
    W wOb = new W();
    X xOb = new X();
    Y yOb = new Y();
    Z zOb = new Z();

    // Добавить обработчики в цепочку событий.
    Console.WriteLine("Добавление событий.");
    evt.SomeEvent += wOb.Whandler;
    evt.SomeEvent += xOb.Xhandler;
    evt.SomeEvent += yOb.Yhandler;

    // Сохранить нельзя - список заполнен.
    evt.SomeEvent += zOb.Zhandler;
    Console.WriteLine();

    // Запустить события.
    evt.OnSomeEvent();
    Console.WriteLine();

   // Удалить обработчик.
    Console.WriteLine("Удаление обработчика xOb.Xhandler.");
    evt.SomeEvent -= xOb.Xhandler;
    evt.OnSomeEvent();

    Console.WriteLine();

    // Попробовать удалить обработчик еще раз.
    Console.WriteLine("Попытка удалить обработчик " +
                      "xOb.Xhandler еще раз.");
    evt.SomeEvent -= xOb.Xhandler;
    evt.OnSomeEvent();

    Console.WriteLine();

    // А теперь добавить обработчик Zhandler.
    Console.WriteLine("Добавление обработчика zOb.Zhandler.");
    evt.SomeEvent += zOb.Zhandler;
    evt.OnSomeEvent();
  }
}
```

Ниже приведен результат выполнения этой программы:

```
Добавление событий.
Список событий заполнен.
```

```
Событие получено объектом W
Событие получено объектом X
Событие получено объектом Y

Удаление обработчика xOb.Xhandler.
Событие получено объектом W
Событие получено объектом Y

Попытка удалить обработчик xOb.Xhandler еще раз.
Обработчик событий не найден.
Событие получено объектом W
Событие получено объектом Y

Добавление обработчика zOb.Zhandler.
Событие получено объектом W
Событие получено объектом X
Событие получено объектом Y
```

Рассмотрим данную программу более подробно. Сначала в ней определяется делегат обработчиков событий `MyEventHandler`. Затем объявляется класс `MyEvent`. В самом его начале определяется массив обработчиков событий `evnt`, состоящий из трех элементов.

```
MyEventHandler[] evnt = new MyEventHandler[3];
```

Этот массив служит для хранения обработчиков событий, добавляемых в цепочку событий. По умолчанию элементы массива `evnt` инициализируются пустым значением (`null`).

Далее объявляется событие `SomeEvent`. В этом объявлении используется приведенная ниже аксессорная форма оператора `event`.

```
public event MyEventHandler SomeEvent {
  // Добавить событие в список.
  add {
    int i;
    for(i=0; i < 3; i++)
      if(evnt[i] == null) {
        evnt[i] = value;
        break;
      }
    if (i == 3) Console.WriteLine("Список событий заполнен.");
  }

  // Удалить событие из списка.
  remove {
    int i;
     for(i=0; i < 3; i++)
      if(evnt[i] == value) {
        evnt[i] = null;
        break;
      }
    if (i == 3) Console.WriteLine("Обработчик событий не найден.");
  }
}
```

Когда в цепочку событий добавляется обработчик событий, вызывается аксессор `add`, и в первом неиспользуемом (т.е. пустом) элементе массива `evnt` запоминается ссылка на этот обработчик, содержащаяся в неявно задаваемом параметре `value`. Если в массиве отсутствуют свободные элементы, то выдается сообщение об ошибке. (Разумеется, в реальном коде при переполнении списка лучше сгенерировать соответствующее исключение.) Массив `evnt` состоит всего из трех элементов, поэтому в нем можно сохранить только три обработчика событий. Когда же обработчик событий удаляется из цепочки событий, то вызывается аксессор `remove` и в массиве `evnt` осуществляется поиск ссылки на этот обработчик, передаваемой в качестве параметра `value`. Если ссылка найдена, то соответствующему элементу массива присваивается пустое значение (`null`), а значит, обработчик удаляется из цепочки событий.

При запуске события вызывается метод `OnSomeEvent()`. В этом методе происходит циклическое обращение к элементам массива `evnt` для вызова по очереди каждого обработчика событий.

Как демонстрирует рассматриваемый здесь пример программы, механизм хранения обработчиков событий нетрудно реализовать, если в этом есть потребность. Но для большинства приложений более подходящим оказывается используемый по умолчанию механизм хранения обработчиков событий, который обеспечивает форма оператора `event` без аксессоров. Тем не менее аксессорная форма оператора `event` используется в особых случаях. Так, если обработчики событий необходимо выполнять в программе в порядке их приоритетности, а не в том порядке, в каком они вводятся в цепочку событий, то для их хранения можно воспользоваться очередью по приоритету.

ПРИМЕЧАНИЕ

В многопоточных приложениях обычно приходится синхронизировать доступ к аксессорам событий. Подробнее о многопоточном программировании речь пойдет в главе 23.

Разнообразные возможности событий

События могут быть определены и в интерфейсах. При этом события должны предоставляться классами, реализующими интерфейсы. События могут быть также определены как абстрактные (`abstract`). В этом случае конкретное событие должно быть реализовано в производном классе. Но аксессорные формы событий не могут быть абстрактными. Кроме того, событие может быть определено как герметичное (`sealed`). И наконец, событие может быть виртуальным, т.е. его можно переопределить в производном классе.

Применение анонимных методов и лямбда-выражений вместе с событиями

Анонимные методы и лямбда-выражения особенно удобны для работы с событиями, поскольку обработчик событий зачастую вызывается только в коде, реализующем механизм обработки событий. Это означает, что создавать автономный метод, как правило, нет никаких причин. А с помощью лямбда-выражений или анонимных методов можно существенно упростить код обработки событий.

Как упоминалось выше, лямбда-выражениям теперь отдается большее предпочтение по сравнению с анонимными методами, поэтому начнем именно с них. Ниже приведен пример программы, в которой лямбда-выражение используется в качестве обработчика событий.

```
// Использовать лямбда-выражение в качестве обработчика событий.

using System;

// Объявить тип делегата для события.
delegate void MyEventHandler(int n);

// Объявить класс, содержащий событие.
class MyEvent {
  public event MyEventHandler SomeEvent;

  // Этот метод вызывается для запуска события.
  public void OnSomeEvent(int n) {
    if(SomeEvent != null)
      SomeEvent(n);
  }
}

class LambdaEventDemo {
  static void Main() {
    MyEvent evt = new MyEvent();

    // Использовать лямбда-выражение в качестве обработчика событий.
    evt.SomeEvent += (n) =>
            Console.WriteLine("Событие получено. Значение равно " + n);

    // Запустить событие.
    evt.OnSomeEvent(1);
    evt.OnSomeEvent(2);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Событие получено. Значение равно 1
Событие получено. Значение равно 2
```

Обратите особое внимание на то, как в этой программе лямбда-выражение используется в качестве обработчика событий.

```
evt.SomeEvent += (n) =>
       Console.WriteLine("Событие получено. Значение равно " + n);
```

Синтаксис для использования лямбда-выражения в качестве обработчика событий остается таким же, как для его применения вместе с любым другим типом делегата.

Несмотря на то что при создании анонимной функции предпочтение следует теперь отдавать лямбда-выражениям, в качестве обработчика событий можно по-прежнему использовать анонимный метод. Ниже приведен вариант обработчика событий из предыдущего примера, измененный с целью продемонстрировать применение анонимного метода.

```
// Использовать анонимный метод в качестве обработчика событий.
evt.SomeEvent += delegate(int n) {
        Console.WriteLine("Событие получено. Значение равно " + n);
};
```

Как видите, синтаксис использования анонимного метода в качестве обработчика событий остается таким же, как и для его применения вместе с любым другим типом делегата.

Рекомендации по обработке событий в среде .NET Framework

В C# разрешается формировать какие угодно разновидности событий. Но ради совместимости программных компонентов со средой .NET Framework следует придерживаться рекомендаций, установленных для этой цели корпорацией Microsoft. Эти рекомендации, по существу, сводятся к следующему требованию: у обработчиков событий должны быть два параметра. Первый из них — ссылка на объект, формирующий событие, второй — параметр типа `EventArgs`, содержащий любую дополнительную информацию о событии, которая требуется обработчику. Таким образом, .NET-совместимые обработчики событий должны иметь следующую общую форму.

```
void обработчик(object отправитель, EventArgs e) {
  // ...
}
```

Как правило, *отправитель* — это параметр, передаваемый вызывающим кодом с помощью ключевого слова `this`. А параметр *e* типа `EventArgs` содержит дополнительную информацию о событии и может быть проигнорирован, если он не нужен.

Сам класс `EventArgs` не содержит поля, которые могут быть использованы для передачи дополнительных данных обработчику. Напротив, `EventArgs` служит в качестве базового класса, от которого получается производный класс, содержащий все необходимые поля. Тем не менее в классе `EventArgs` имеется одно поле `Empty` типа `static`, которое представляет собой объект типа `EventArgs` без данных.

Ниже приведен пример программы, в которой формируется .NET-совместимое событие.

```
// Пример формирования .NET-совместимого события.

using System;

// Объявить класс, производный от класса EventArgs.
class MyEventArgs : EventArgs {
  public int EventNum;
}

// Объявить тип делегата для события.
delegate void MyEventHandler(object source, MyEventArgs arg);

// Объявить класс, содержащий событие.
class MyEvent {
  static int count = 0;

  public event MyEventHandler SomeEvent;
```

```
  // Этот метод запускает событие SomeEvent.
  public void OnSomeEvent() {
    MyEventArgs arg = new MyEventArgs();

    if(SomeEvent != null) {
      arg.EventNum = count++;
      SomeEvent(this, arg);
    }
  }
}

class X {
  public void Handler(object source, MyEventArgs arg) {
    Console.WriteLine("Событие " + arg.EventNum +
                      " получено объектом класса X.");
    Console.WriteLine("Источник: " + source);
    Console.WriteLine();
  }
}

class Y {
  public void Handler(object source, MyEventArgs arg) {
    Console.WriteLine("Событие " + arg.EventNum +
                      " получено объектом класса Y.");
    Console.WriteLine("Источник: " + source);
    Console.WriteLine();
  }
}

class EventDemo6 {
  static void Main() {
    X ob1 = new X();
    Y ob2 = new Y();
    MyEvent evt = new MyEvent();

    // Добавить обработчик Handler() в цепочку событий.
    evt.SomeEvent += ob1.Handler;
    evt.SomeEvent += ob2.Handler;

    // Запустить событие.
    evt.OnSomeEvent();
    evt.OnSomeEvent();
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Событие 0 получено объектом класса X
Источник: MyEvent

Событие 0 получено объектом класса Y
Источник: MyEvent

Событие 1 получено объектом класса X
Источник: MyEvent
```

```
Событие 1 получено объектом класса Y
Источник: MyEvent
```

В данном примере создается класс `MyEventArgs`, производный от класса `EventArgs`. В классе `MyEventArgs` добавляется лишь одно его собственное поле: `EventNum`. Затем объявляется делегат `MyEventHandler`, принимающий два параметра, требующиеся для среды .NET Framework. Как пояснялось выше, первый параметр содержит ссылку на объект, формирующий событие, а второй параметр — ссылку на объект класса `EventArgs` или производного от него класса. Обработчики событий `Handler()`, определяемые в классах `X` и `Y`, принимают параметры тех же самых типов.

В классе `MyEvent` объявляется событие `SomeEvent` типа `MyEventHandler`. Это событие запускается в методе `OnSomeEvent()` с помощью делегата `SomeEvent`, которому в качестве первого аргумента передается ссылка `this`, а вторым аргументом служит экземпляр объекта типа `MyEventArgs`. Таким образом, делегату типа `MyEventHandler` передаются надлежащие аргументы в соответствии с требованиями совместимости со средой .NET.

Применение делегатов `EventHandler<TEventArgs>` и `EventHandler`

В приведенном выше примере программы объявлялся собственный делегат события. Но как правило, в этом не никакой необходимости, поскольку в среде .NET Framework предоставляется встроенный обобщенный делегат под названием `EventHandler<TEventArgs>`. (Более подробно обобщенные типы рассматриваются в главе 18.) В данном случае тип `TEventArgs` обозначает тип аргумента, передаваемого параметру `EventArgs` события. Например, в приведенной выше программе событие `SomeEvent` может быть объявлено в классе `MyEvent` следующим образом.

```
public event EventHandler<MyEventArgs> SomeEvent;
```

В общем, рекомендуется пользоваться именно таким способом, а не определять собственный делегат.

Для обработки многих событий параметр типа `EventArgs` оказывается ненужным. Поэтому с целью упростить создание кода в подобных ситуациях в среду .NET Framework внедрен необобщенный делегат типа `EventHandler`. Он может быть использован для объявления обработчиков событий, которым не требуется дополнительная информация о событиях. Ниже приведен пример использования делегата `EventHandler`.

```
// Использовать встроенный делегат EventHandler.

using System;

// Объявить класс, содержащий событие.
class MyEvent {
  public event EventHandler SomeEvent; // использовать делегат EventHandler

  // Этот метод вызывается для запуска события.
  public void OnSomeEvent() {
    if(SomeEvent != null)
      SomeEvent(this, EventArgs.Empty);
```

```
  }
}

class EventDemo7 {
  static void Handler(object source, EventArgs arg) {
    Console.WriteLine("Произошло событие");
    Console.WriteLine("Источник: " + source);
  }

  static void Main() {
    MyEvent evt = new MyEvent();
    // Добавить обработчик Handler() в цепочку событий.
    evt.SomeEvent += Handler;

    // Запустить событие.
    evt.OnSomeEvent();
  }
}
```

В данном примере параметр типа `EventArgs` не используется, поэтому в качестве этого параметра передается объект-заполнитель `EventArgs.Empty`. Результат выполнения кода из данного примера следующий.

```
Произошло событие
Источник: MyEvent
```

Практический пример обработки событий

События нередко применяются в таких ориентированных на обмен сообщениями средах, как Windows. В подобной среде программа просто ожидает до тех пор, пока не будет получено конкретное сообщение, а затем она предпринимает соответствующее действие. Такая архитектура вполне пригодна для обработки событий средствами C#, поскольку дает возможность создавать обработчики событий для реагирования на различные сообщения и затем просто вызывать обработчик при получении конкретного сообщения. Так, щелчок левой кнопкой мыши может быть связан с событием `LButtonClick`. При получении сообщения о щелчке левой кнопкой мыши вызывается метод `OnLButtonClick()`, и об этом событии уведомляются все зарегистрированные обработчики.

Разработка программ для Windows, демонстрирующих такой подход, выходит за рамки этой главы, тем не менее, рассмотрим пример, дающий представление о принципе, по которому действует данный подход. В приведенной ниже программе создается обработчик событий, связанных с нажатием клавиш. Всякий раз, когда на клавиатуре нажимается клавиша, запускается событие `KeyPress` при вызове метода `OnKeyPress()`. Следует заметить, что в этой программе формируются .NET-совместимые события и что их обработчики предоставляются в лямбда-выражениях.

```
// Пример обработки событий, связанных с нажатием клавиш на клавиатуре.

using System;

// Создать класс, производный от класса EventArgs и
// хранящий символ нажатой клавиши.
```

```
class KeyEventArgs : EventArgs {
  public char ch;
}

// Объявить класс события, связанного с нажатием клавиш на клавиатуре.
class KeyEvent {
  public event EventHandler <KeyEventArgs> KeyPress;

   // Этот метод вызывается при нажатии клавиши.
  public void OnKeyPress(char key) {
    KeyEventArgs k = new KeyEventArgs();

    if(KeyPress != null) {
      k.ch = key;
      KeyPress(this, k);
    }
  }
}

// Продемонстрировать обработку события типа KeyEvent.
class KeyEventDemo {
  static void Main() {
    KeyEvent kevt = new KeyEvent();
    ConsoleKeyInfo key;
     int count = 0;

    // Использовать лямбда-выражение для отображения факта нажатия клавиши.
    kevt.KeyPress += (sender, e) =>
    Console.WriteLine(" Получено сообщение о нажатии клавиши: " + e.ch);

    // Использовать лямбда-выражение для подсчета нажатых клавиш.
    kevt.KeyPress += (sender, e) =>
    count++; // count — это внешняя переменная

    Console.WriteLine("Введите несколько символов. " +
                      "По завершении введите точку.");
    do {
      key = Console.ReadKey();
      kevt.OnKeyPress(key.KeyChar);
    } while(key.KeyChar != '.');

      Console.WriteLine("Было нажато " + count + " клавиш.");
  }
}
```

Вот, например, к какому результату приводит выполнение этой программы.

```
Введите несколько символов. По завершении введите точку.
t Получено сообщение о нажатии клавиши: t
e Получено сообщение о нажатии клавиши: e
s Получено сообщение о нажатии клавиши: s
t Получено сообщение о нажатии клавиши: t
. Получено сообщение о нажатии клавиши: .
Было нажато 5 клавиш.
```

В самом начале этой программы объявляется класс `KeyEventArgs`, производный от класса `EventArgs` и служащий для передачи сообщения о нажатии клавиши обработчику событий. Затем объявляется обобщенный делегат `EventHandler`, определяющий обработчик событий, связанных с нажатием клавиш. Эти события инкапсулируются в классе `KeyEvent`, где определяется событие `KeyPress`.

В методе `Main()` сначала создается объект `kevt` класса `KeyEvent`. Затем в цепочку событий `kevt.KeyPress` добавляется обработчик, предоставляемый лямбда-выражением. В этом обработчике отображается факт каждого нажатия клавиши, как показано ниже.

```
kevt.KeyPress += (sender, e) =>
    Console.WriteLine(" Получено сообщение о нажатии клавиши: " + e.ch);
```

Далее в цепочку событий `kevt.KeyPress` добавляется еще один обработчик, предоставляемый лямбда-выражением. В этом обработчике подсчитывается количество нажатых клавиш, как показано ниже.

```
kevt.KeyPress += (sender, e) =>
    count++; // count — это внешняя переменная
```

Обратите внимание на то, что `count` является локальной переменной, объявленной в методе `Main()` и инициализированной нулевым значением.

Далее начинает выполняться цикл, в котором метод `kevt.OnKeyPress()` вызывается при нажатии клавиши. Об этом событии уведомляются все зарегистрированные обработчики событий. По окончании цикла отображается количество нажатых клавиш. Несмотря на всю свою простоту, данный пример наглядно демонстрирует саму суть обработки событий средствами C#. Аналогичный подход может быть использован и для обработки других событий. Безусловно, в некоторых случаях анонимные обработчики событий могут оказаться непригодными, и тогда придется внедрить именованные методы.

16 ГЛАВА

Пространства имен, препроцессор и сборки

В этой главе речь пойдет о трех средствах C#, позволяющих улучшить организованность и доступность программы. Этими средствами являются пространства имен, препроцессор и сборки.

Пространства имен

О пространстве имен уже вкратце упоминалось в главе 2 в связи с тем, что это основополагающее понятие для C#. В действительности пространство имен в той или иной степени используется в каждой программе на C#. Потребность в подробном рассмотрении пространств имен не возникала до сих пор потому, что для каждой программы на C# автоматически предоставляется используемое по умолчанию глобальное пространство имен. Следовательно, в примерах программ, представленных в предыдущих главах, использовалось глобальное пространство имен. Но во многих реальных программах приходится создавать собственные пространства имен или же организовать взаимодействие с другими пространствами имен. Подобные пространства будут представлены далее во всех подробностях.

Пространство имен определяет область объявлений, в которой допускается хранить одно множество имен отдельно от другого. По существу, имена, объявленные в одном пространстве имен, не будут вступать в конфликт с аналогичными именами, объявленными в другой области. Так, в библиотеке классов для среды .NET Framework, которая одновременно является библиотекой классов C#, используется пространство имен `System`. Именно поэтому строка кода

```
using System;
```

обычно вводится в самом начале любой программы на C#. Как пояснялось в главе 14, классы ввода-вывода определены в пространстве имен `System.IO`, подчиненном пространству имен `System`. Ему подчинены и многие другие пространства имен, относящиеся к разным частям библиотеки классов C#.

Пространства имен важны потому, что за последние годы в программировании "расплодились" в огромном количестве имена переменных, методов, свойств и классов, применяемых в библиотечных программах, стороннем и собственном коде. Поэтому без отдельных пространств все эти имена будут соперничать за место в глобальном пространстве имен, порождая конфликтные ситуации. Так, если в программе определен класс `Finder`, то этот класс может вступить в конфликт с другим классом `Finder`, доступным в сторонней библиотеке, используемой в этой программе. К счастью, подобного конфликта можно избежать, используя отдельные пространства имен, ограничивающие область видимости объявленных в них имен.

Объявление пространства имен

Пространство имен объявляется с помощью ключевого слова `namespace`. Ниже приведена общая форма объявления пространства имен:

```
namespace имя {
  // члены
}
```

где *имя* обозначает конкретное имя объявляемого пространства имен. При объявлении пространства имен определяется область его действия. Все, что объявляется непосредственно в этом пространстве, оказывается в пределах его области действия. В пространстве имен можно объявить классы, структуры, делегаты, перечисления, интерфейсы или другие пространства имен.

Ниже приведен пример объявления `namespace` для создания пространства имен `Counter`. В этом пространстве локализуется имя, используемое для реализации простого класса вычитающего счетчика `CountDown`.

```
// Объявить пространство имен для счетчиков.

namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
     if(val > 0) return val--;
     else return 0;
    }
```

```
  }
} // Это конец пространства имен Counter.
```

Обратите внимание на то, что класс `CountDown` объявляется в пределах области действия пространства имен `Counter`. Для того чтобы проработать этот пример на практике, поместите приведенный выше код в файл `Counter.cs`.

Ниже приведен пример программы, демонстрирующий применение пространства имен `Counter`.

```
// Продемонстрировать применение пространства имен Counter.

using System;

class NSDemo {
  static void Main() {

    // Обратите внимание на то, как класс CountDown
    // определяется с помощью пространства имен Counter.
    Counter.CountDown cd1 = new Counter.CountDown(10);
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    // Еще раз обратите внимание на то, как класс CountDown
    // определяется с помощью пространства имен Counter.
    Counter.CountDown cd2 = new Counter.CountDown(20);

    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Reset(4);
    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
10 9 8 7 6 5 4 3 2 1 0
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
4 3 2 1 0
```

Для того чтобы скомпилировать эту программу, вы должны включить приведенный выше код в отдельный файл и указать его вместе с упоминавшимся выше файлом, содержащим код объявления пространства имен `Counter`. Если этот код

находится в файле NSDemo.cs, а код объявления пространства имен Counter — в файле Counter.cs, то для компиляции программы используется следующая командная строка.

```
csc NSDemo.cs counter.cs
```

Некоторые важные аспекты данной программы заслуживают более пристального внимания. Во-первых, при создании объекта класса CountDown необходимо дополнительно определить его имя с помощью пространства имен Counter, как показано ниже. Ведь класс CountDown объявлен в пространстве имен Counter.

```
Counter.CountDown cd1 = new Counter.CountDown(10);
```

Это правило можно обобщить: всякий раз, когда используется член пространства имен, его имя необходимо дополнительно определить с помощью этого пространства имен. В противном случае член пространства имен не будет обнаружен компилятором.

Во-вторых, как только объект типа Counter будет создан, дополнительно определять его члены с помощью пространства имен уже не придется. Следовательно, метод cd1.Count() может быть вызван непосредственно без дополнительного указания пространства имен, как в приведенной ниже строке кода.

```
i = cd1.Count();
```

И в-третьих, ради наглядности примера рассматриваемая здесь программа была разделена на два отдельных файла. В одном файле содержится код объявления пространства имен Counter, а в другом — код самой программы NSDemo. Но оба фрагмента кода можно было бы объединить в единый файл. Более того, в одном файле исходного кода может содержаться два или более пространства имен со своими собственными областями объявлений. Когда оканчивается действие внутреннего пространства имен, возобновляется действие внешнего пространства имен — в примере с Counter это глобальное пространство имен. Ради большей ясности в последующих примерах все пространства имен, требующиеся в программе, будут представлены в одном и том же файле. Следует, однако, иметь в виду, что их допускается распределять по отдельным файлам, что практикуется чаще в выходном коде.

Предотвращение конфликтов имен с помощью пространств имен

Главное преимущество пространств имен заключается в том, что объявленные в них имена не вступают в конфликт с именами, объявленными за их пределами. Например, в приведенной ниже программе определяются два пространства имен. Первым из них является представленное ранее пространство имен Counter, а вторым — Counter2. Оба пространства имен содержат классы с одинаковым именем CountDown, но поскольку это разные пространства, то оба класса CountDown не вступают в конфликт друг с другом. Кроме того, оба пространства имен определены в одном и том же файле. Как пояснялось выше, это вполне допустимо. Безусловно, каждое из этих пространств имен можно было бы выделить в отдельный файл, если бы в этом возникла потребность.

```
// Пространства имен предотвращают конфликты имен.

using System;
```

```
// Объявить пространство имен Counter.
namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// Объявить пространство имен Counter2.
namespace Counter2 {
  /* Этот класс CountDown относится к пространству
     имен Counter2 и поэтому не вступает в конфликт
     с аналогичным классом из пространства имен Counter.
  */
  class CountDown {
    public void Count() {
          Console.WriteLine("Это метод Count() из " +
                            "пространства имен Counter2.");
    }
  }
}

class NSDemo2 {
  static void Main() {

    // Это класс CountDown из пространства имен Counter.
    Counter.CountDown cd1 = new Counter.CountDown(10);

    // Это класс CountDown из пространства имен Counter2.
    Counter2.CountDown cd2 = new Counter2.CountDown();
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Count();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
10 9 8 7 6 5 4 3 2 1 0
Это метод Count() из пространства имен Counter2.
```

Как следует из приведенного выше результата, класс `CountDown` из пространства имен `Counter` существует отдельно от класса того же названия из пространства имен `Counter2`, и поэтому конфликт имен не возникает. Несмотря на всю простоту данного примера, он наглядно показывает, как удается избежать конфликта имен в собственном коде и коде, написанном другими разработчиками, поместив классы с одинаковыми именами в разные пространства имен.

Директива `using`

Если в программе присутствуют частые ссылки на члены конкретного пространства имен, то указывать это пространство всякий раз, когда требуется ссылка на него, не очень удобно. Преодолеть это затруднение помогает директива `using`. В подавляющем большинстве приводившихся ранее примеров программ с помощью этой директивы делалось видимым глобальное для C# пространство имен `System`, поэтому она отчасти вам уже знакома. Как и следовало ожидать, с помощью директивы `using` можно сделать видимыми вновь создаваемые пространства имен.

Существуют две формы директивы `using`. Ниже приведена первая из них:

```
using имя;
```

где *имя* обозначает имя того пространства имен, к которому требуется получить доступ. Все члены, определенные в указанном пространстве имен, становятся видимыми, и поэтому могут быть использованы без дополнительного определения их имен. Директиву `using` необходимо вводить в самом начале каждого файла исходного кода перед любыми другими объявлениями или же в начале тела пространства имен.

Приведенная ниже программа является вариантом предыдущего примера, переработанным с целью продемонстрировать применение директивы `using`, делающей видимым создаваемое пространство имен.

```
// Продемонстрировать применение директивы using.

using System;

// Сделать видимым пространство имен Counter.
using Counter;

// Объявить пространство имен для счетчиков.
namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
```

```
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo3 {
  static void Main() {

    // Теперь класс CountDown может быть использован непосредственно.
    CountDown cd1 = new CountDown(10);
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    CountDown cd2 = new CountDown(20);

    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Reset(4);
    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

В эту версию программы внесены два существенных изменения. Первое из них состоит в применении директивы `using` в самом начале программы, как показано ниже.

```
using Counter;
```

Благодаря этому становится видимым пространство имен `Counter`. Второе изменение состоит в том, что класс `CountDown` больше не нужно дополнительно определять с помощью пространства имен `Counter`, как демонстрирует приведенная ниже строка кода из метода `Main()`.

```
CountDown cd1 = new CountDown(10);
```

Теперь пространство имен `Counter` становится видимым, и поэтому класс `CountDown` может быть использован непосредственно.

Рассматриваемая здесь программа иллюстрирует еще одно важное обстоятельство: применение одного пространства имен не отменяет действие другого. Когда пространство имен делается видимым, это просто дает возможность использовать его содержимое без дополнительного определения имен. Следовательно, в данном примере оба пространства имен, `System` и `Counter`, становятся видимыми.

Вторая форма директивы `using`

Вторая форма директивы `using` позволяет определить еще одно имя (так называемый *псевдоним*) типа данных или пространства имен. Эта форма приведена ниже:

```
using псевдоним = имя;
```

где *псевдоним* становится еще одним именем типа (например, типа класса) или пространства имен, обозначаемого как *имя*. После того как псевдоним будет создан, он может быть использован вместо первоначального имени.

Ниже приведен вариант программы из предыдущего примера, измененный с целью показать создание и применение псевдонима `MyCounter` вместо составного имени `Counter.CountDown`.

```
// Продемонстрировать применение псевдонима.

using System;

// Создать псевдоним для составного имени Counter.CountDown.
using MyCounter = Counter.CountDown;

// Объявить пространство имен для счетчиков.
namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
    if(val > 0) return val--;
    else return 0;
    }
  }
}

class NSDemo4 {
  static void Main() {

    // Здесь и далее псевдоним MyCounter используется
    // вместо составного имени Counter.CountDown.
```

```
    MyCounter cd1 = new MyCounter(10);
    int i;

    do {
      i = cd1.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    MyCounter cd2 = new MyCounter(20);

    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.Reset(4);
    do {
      i = cd2.Count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Псевдоним `MyCounter` создается с помощью следующего оператора.

```
using MyCounter = Counter.CountDown;
```

После того как псевдоним будет определен в качестве другого имени класса `Counter.CountDown`, его можно использовать для объявления объектов без дополнительного определения имени данного класса. Например, в следующей строке кода из рассматриваемой здесь программы создается объект класса `CountDown`.

```
MyCounter cd1 = new MyCounter(10);
```

Аддитивный характер пространств имен

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода. Например, в приведенной ниже программе два пространства имен определяются под одним и тем же именем `Counter`. Одно из них содержит класс `CountDown`, а другое — класс `CountUp`. Во время компиляции содержимое обоих пространств имен `Counter` складывается.

```
// Аддитивный характер пространств имен.

using System;

// Сделать видимым пространство имен Counter.
using Counter;

// Это одно пространство имен Counter.
```

```
namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    public void Reset(int n) {
      val = n;
    }

    public int Count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// А это другое пространство имен Counter.
namespace Counter {
  // Простой суммирующий счетчик.
  class CountUp {
    int val;
    int target;

    public int Target {
      get{
        return target;
      }
    }

    public CountUp(int n) {
      target = n;
      val = 0;
    }

    public void Reset(int n) {
      target = n;
      val = 0;
    }

    public int Count() {
    if(val < target) return val++;
    else return target;
    }
  }
}

class NSDemo5 {
  static void Main() {
    CountDown cd = new CountDown(10);
    CountUp cu = new CountUp(8);
```

```
    int i;

    do {
      i = cd.Count();
      Console.Write(i + " ");
     } while(i > 0);
     Console.WriteLine();

    do {
      i = cu.Count();
      Console.Write(i + " ");
    } while(i < cu.Target);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Обратите также внимание на то, что директива

```
using Counter;
```

делает видимым все содержимое пространства имен `Counter`. Это дает возможность обращаться к классам `CountDown` и `CountUp` непосредственно, т.е. без дополнительного указания пространства имен. При этом разделение пространства имен `Counter` на две части не имеет никакого значения.

Вложенные пространства имен

Одно пространство имен может быть вложено в другое. В качестве примера рассмотрим следующую программу.

```
// Вложенные пространства имен.

using System;

namespace NS1 {
  class ClassA {
    public ClassA() {
    Console.WriteLine("Конструирование класса ClassA");
    }
  }
  namespace NS2 { // вложенное пространство имен
    class ClassB {
      public ClassB() {
        Console.WriteLine("Конструирование класса ClassB");
      }
    }
  }
}

class NestedNSDemo {
  static void Main() {
```

```
    NS1.ClassA a = new NS1.ClassA();

// NS2.ClassB b = new NS2.ClassB(); // Неверно!!! Пространство NS2 невидимо

    NS1.NS2.ClassB b = new NS1.NS2.ClassB(); // Верно!
  }
}
```

Выполнение этой программы дает следующий результат.

```
Конструирование класса ClassA
Конструирование класса ClassB
```

В этой программе пространство имен `NS2` вложено в пространство имен `NS1`. Поэтому для обращения к классу `ClassB` необходимо дополнительно указать пространства имен `NS1` и `NS2`. Указания одного лишь пространства имен `NS2` для этого недостаточно. Как следует из приведенного выше примера, пространства имен дополнительно указываются через точку. Следовательно, для обращения к классу `ClassB` в методе `Main()` необходимо указать его полное имя — `NS1.NS2.ClassB`.

Пространства имен могут быть вложенными больше, чем на два уровня. В этом случае член вложенного пространства имен должен быть дополнительно определен с помощью всех охватывающих пространств имен.

Вложенные пространства имен можно указать в одном операторе `namespace`, разделив их точкой. Например, вложенные пространства имен

```
namespace OuterNS {
  namespace InnerNS {
    // ...
  }
}
```

могут быть указаны следующим образом.

```
namespace OuterNS.InnerNS {
  // ...
}
```

Глобальное пространство имен

Если в программе не объявлено пространство имен, то по умолчанию используется глобальное пространство имен. Именно поэтому в примерах программ, представленных в предыдущих главах книги, не нужно было обращаться для этой цели к ключевому слову `namespace`. Глобальное пространство удобно для коротких программ, как в примерах из этой книги, но в большинстве случаев реальный код содержится в объявляемом пространстве имен. Главная причина инкапсуляции кода в объявляемом пространстве имен — предотвращение конфликтов имен. Пространства имен служат дополнительным средством, помогающим улучшить организацию программ и приспособить их к работе в сложной среде с современной сетевой структурой.

Применение описателя псевдонима пространства имен `::`

Пространства имен помогают предотвратить конфликты имен, но не устранить их полностью. Такой конфликт может, в частности, произойти, когда одно и то же имя

объявляется в двух разных пространствах имен и затем предпринимается попытка сделать видимыми оба пространства. Допустим, что два пространства имен содержат класс `MyClass`. Если попытаться сделать видимыми оба пространства имен с помощью директив `using`, то имя `MyClass` из первого пространства вступит в конфликт с именем `MyClass` из второго пространства, обусловив появление ошибки неоднозначности. В таком случае для указания предполагаемого пространства имен явным образом можно воспользоваться *описателем псевдонима пространства имен* `::`.

Ниже приведена общая форма оператора `::`.

псевдоним_пространства_имен`::`*идентификатор*

Здесь *псевдоним_пространства_имен* обозначает конкретное имя псевдонима пространства имен, а *идентификатор* — имя члена этого пространства.

Для того чтобы стало понятнее назначение описателя псевдонима пространства имен, рассмотрим следующий пример программы, в которой создаются два пространства имен, `Counter` и `AnotherCounter`, и в обоих пространствах объявляется класс `CountDown`. Затем оба пространства имен становятся видимыми с помощью директив `using`. И наконец, в методе `Main()` предпринимается попытка получить экземпляр объекта типа `CountDown`.

```
// Продемонстрировать необходимость описателя ::.

using System;

// Использовать оба пространства имен Counter и AnotherCounter.

using Counter;
using AnotherCounter;

// Объявить пространство имен для счетчиков.
namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;
    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

// Объявить еще одно пространство имен для счетчиков.
namespace AnotherCounter {
  // Объявить еще один класс CountDown, принадлежащий
  // пространству имен AnotherCounter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
```

```
  }
}

class WhyAliasQualifier {
  static void Main() {
    int i;

    // Следующая строка, по существу, неоднозначна!
    // Неясно, делается ли в ней ссылка на класс CountDown
    // из пространства имен Counter или AnotherCounter?
    CountDown cd1 = new CountDown(10); // Ошибка! ! !

    // ...
  }
}
```

Если попытаться скомпилировать эту программу, то будет получено сообщение об ошибке, уведомляющее о неоднозначности в следующей строке кода из метода `Main()`.

```
CountDown cd1 = new CountDown(10); // Ошибка! ! !
```

Причина подобной неоднозначности заключается в том, что в обоих пространствах имен, `Counter` и `AnotherCounter`, объявлен класс `CountDown` и оба пространства сделаны видимыми. Поэтому неясно, к какому именно варианту класса `CountDown` следует отнести приведенное выше объявление. Для устранения подобного рода недоразумений и предназначен описатель `::`.

Для того чтобы воспользоваться описателем `::`, необходимо сначала определить псевдоним для пространства имен, которое требуется описать, а затем дополнить описание неоднозначного элемента этим псевдонимом. Ниже приведен вариант предыдущего примера программы, в котором устраняется упомянутая выше неоднозначность.

```
// Продемонстрировать применение описателя ::.

using System;
using Counter;
using AnotherCounter;

// Присвоить классу Counter псевдоним Ctr.
using Ctr = Counter;

// Объявить пространство имен для счетчиков.
namespace Counter {

  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
```

```
  }
}

// Объявить еще одно пространство имен для счетчиков.
namespace AnotherCounter {
  // Объявить еще один класс CountDown, принадлежащий
  // пространству имен AnotherCounter.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

class AliasQualifierDemo {
  static void Main() {
    // Здесь оператор :: разрешает конфликт, предписывая компилятору
    // использовать класс CountDown из пространства имен Counter.
    Ctr::CountDown cd1 = new Ctr::CountDown(10);

    // ...
  }
}
```

В этом варианте программы для класса `Counter` сначала указывается псевдоним `Ctr` в следующей строке кода.

```
using Ctr = Counter;
```

А затем этот псевдоним используется в методе `Main()` для дополнительного описания класса `CountDown`, как показано ниже.

```
Ctr::CountDown cd1 = new Ctr::CountDown(10);
```

Описатель `::` устраняет неоднозначность, поскольку он явно указывает на то, что следует обратиться к классу `CountDown` из пространства `Ctr`, а фактически — `Counter`. Именно это и делает теперь программу пригодной для компиляции.

Описатель `::` можно также использовать вместе с предопределенным идентификатором `global` для ссылки на глобальное пространство имен. Например, в приведенной ниже программе класс `CountDown` объявляется как в пространстве имен `Counter`, так и в глобальном пространстве имен. А для доступа к варианту класса `CountDown` в глобальном пространстве имен служит предопределенный псевдоним `global`.

```
// Использовать псевдоним глобального пространства имен.

using System;

// Присвоить классу Counter псевдоним Ctr.
using Ctr = Counter;

// Объявить пространство имен для счетчиков.
```

```
namespace Counter {
  // Простой вычитающий счетчик.
  class CountDown {
    int val;

    public CountDown(int n) {
      val = n;
    }

    // ...
  }
}

// Объявить еще один класс CountDown, принадлежащий
// глобальному пространству имен.
class CountDown {
  int val;

  public CountDown(int n) {
    val = n;
  }

  // ...
}

class GlobalAliasQualifierDemo {
  static void Main() {

    // Здесь описатель :: предписывает компилятору использовать
    // класс CountDown из пространства имен Counter.
    Ctr::CountDown cd1 = new Ctr::CountDown(10);

    // Далее создать объект класса CountDown из
    // глобального пространства имен.
    global::CountDown cd2 = new global::CountDown(10);

    // ...
  }
}
```

Обратите внимание на то, что идентификатор `global` служит для доступа к классу `CountDown` из используемого по умолчанию пространства имен.

```
global::CountDown cd2 = new global::CountDown(10);
```

Этот подход можно распространить на любую ситуацию, в которой требуется указывать используемое по умолчанию пространство имен.

И последнее: описатель псевдонима пространства имен можно применять вместе с псевдонимами типа `extern`, как будет показано в главе 20.

Препроцессор

В C# определен ряд директив препроцессора, оказывающих влияние на интерпретацию исходного кода программы компилятором. Эти директивы определяют поря-

док интерпретации текста программы перед ее трансляцией в объектный код в том исходном файле, где они появляются. Термин *директива препроцессора* появился в связи с тем, что подобные инструкции по традиции обрабатывались на отдельной стадии компиляции, называемой *препроцессором*. Обрабатывать директивы на отдельной стадии препроцессора в современных компиляторах уже не нужно, но само ее название закрепилось.

Ниже приведены директивы препроцессора, определенные в C#.

`#define`	`#elif`	`#else`	`#endif`
`#endregion`	`#error`	`#if`	`#line`
`#pragma`	`#region`	`#undef`	`#warning`

Все директивы препроцессора начинаются со знака #. Кроме того, каждая директива препроцессора должна быть выделена в отдельную строку кода.

Принимая во внимание современную объектно-ориентированную архитектуру языка C#, потребность в директивах препроцессора в нем не столь велика, как в языках программирования предыдущих поколений. Тем не менее они могут быть иногда полезными, особенно для условной компиляции. В этом разделе все директивы препроцессора рассматриваются по очереди.

Директива #define

Директива `#define` определяет последовательность символов, называемую *идентификатором*. Присутствие или отсутствие идентификатора может быть определено с помощью директивы `#if` или `#elif` и поэтому используется для управления процессом компиляции. Ниже приведена общая форма директивы `#define`.

```
#define идентификатор
```

Обратите внимание на отсутствие точки с запятой в конце этого оператора. Между директивой `#define` и идентификатором может быть любое количество пробелов, но после самого идентификатора должен следовать только символ новой строки. Так, для определения идентификатора `EXPERIMENTAL` служит следующая директива.

```
#define EXPERIMENTAL
```

ПРИМЕЧАНИЕ

В C/C++ директива `#define` может использоваться для подстановки исходного текста, например для определения имени значения, а также для создания макрокоманд, похожих на функции. А в C# такое применение директивы `#define` не поддерживается. В этом языке директива #define служит только для определения идентификатора.

Директивы #if и #endif

Обе директивы, `#if` и `#endif`, допускают условную компиляцию последовательности кода в зависимости от истинного результата вычисления выражения, включающего в себя один или несколько идентификаторов. Идентификатор считается истинным, если он определен, а иначе — ложным. Так, если идентификатор определен директивой `#define`, то он будет оценен как истинный. Ниже приведена общая форма директивы `#if`.

```
#if идентификаторное_выражение
  последовательность операторов
#endif
```

Если *идентификаторное_выражение*, следующее после директивы `#if`, истинно, то компилируется код (*последовательность операторов*), указываемый между ним и директивой `#endif`. В противном случае этот промежуточный код пропускается. Директива `#endif` обозначает конец блока директивы `#if`.

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: `!`, `==`, `!=`, `&&` и `||`, а также круглых скобок.

Ниже приведен пример применения упомянутых выше директив.

```
// Продемонстрировать применение директив
// #if, #endif и #define.

#define EXPERIMENTAL

using System;

class Test {
  static void Main() {
    #if EXPERIMENTAL
      Console.WriteLine("Компилируется для экспериментальной версии.");
    #endif

    Console.WriteLine("Присутствует во всех версиях.");
  }
}
```

Этот код выдает следующий результат.

```
Компилируется для экспериментальной версии.
Присутствует во всех версиях.
```

В приведенном выше коде определяется идентификатор `EXPERIMENTAL`. Поэтому когда в этом коде встречается директива `#if`, идентификаторное выражение вычисляется как истинное и затем компилируется первый оператор, содержащий вызов метода `WriteLine()`. Если же удалить определение идентификатора `EXPERIMENTAL` и перекомпилировать данный код, то первый оператор, содержащий вызов метода `WriteLine()`, не будет скомпилирован, поскольку идентификаторное выражение директивы `#if` вычисляется как ложное. Но второй оператор, содержащий вызов метода `WriteLine()`, компилируется в любом случае, потому что он не входит в блок директивы `#if`.

Как пояснялось выше, в директиве `#if` допускается указывать идентификаторное выражение. В качестве примера рассмотрим следующую программу.

```
// Использовать идентификаторное выражение.

#define EXPERIMENTAL
#define TRIAL

using System;
```

```
class Test {
  static void Main() {
    #if EXPERIMENTAL
      Console.WriteLine("Компилируется для экспериментальной версии.");
    #endif

    #if EXPERIMENTAL && TRIAL
      Console.Error.WriteLine("Проверка пробной экспериментальной
версии.");
    #endif

    Console.WriteLine("Присутствует во всех версиях.");
  }
}
```

Эта программа дает следующий результат.

```
Компилируется для экспериментальной версии.
Проверка пробной экспериментальной версии.
Присутствует во всех версиях.
```

В данном примере определены два идентификатора: `EXPERIMENTAL` и `TRIAL`. Второй оператор, содержащий вызов метода `WriteLine()`, компилируется *лишь* в том случае, если определены оба идентификатора.

Для компилирования кода в том случае, если идентификатор не определен, можно воспользоваться оператором `!`, как в приведенном ниже примере.

```
#if !EXPERIMENTAL
  Console.WriteLine("Этот код не экспериментальный!");
#endif
```

Вызов метода будет скомпилирован только в том случае, если идентификатор `EXPERIMENTAL` *не* определен.

Директивы #else и #elif

Директива `#else` действует аналогично условному оператору `else` языка C#, определяя альтернативный ход выполнения программы, если этого не может сделать директива `#if`. С учетом директивы `#else` предыдущий пример программы может быть расширен следующим образом.

```
// Продемонстрировать применение директивы #else.

#define EXPERIMENTAL

using System;

class Test {
  static void Main() {
    #if EXPERIMENTAL
      Console.WriteLine("Компилируется для экспериментальной версии.");
    #else
      Console.WriteLine("Компилируется для окончательной версии.");
    #endif
```

```
    #if EXPERIMENTAL && TRIAL
      Console.Error.WriteLine("Проверка пробной экспериментальной
версии.");
    #else
      Console.Error.WriteLine("Это не пробная экспериментальная версия.");
    #endif

    Console.WriteLine("Присутствует во всех версиях.");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Компилируется для экспериментальной версии.
Это не пробная экспериментальная версия.
Присутствует во всех версиях.
```

В данном примере идентификатор `TRIAL` не определен, и поэтому часть `#else` второй условной последовательности кода не компилируется.

Обратите внимание на то, что директива `#else` обозначает конец блока директивы `#if` и в то же время — начало блока самой директивы `#else`. Это необходимо потому, что с любой директивой `#if` может быть связана только одна директива `#endif`. Более того, с любой директивой `#if` может быть связана только одна директива `#else`.

Обозначение `#elif` означает "иначе если", а сама директива `#elif` определяет последовательность условных операций `if-else-if` для многовариантной компиляции. После директивы `#elif` указывается идентификаторное выражение. Если это выражение истинно, то компилируется следующий далее кодовый блок, а остальные выражения директивы `#elif` не проверяются. В противном случае проверяется следующий по порядку блок. Если же ни одну из директив `#elif` не удается выполнить, то при наличии директивы `#else` выполняется последовательность кода, связанная с этой директивой, а иначе не компилируется ни один из кодовых блоков директивы `#if`.

Ниже приведена общая форма директивы `#elif`.

```
#if идентификаторное_выражение
  последовательность операторов
#elif идентификаторное_выражение
  последовательность операторов
#elif идентификаторное_выражение
  последовательность операторов
// . . .
#endif
```

В приведенном ниже примере демонстрируется применение директивы `#elif`.

```
// Продемонстрировать применение директивы #elif.

#define RELEASE

using System;

class Test {
  static void Main() {
```

```
    #if EXPERIMENTAL
      Console.WriteLine("Компилируется для экспериментальной версии.");
    #elif RELEASE
      Console.WriteLine("Компилируется для окончательной версии.");
    #else
      Console.WriteLine("Компилируется для внутреннего тестирования.");
    #endif

    #if TRIAL && !RELEASE
      Console.WriteLine("Пробная версия.");
    #endif

    Console.WriteLine("Присутствует во всех версиях.");
  }
}
```

Этот код выдает следующий результат.

```
Компилируется для окончательной версии.
Присутствует во всех версиях.
```

Директива #undef

С помощью директивы `#undef` удаляется определенный ранее идентификатор. Это, по существу, означает, что он становится "неопределенным". Ниже приведена общая форма директивы `#undef`.

```
#undef идентификатор
```

Рассмотрим следующий пример кода.

```
#define SMALL

#if SMALL
  // ...
#undef SMALL
// теперь идентификатор SMALL не определен.
```

После директивы `#undef` идентификатор `SMALL` уже оказывается неопределенным.

Директива `#undef` применяется главным образом для локализации идентификаторов только в тех фрагментах кода, в которых они действительно требуются.

Директива #error

Директива `#error` вынуждает компилятор прервать компиляцию. Она служит в основном для отладки. Ниже приведена общая форма директивы `#error`.

```
#error сообщение_об_ошибке
```

Когда в коде встречается директива `#error`, выводится сообщение об ошибке. Например, когда компилятору встречается строка кода

```
#error Это тестовая ошибка!
```

компиляция прерывается и выводится сообщение `"Это тестовая ошибка!"`.

Директива #warning

Директива `#warning` действует аналогично директиве `#error`, за исключением того, что она выводит предупреждение, а не ошибку. Следовательно, компиляция не прерывается. Ниже приведена общая форма директивы `#warning`.

```
#warning предупреждающее_сообщение
```

Директива #line

Директива `#line` задает номер строки и имя файла, содержащего эту директиву. Номер строки и имя файла используются при выводе ошибок или предупреждений во время компиляции. Ниже приведена общая форма директивы `#line`.

```
#line номер "имя_файла"
```

Имеются еще два варианта директивы `#line`. В первом из них она указывается с ключевым словом `default`, обозначающим возврат нумерации строк в исходное состояние, как в приведенном ниже примере.

```
#line default
```

А во втором варианте директива `#line` указывается с ключевым словом `hidden`. При пошаговой отладке программы строки кода, находящиеся между директивой

```
#line hidden
```

и следующей директивой `#line` без ключевого слова `hidden`, пропускаются отладчиком.

Директивы #region и #endregion

С помощью директив `#region` и `#endregion` определяется область, которая разворачивается или сворачивается при структурировании исходного кода в интегрированной среде разработки Visual Studio. Ниже приведена общая форма этих директив:

```
#region текст
  // последовательность кода
#endregion текст
```

где *текст* обозначает необязательную символьную строку.

Директива #pragma

С помощью директивы `#pragma` инструкции задаются компилятору в виде опций. Ниже приведена общая форма этой директивы:

```
#pragma опция
```

где *опция* обозначает инструкцию, передаваемую компилятору.

В текущей версии C# предусмотрены две опции для директивы `#pragma`. Первая из них, `warning`, служит для разрешения или запрета отдельных предупреждений со стороны компилятора. Она принимает две формы:

```
#pragma warning disable предупреждения
#pragma warning restore предупреждения
```

где *предупреждения* обозначает разделяемый запятыми список номеров предупреждений. Для отмены предупреждения используется опция `disable`, а для его разрешения — опция `restore`.

Например, в приведенной ниже директиве `#pragma` запрещается выдача предупреждения №168, уведомляющего о том, что переменная объявлена, но не используется.

```
#pragma warning disable 168
```

Второй для директивы `#pragma` является опция `checksum`. Она служит для формирования контрольной суммы в проектах ASP.NET. Ниже приведена ее общая форма:

```
#pragma checksum "имя_файла" "{GUID}" "контрольная_сумма"
```

где *имя_файла* обозначает конкретное имя файла; *GUID* — глобально уникальный идентификатор, с которым связано *имя_файла*; *контрольная_сумма* — шестнадцатеричное число, представляющее контрольную сумму. У этой контрольной суммы должно быть четное число цифр.

Сборки и модификатор доступа `internal`

Сборка является неотъемлемой частью программирования на C#. Она представляет собой один или несколько файлов, содержащих все необходимые сведения о развертывании программы и ее версии. Сборки составляют основу среды .NET. Они предоставляют механизмы для надежного взаимодействия компонентов, межъязыковой возможности взаимодействия и управления версиями. Кроме того, сборки определяют область действия программного кода.

Сборка состоит из четырех разделов. Первый раздел представляет собой *декларацию* сборки. Декларация содержит сведения о самой сборке. К этой информации относится, в частности, имя сборки, номер ее версии, сведения о соответствии типов и параметры культурной среды (язык и региональные стандарты). Второй раздел сборки содержит метаданные типов, т.е. сведения о типах данных, используемых в программе. Среди прочих преимуществ *метаданные типов* способствуют межъязыковой возможности взаимодействия. Третий раздел сборки содержит *программный код* в формате MSIL (Microsoft Intermediate Language — промежуточный язык корпорации Microsoft). И четвертый раздел сборки содержит ресурсы, используемые программой.

Правда, при программировании на C# сборки получаются автоматически, требуя от программирующего лишь минимальных усилий. Дело в том, что исполняемый файл, создаваемый во время компиляции программы на C#, на самом деле представляет собой сборку, содержащую исполняемый код этой программы, а также другие виды информации. Таким образом, когда компилируется программа на C#, сборка получается автоматически.

У сборок имеется много других особенностей, и с ними связано немало актуальных вопросов программирования, но, к сожалению, их обсуждение выходит за рамки этой книги. Ведь сборки являются неотъемлемой частью процесса разработки программного обеспечения в среде .NET, но формально они не относятся к средствам языка C#. Тем не менее в C# имеется одно средство, непосредственно связанное со сборкой. Это модификатор доступа `internal`, рассматриваемый в следующем разделе.

Модификатор доступа `internal`

Помимо модификаторов доступа `public`, `private` и `protected`, использовавшихся в представленных ранее примерах программ, в C# предусмотрен также модификатор доступа `internal`. Этот модификатор определяет доступность члена во всех файлах сборки и его недоступность за пределами сборки. Проще говоря, о члене, обозначенном как `internal`, известно только в самой программе, но не за ее пределами. Модификатор доступа `internal` особенно полезен для создания программных компонентов.

Модификатор доступа `internal` можно применять к классам и их членам, а также к структурам и членам структур. Кроме того, модификатор `internal` разрешается использовать в объявлениях интерфейсов и перечислений.

Из модификаторов `protected` и `internal` можно составить спаренный модификатор доступа `protected internal`. Уровень доступа `protected internal` может быть задан только для членов класса. Член, объявленный как `protected internal`, доступен лишь в пределах собственной сборки или для производных типов.

Ниже приведен пример применения модификатора доступа `internal`.

```
// Использовать модификатор доступа internal.

using System;

class InternalTest {
  internal int x;
}

class InternalDemo {
  static void Main() {
   InternalTest ob = new InternalTest();

   ob.x = 10; // доступно, потому что находится в том же файле

    Console.WriteLine("Значение ob.x: " + ob.x);
  }
}
```

В классе `InternalTest` поле `x` объявляется как `internal`. Это означает, что поле `x` доступно в самой программе, но, как показывает код класса `InternalDemo`, оно недоступно за пределами программы.

17
ГЛАВА

Динамическая идентификация типов, рефлексия и атрибуты

В этой главе рассматриваются три эффективных средства: динамическая идентификация типов, рефлексия и атрибуты. *Динамическая идентификация типов* представляет собой механизм, позволяющий определить тип данных во время выполнения программы. Рефлексия — это средство для получения сведений о типе данных. Используя эти сведения, можно конструировать и применять объекты во время выполнения. Это довольно эффективное средство, поскольку оно дает возможность расширять функции программы динамически, т.е. в процессе ее выполнения. *Атрибут* описывает характеристики определенного элемента программы на C#. Атрибуты можно, в частности, указать для классов, методов и полей. Во время выполнения программы разрешается опрашивать атрибуты для получения сведений о них. Для этой цели в атрибутах используется динамическая идентификация типов и рефлексия.

Динамическая идентификация типов

Динамическая идентификация типов (RTTI) позволяет определить тип объекта во время выполнения программы. Она оказывается полезной по целому ряду причин. В частности, по ссылке на базовый класс можно довольно точно определить тип объекта, доступного по этой ссылке. Динамическая идентификация типов позволяет также проверить заранее, насколько удачным будет исход приведения типов, предотвращая исключительную ситуацию в связи с неправильным приведением типов. Кроме того, динамическая идентификация типов является главной составляющей рефлексии.

Для поддержки динамической идентификации типов в C# предусмотрены три ключевых слова: `is`, `as` и `typeof`. Каждое из этих ключевых слов рассматривается далее по очереди.

Проверка типа с помощью оператора `is`

Конкретный тип объекта можно определить с помощью оператора `is`. Ниже приведена его общая форма:

выражение `is` *тип*

где *выражение* обозначает отдельное выражение, описывающее объект, *тип* которого проверяется. Если *выражение* имеет совместимый или такой же тип, как и проверяемый *тип*, то результат этой операции получается истинным, в противном случае — ложным. Так, результат будет истинным, если *выражение* имеет проверяемый *тип* в той или иной форме. В операторе `is` оба типа определяются как совместимые, если они одного и того же типа или если предусмотрено преобразование ссылок, упаковка или распаковка.

Ниже приведен пример применения оператора `is`.

```
// Продемонстрировать применение оператора is.

using System;

class A {}
class B : A {}

class UseIs {
  static void Main() {
    A a = new A();
    B b = new B();
    if(a is A)
      Console.WriteLine("a имеет тип A");
    if(b is A)
      Console.WriteLine("b совместим с A, поскольку он производный от A");
    if(a is B)
      Console.WriteLine("Не выводится, поскольку a не производный от B");

    if(b is B)
      Console.WriteLine("B имеет тип B");
    if(a is object)
      Console.WriteLine("a имеет тип object");
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
a имеет тип A
b совместим с A, поскольку он производный от A
b имеет тип B
a имеет тип object
```

Большая часть выражений `is` в приведенном выше примере не требует пояснений, но два из них необходимо все же разъяснить. Прежде всего, обратите внимание на следующую строку кода.

```
if(b is A)
  Console.WriteLine("b совместим с A, поскольку он производный от A");
```

Условный оператор `if` выполняется, поскольку `b` является объектом типа `B`, производным от типа `A`. Но обратное несправедливо. Так, если в строке кода

```
if(a is B)
  Console.WriteLine("Не выводится, поскольку a не производный от B");
```

условный оператор `if` не выполняется, поскольку `a` является объектом типа `A`, не производного от типа `B`. Поэтому `a` *не* относится к типу `B`.

Применение оператора as

Иногда преобразование типов требуется произвести во время выполнения, но не генерировать исключение, если исход этого преобразования окажется неудачным, что вполне возможно при приведении типов. Для этой цели служит оператор `as`, имеющий следующую общую форму:

выражение `as` *тип*

где *выражение* обозначает отдельное выражение, преобразуемое в указанный *тип*. Если исход такого преобразования оказывается удачным, то возвращается ссылка на *тип*, а иначе — пустая ссылка. Оператор `as` может использоваться только для преобразования ссылок, идентичности, упаковки, распаковки.

В некоторых случаях оператор `as` может служить удобной альтернативой оператору `is`. В качестве примера рассмотрим следующую программу, в которой оператор `is` используется для предотвращения неправильного приведения типов.

```
// Использовать оператор is для предотвращения неправильного приведения типов.

using System;

class A {}
class B : A {}

class CheckCast {
  static void Main() {
    A a = new A();
    B b = new B();

    // Проверить, можно ли привести a к типу B.
    if(a is B) // если да, то выполнить приведение типов
      b = (B) a;
    else // если нет, то пропустить приведение типов
      b = null;

    if(b==null)
      Console.WriteLine("Приведение типов b = (B) НЕ допустимо.");
    else
      Console.WriteLine("Приведение типов b = (B) допустимо.");
  }
}
```

Эта программа дает следующий результат.

```
Приведение типов b = (B) НЕ допустимо.
```

Как следует из результата выполнения приведенной выше программы, тип объекта a не совместим с типом B, и поэтому его приведение к типу B не допустимо и предотвращается в условном операторе if. Но такую проверку приходится выполнять в два этапа. Сначала требуется убедиться в обоснованности операции приведения типов, а затем выполнить ее. Оба этапа могут быть объединены в один с помощью оператора as, как демонстрирует приведенная ниже программа.

```
// Продемонстрировать применение оператора as.

using System;

class A {}
class B : A {}

class CheckCast {
  static void Main() {
    A a = new A();
    B b = new B();
    b = a as B; // выполнить приведение типов, если это возможно
    if(b==null)
      Console.WriteLine("Приведение типов b = (B) НЕ допустимо.");
    else
      Console.WriteLine("Приведение типов b = (B) допустимо.");
  }
}
```

Эта программа дает прежний результат.

```
Приведение типов b = (B) НЕ допустимо.
```

В данном варианте программы в одном и том же операторе as сначала проверяется обоснованность операции приведения типов, а затем выполняется сама операция приведения типов, если она допустима.

Применение оператора typeof

Несмотря на всю свою полезность, операторы as и is проверяют лишь совместимость двух типов. Но зачастую требуется информация о самом типе. Для этой цели в C# предусмотрен оператор typeof. Он извлекает объект класса System.Type для заданного типа. С помощью этого объекта можно определить характеристики конкретного типа данных. Ниже приведена общая форма оператора typeof:

```
typeof(тип)
```

где *тип* обозначает получаемый тип. Информация, описывающая *тип*, инкапсулируется в возвращаемом объекте класса Type.

Получив объект класса Type для заданного типа, можно извлечь информацию о нем, используя различные свойства, поля и методы, определенные в классе Type. Класс Type довольно обширен и содержит немало членов, поэтому его рассмотрение придется отложить до следующего раздела, посвященного рефлексии. Но в качестве краткого введения в этот класс ниже приведена программа, в которой используются

три его свойства: FullName, IsClass и IsAbstract. Для получения полного имени типа служит свойство FullName. Свойство IsClass возвращает логическое значение true, если тип относится к классу. А свойство IsAbstract возвращает логическое значение true, если класс является абстрактным.

```
// Продемонстрировать применение оператора typeof.

using System;
using System.IO;

class UseTypeof {
  static void Main() {
    Type t = typeof(StreamReader);

      Console.WriteLine(t.FullName);

      if(t.IsClass) Console.WriteLine("Относится к классу.");
    if(t.IsAbstract) Console.WriteLine("Является абстрактным классом.");
    else Console.WriteLine("Является конкретным классом.");
  }
}
```

Эта программа дает следующий результат.

```
System.IO.StreamReader
Относится к классу.
Является конкретным классом.
```

В данной программе сначала извлекается объект класса Type, описывающий тип StreamReader. Затем выводится полное имя этого типа данных и определяется его принадлежность к классу, а далее — к абстрактному или конкретному классу.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Термин *рефлексия*, или отражение, происходит от принципа действия этого средства: объект класса Type отражает базовый тип, который он представляет. Для получения информации о типе данных объекту класса Type делаются запросы, а он возвращает (отражает) обратно информацию, связанную с определяемым типом. Рефлексия является эффективным механизмом, поскольку она позволяет выявлять и использовать возможности типов данных, известные только во время выполнения.

Многие классы, поддерживающие рефлексию, входят в состав прикладного интерфейса .NET Reflection API, относящегося к пространству имен System.Reflection. Поэтому для применения рефлексии в код программы обычно вводится следующая строка.

```
using System.Reflection;
```

Класс System.Type — ядро подсистемы рефлексии

Класс System.Type составляет ядро подсистемы рефлексии, поскольку он инкапсулирует тип данных. Он содержит многие свойства и методы, которыми можно

пользоваться для получения информации о типе данных во время выполнения. Класс `Type` является производным от абстрактного класса `System.Reflection.MemberInfo`.

В классе `MemberInfo` определены приведенные ниже свойства, доступные только для чтения.

Свойство	Описание
`Type DeclaringType`	Тип класса или интерфейса, в котором объявляется отражаемый член
`MemberTypes MemberType`	Тип члена. Это значение обозначает, является ли член полем, методом, свойством, событием или конструктором
`int MetadataToken`	Значение, связанное к конкретными метаданными
`Module Module`	Объект типа `Module`, представляющий модуль (исполняемый файл), в котором находится отражаемый тип
`string Name`	Имя типа
`Type ReflectedType`	Тип отражаемого объекта

Следует иметь в виду, что свойство `MemberType` возвращает тип `MemberTypes` — перечисление, в котором определяются значения, обозначающие различные типы членов. К их числу относятся следующие.

```
MemberTypes.Constructor
MemberTypes.Method
MemberTypes.Field
MemberTypes.Event
MemberTypes.Property
```

Следовательно, тип члена можно определить, проверив свойство `MemberType`. Так, если свойство `MemberType` имеет значение `MemberTypes.Method`, то проверяемый член является методом.

В класс `MemberInfo` входят два абстрактных метода: `GetCustomAttributes()` и `IsDefined()`. Оба метода связаны с атрибутами. Первый из них получает список специальных атрибутов, имеющих отношение к вызывающему объекту, а второй устанавливает, определен ли атрибут для вызывающего метода. В версию .NET Framework Version 4.0 внедрен метод `GetCustomAttributesData()`, возвращающий сведения о специальных атрибутах. (Подробнее об атрибутах речь пойдет далее в этой главе.)

Класс `Type` добавляет немало своих собственных методов и свойств к числу тех, что определены в классе `MemberInfo`. В качестве примера ниже перечислен ряд наиболее часто используемых методов класса `Type`.

Метод	Назначение
`ConstructorInfo[] GetConstructors()`	Получает список конструкторов для заданного типа
`EventInfo[] GetEvents()`	Получает список событий для заданного типа
`FieldInfo[] GetFields()`	Получает список полей для заданного типа
`Type[] GetGenericArguments()`	Получает список аргументов типа, связанных с закрыто сконструированным обобщенным типом, или же список параметров типа, если заданный тип определен как обобщенный. Для открыто сконструированного типа этот

Окончание таблицы

Метод	Назначение
	список может содержать как аргументы, так и параметры типа. (Более подробно обобщения рассматриваются в главе 18.)
`MemberInfo[] GetMembers()`	Получает список членов для заданного типа
`MethodInfo[] GetMethods()`	Получает список методов для заданного типа
`PropertyInfo[] GetProperties()`	Получает список свойств для заданного типа

Далее приведен ряд наиболее часто используемых свойств, доступных только для чтения и определенных в классе `Type`.

Свойство	Назначение
`Assembly Assembly`	Получает сборку для заданного типа
`TypeAttributes Attributes`	Получает атрибуты для заданного типа
`Type BaseType`	Получает непосредственный базовый тип для заданного типа
`string FullName`	Получает полное имя заданного типа
`bool IsAbstract`	Истинно, если заданный тип является абстрактным
`bool isArray`	Истинно, если заданный тип является массивом
`bool IsClass`	Истинно, если заданный тип является классом
`bool IsEnum`	Истинно, если заданный тип является перечислением
`bool IsGenericParameter`	Истинно, если заданный тип является параметром обобщенного типа. (Более подробно обобщения рассматриваются в главе 18.)
`bool IsGenericType`	Истинно, если заданный тип является обобщенным. (Более подробно обобщения рассматриваются в главе 18.)
`string Namespace`	Получает пространство имен для заданного типа

Применение рефлексии

С помощью методов и свойств класса `Type` можно получить подробные сведения о типе данных во время выполнения программы. Это довольно эффективное средство. Ведь получив сведения о типе данных, можно сразу же вызвать его конструкторы и методы или воспользоваться его свойствами. Следовательно, рефлексия позволяет использовать код, который не был доступен во время компиляции.

Прикладной интерфейс Reflection API весьма обширен и поэтому не может быть полностью рассмотрен в этой главе. Ведь для этого потребовалась бы целая книга! Но прикладной интерфейс Reflection API имеет ясную логическую структуру, а следовательно, уяснив одну его часть, нетрудно понять и все остальное. Принимая во внимание это обстоятельство, в последующих разделах демонстрируются четыре основных способа применения рефлексии: получение сведений о методах, вызов методов, конструирование объектов и загрузка типов данных из сборок.

Получение сведений о методах

Имея в своем распоряжении объект класса `Type`, можно получить список методов, поддерживаемых отдельным типом данных, используя метод `GetMethods()`. Ниже приведена одна из форм, подходящих для этой цели.

```
MethodInfo[] GetMethods()
```

Этот метод возвращает массив объектов класса `MethodInfo`, которые описывают методы, поддерживаемые вызывающим типом. Класс `MethodInfo` находится в пространстве имен `System.Reflection`.

Класс `MethodInfo` является производным от абстрактного класса `MethodBase`, который в свою очередь наследует от класса `MemberInfo`. Это дает возможность пользоваться всеми свойствами и методами, определенными в этих трех классах. Например, для получения имени метода служит свойство `Name`. Особый интерес вызывают два члена класса `MethodInfo`: `ReturnType` и `GetParameters()`.

Возвращаемый тип метода находится в доступном только для чтения свойстве `ReturnType`, которое является объектом класса `Type`.

Метод `GetParameters()` возвращает список параметров, связанных с анализируемым методом. Ниже приведена его общая форма.

```
ParameterInfo[] GetParameters();
```

Сведения о параметрах содержатся в объекте класса `ParameterInfo`. В классе `ParameterInfo` определено немало свойств и методов, описывающих параметры. Особое значение имеют два свойства: `Name` — представляет собой строку, содержащую имя параметра, а `ParameterType` — описывает тип параметра, который инкапсулирован в объекте класса `Type`.

В качестве примера ниже приведена программа, в которой рефлексия используется для получения методов, поддерживаемых классом `MyClass`. В этой программе выводится возвращаемый тип и имя каждого метода, а также имена и типы любых параметров, которые может иметь каждый метод.

```
// Анализ методов с помощью рефлексии.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i, int j) {
    x = i;
    y = j;
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if(x < i && i < y) return true;
    else return false;
```

```
  }

  public void Set(int a, int b) {
    x = a;
    y = b;
  }

  public void Set(double a, double b) {
    x = (int) a;
    y = (int) b;
  }

  public void Show() {
    Console.WriteLine(" x: {0}, y: {1}", x, y);
  }
}

class ReflectDemo {
  static void Main() {
    Type t = typeof(MyClass); // получить объект класса Type,
                              // представляющий класс MyClass

    Console.WriteLine("Анализ методов, определенных " +
                      "в классе " + t.Name);
    Console.WriteLine();

    Console.WriteLine("Поддерживаемые методы: ");

    MethodInfo[] mi = t.GetMethods();

    // Вывести методы, поддерживаемые в классе MyClass.
    foreach(MethodInfo m in mi) {
      // Вывести возвращаемый тип и имя каждого метода.
      Console.Write(" " + m.ReturnType.Name + " " + m.Name + "(");

      // Вывести параметры.
      ParameterInfo[] pi = m.GetParameters();
      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name + " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }
      Console.WriteLine(")");

      Console.WriteLine();
    }
  }
}
```

Эта программа дает следующий результат.

```
Анализ методов, определенных в классе MyClass

Поддерживаемые методы:
   Int32 Sum()
```

```
Boolean IsBetween(Int32 i)

Void Set(Int32 a, Int32 b)

Void Set(Double a, Double b)

Void Show()

String ToString()

Boolean Equals(Object obj)

Int32 GetHashCode()

Type GetType()
```

Как видите, помимо методов, определенных в классе `MyClass`, в данной программе выводятся также методы, определенные в классе `object`, поскольку все типы данных в C# наследуют от класса `object`. Кроме того, в качестве имен типов указываются имена структуры .NET. Обратите также внимание на то, что метод `Set()` выводится дважды, поскольку он перегружается. Один из его вариантов принимает аргументы типа `int`, а другой — аргументы типа `double`.

Рассмотрим эту программу более подробно. Прежде всего следует заметить, что в классе `MyClass` определен открытый конструктор и ряд открытых методов, в том числе и перегружаемый метод `Set()`.

Объект класса `Type`, представляющий класс `MyClass`, создается в методе `Main()` в следующей строке кода.

```
Type t = typeof(MyClass); // получить объект класса Type,
                          // представляющий класс MyClass
```

Напомним, что оператор `typeof` возвращает объект класса `Type`, представляющий конкретный тип данных (в данном случае — класс `MyClass`).

С помощью переменной `t` и прикладного интерфейса `Reflection API` в данной программе затем выводятся сведения о методах, поддерживаемых в классе `MyClass`. Для этого в приведенной ниже строке кода сначала выводится список соответствующих методов.

```
MethodInfo[] mi = t.GetMethods();
```

Затем в цикле `foreach` организуется обращение к элементам массива `mi`. На каждом шаге этого цикла выводится возвращаемый тип, имя и параметры отдельного метода, как показано в приведенном ниже фрагменте кода.

```
foreach(MethodInfo m in mi) {
  // Вывести возвращаемый тип и имя каждого метода.
    Console.Write(" " + m.ReturnType.Name + " " + m.Name + "(");

     // Вывести параметры.
     ParameterInfo[] pi = m.GetParameters();
     for(int i=0; i < pi.Length; i++) {
       Console.Write(pi[i].ParameterType.Name + " " + pi[i].Name);
       if(i+1 < pi.Length) Console.Write(", ");
     }
```

В этом фрагменте кода параметры, связанные с каждым методом, сначала создаются с помощью метода `GetParameters()` и сохраняются в массиве `pi`. Затем в цикле `for` происходит обращение к элементам массива `pi` и выводится тип и имя каждого параметра. Самое главное, что все эти сведения создаются динамически во время выполнения программы, не опираясь на предварительную осведомленность о классе `MyClass`.

Вторая форма метода `GetMethods()`

Существует вторая форма метода `GetMethods()`, позволяющая указывать различные флажки для отфильтровывания извлекаемых сведений о методах. Ниже приведена эта общая форма метода `GetMethods()`.

```
MethodInfo[] GetMethods(BindingFlags флажки)
```

В этом варианте создаются только те методы, которые соответствуют указанным критериям. `BindingFlags` представляет собой перечисление. Ниже перечислен ряд наиболее часто используемых его значений.

Значение	Описание
`DeclaredOnly`	Извлекаются только те методы, которые определены в заданном классе. Унаследованные методы в извлекаемые сведения не включаются
`Instance`	Извлекаются методы экземпляра
`NonPublic`	Извлекаются методы, не являющиеся открытыми
`Public`	Извлекаются открытые методы
`Static`	Извлекаются статические методы

Два или несколько флажков можно объединить с помощью логической операции ИЛИ. Но как минимум флажок `Instance` или `Static` следует указывать вместе с флажком `Public` или `NonPublic`. В противном случае не будут извлечены сведения ни об одном из методов.

Форма `BindingFlags` метода `GetMethods()` чаще всего применяется для получения списка методов, определенных в классе, без дополнительного извлечения наследуемых методов. Это особенно удобно в тех случаях, когда требуется исключить получение сведений о методах, определяемых в классе конкретного объекта. В качестве примера попробуем выполнить следующую замену в вызове метода `GetMethods()` из предыдущей программы.

```
// Теперь получаются сведения только о тех методах,
// которые объявлены в классе MyClass.
MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                               BindingFlags.Instance |
                               BindingFlags.Public);
```

После этой замены программа дает следующий результат.

```
Анализ методов, определенных в классе MyClass

Поддерживаемые методы:
   Int32 Sum()

   Boolean IsBetween(Int32 i)
```

```
Void Set(Int32 a, Int32 b)

Void Set(Double a, Double b)

Void Show()
```

Как видите, теперь выводятся только те методы, которые явно определены в классе `MyClass`.

Вызов методов с помощью рефлексии

Как только методы, поддерживаемые определенным типом данных, становятся известны, их можно вызывать. Для этой цели служит метод `Invoke()`, входящий в состав класса `MethodInfo`. Ниже приведена одна из форм этого метода:

```
object Invoke(object obj, object[] parameters)
```

где *obj* обозначает ссылку на объект, для которого вызывается метод. Для вызова статических методов (`static`) в качестве параметра *obj* передается пустое значение (`null`). Любые аргументы, которые должны быть переданы методу, указываются в массиве *parameters*. Если же аргументы не нужны, то вместо массива *parameters* указывается пустое значение (`null`). Кроме того, количество элементов массива *parameters* должно точно соответствовать количеству передаваемых аргументов. Так, если требуется передать два аргумента, то массив *parameters* должен состоять из двух элементов, но не из трех или четырех. Значение, возвращаемое вызываемым методом, передается методу `Invoke()`, который и возвращает его.

Для вызова конкретного метода достаточно вызвать метод `Invoke()` для экземпляра объекта типа `MethodInfo`, получаемого при вызове метода `GetMethods()`. Эта процедура демонстрируется в приведенном ниже примере программы.

```
// Вызвать методы с помощью рефлексии.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i, int j) {
    x = i;
    y = j;
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }
```

```
  public void Set(int a, int b) {
    Console.Write("В методе Set(int, int). ");
    x = a;
    y = b;
    Show();
  }

  // Перегрузить метод Set.
  public void Set(double a, double b) {
    Console.Write("В методе Set(double, double). ");
    x = (int) a;
    y = (int) b;
    Show();
  }

  public void Show() {
    Console.WriteLine("Значение x: {0}, значение y: {1}", x, y);
  }
}

class InvokeMethDemo {
  static void Main() {
    Type t = typeof(MyClass);
    MyClass reflectOb = new MyClass(10, 20);
    int val;

    Console.WriteLine("Вызов методов, определенных в классе " + t.Name);
    Console.WriteLine();
    MethodInfo[] mi = t.GetMethods();

    // Вызвать каждый метод.
    foreach(MethodInfo m in mi) {
      // Получить параметры.
      ParameterInfo[] pi = m.GetParameters();

      if(m.Name.CompareTo("Set")==0 &&
             pi[0].ParameterType == typeof(int)) {
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Set")==0 &&
              pi[0].ParameterType == typeof(double)) {
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Sum")==0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("Сумма равна " + val);
       }
       else if(m.Name.CompareTo("IsBetween")==0) {
```

```
          object[] args = new object[1];
          args[0] = 14;
          if((bool) m.Invoke(reflectOb, args))
            Console.WriteLine("Значение 14 находится между x и y");
        }
        else if(m.Name.CompareTo("Show")==0) {
          m.Invoke(reflectOb, null);
        }
      }
    }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Вызов методов, определенных в классе MyClass

Сумма равна 30
Значение 14 находится между x и y
В методе Set(int, int). Значение x: 9, значение y: 18
В методе Set(double, double). Значение x: 1, значение y: 23
Значение x: 1, значение y: 23
```

Рассмотрим подробнее порядок вызова методов. Сначала создается список методов. Затем в цикле `foreach` извлекаются сведения об их параметрах. Далее каждый метод вызывается с указанием соответствующего типа и числа аргументов в последовательном ряде условных операторов `if/else`. Обратите особое внимание на перегрузку метода `Set()` в приведенном ниже фрагменте кода.

```
if(m.Name.CompareTo("Set")==0 &&
     pi[0].ParameterType == typeof(int)) {
  object[] args = new object[2];
  args[0] = 9;
  args[1] = 18;
  m.Invoke(reflectOb, args);
}
else if(m.Name.CompareTo("Set")==0 &&
        pi[0].ParameterType == typeof(double)) {
  object[] args = new object[2];
  args[0] = 1.12;
  args[1] = 23.4;
  m.Invoke(reflectOb, args);
}
```

Если имя метода — `Set`, то проверяется тип первого параметра, чтобы выявить конкретный вариант этого метода. Так, если это метод `Set(int, int)`, то его аргументы загружаются в массив `args`. В противном случае используются аргументы типа `double`.

Получение конструкторов конкретного типа

В предыдущем примере при вызове методов, определенных в классе `MyClass`, преимущества рефлексии не использовались, поскольку объект типа `MyClass` создавался явным образом. В таком случае было бы намного проще вызвать для него методы обычным образом. Но сильные стороны рефлексии проявляются наиболее заметно лишь в том случае, если объект создается динамически во время выполнения. И для

этого необходимо получить сначала список конструкторов, а затем экземпляр объекта заданного типа, вызвав один из этих конструкторов. Такой механизм позволяет получать во время выполнения экземпляр объекта любого типа, даже не указывая его имя в операторе объявления.

Конструкторы конкретного типа получаются при вызове метода `GetConstructors()` для объекта класса `Type`. Ниже приведена одна из наиболее часто используемых форм этого метода.

```
ConstructorInfo[] GetConstructors()
```

Метод `GetConstructors()` возвращает массив объектов класса `ConstructorInfo`, описывающих конструкторы.

Класс `ConstructorInfo` является производным от абстрактного класса `MethodBase`, который в свою очередь наследует от класса `MemberInfo`. В нем также определен ряд собственных методов. К их числу относится интересующий нас метод `GetConstructors()`, возвращающий список параметров, связанных с конструктором. Этот метод действует таким же образом, как и упоминавшийся ранее метод `GetParameters()`, определенный в классе `MethodInfo`.

Как только будет обнаружен подходящий конструктор, для создания объекта вызывается метод `Invoke()`, определенный в классе `ConstructorInfo`. Ниже приведена одна из форм этого метода.

```
object Invoke(object[] parameters)
```

Любые аргументы, которые требуется передать методу, указываются в массиве *parameters*. Если же аргументы не нужны, то вместо массива *parameters* указывается пустое значение (`null`). Но в любом случае количество элементов массива *parameters* должно совпадать с количеством передаваемых аргументов, а типы аргументов — с типами параметров. Метод `Invoke()` возвращает ссылку на сконструированный объект.

В приведенном ниже примере программы рефлексия используется для создания экземпляра объекта класса `MyClass`.

```
// Создать объект с помощью рефлексии.

using System;
using System.Reflection;

class MyClass {
  int x;
  int y;

  public MyClass(int i) {
    Console.WriteLine("Конструирование класса MyClass(int, int). ");
    x = y = i;
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Конструирование класса MyClass(int, int). ");
    x = i;
    y = j;
    Show();
  }
```

```
  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void Set(int a, int b) {
    Console.Write("В методе Set(int, int). ");
    x = a;
    y = b;
    Show();
  }

  // Перегрузить метод Set.
  public void Set(double a, double b) {
    Console.Write("В методе(double, double). ");
    x = (int) a;
    y = (int) b;
    Show();
  }

  public void Show() {
    Console.WriteLine("Значение x: {0}, значение y: {1}", x, y);
  }
}

class InvokeConsDemo {
  static void Main() {
    Type t = typeof(MyClass);
    int val;

    // Получить сведения о конструкторе.
    ConstructorInfo[] ci = t.GetConstructors();

    Console.WriteLine("Доступные конструкторы: ");
    foreach(ConstructorInfo c in ci) {
      // Вывести возвращаемый тип и имя.
      Console.Write(" " + t.Name + "(");

      // Вывести параметры.
      ParameterInfo[] pi = c.GetParameters();

      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name + " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }

      Console.WriteLine(")");

    }
    Console.WriteLine();
```

```
// Найти подходящий конструктор.
int x;
for(x=0; x < ci.Length; x++) {
  ParameterInfo[] pi = ci[x].GetParameters();
  if(pi.Length == 2) break;
}

if(x == ci.Length) {
  Console.WriteLine("Подходящий конструктор не найден.");
  return;
}
else
  Console.WriteLine("Найден конструктор с двумя параметрами.\n");

// Сконструировать объект.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);

Console.WriteLine("\nВызов методов для объекта reflectOb.");
Console.WriteLine();
MethodInfo[] mi = t.GetMethods();

// Вызвать каждый метод.
foreach(MethodInfo m in mi) {
  // Получить параметры.
  ParameterInfo[] pi = m.GetParameters();
  if(m.Name.CompareTo("Set")==0 &&
       pi[0].ParameterType == typeof(int)) {
    // Это метод Set(int, int).
    object[] args = new object[2];
    args[0] = 9;
    args[1] = 18;
    m.Invoke(reflectOb, args);
  }
  else if(m.Name.CompareTo("Set")==0 &&
          pi[0].ParameterType == typeof(double)) {
    // Это метод Set(double, double).
    object[] args = new object[2];
    args[0] = 1.12;
    args[1] = 23.4;
    m.Invoke(reflectOb, args);
  }
  else if(m.Name.CompareTo("Sum")==0) {
    val = (int) m.Invoke(reflectOb, null);
    Console.WriteLine("Сумма равна " + val);
  }
  else if(m.Name.CompareTo("IsBetween")==0) {
    object[] args = new object[1];
    args[0] = 14;
    if((bool) m.Invoke(reflectOb, args))
      Console.WriteLine("Значение 14 находится между x и y");
  }
```

```
      else if(m.Name.CompareTo("Show")==0) {
        m.Invoke(reflectOb, null);
      }
    }
  }
}
```

Эта программа дает следующий результат.

```
Доступные конструкторы:
   MyClass(Int32 i)
   MyClass(Int32 i, Int32 j)

Найден конструктор с двумя параметрами.

Конструирование класса MyClass(int, int)
Значение x: 10, значение y: 20

Вызов методов для объекта reflectOb

Сумма равна 30
Значение 14 находится между x и y
В методе Set(int, int). Значение x: 9, значение y: 18
В методе Set(double, double). Значение x: 1, значение y: 23
Значение x: 1, значение y: 23
```

А теперь рассмотрим порядок применения рефлексии для конструирования объекта класса `MyClass`. Сначала получается перечень открытых конструкторов в следующей строке кода.

```
ConstructorInfo[] ci = t.GetConstructors();
```

Затем для наглядности примера выводятся полученные конструкторы. После этого осуществляется поиск по списку конструктора, принимающего два аргумента, как показано в приведенном ниже фрагменте кода.

```
for(x=0; x < ci.Length; x++) {
  ParameterInfo[] pi = ci[x].GetParameters();
  if(pi.Length == 2) break;
}
```

Если такой конструктор найден, как в данном примере, то в следующем фрагменте кода получается экземпляр объекта заданного типа.

```
// Сконструировать объект.
object[] consargs = new object[2];
consargs[0] = 10;
consargs[1] = 20;
object reflectOb = ci[x].Invoke(consargs);
```

После вызова метода `Invoke()` переменная экземпляра `reflectOb` будет ссылаться на объект типа `MyClass`. А далее в программе выполняются соответствующие методы для экземпляра этого объекта.

Следует, однако, иметь в виду, что ради простоты в данном примере предполагается наличие лишь одного конструктора с двумя аргументами типа `int`. Очевидно, что в реальном коде придется дополнительно проверять соответствие типов каждого параметра и аргумента.

Получение типов данных из сборок

В предыдущем примере все сведения о классе `MyClass` были получены с помощью рефлексии, за исключением одного элемента: типа самого класса `MyClass`. Несмотря на то что сведения о классе получались в предыдущем примере динамически, этот пример опирался на тот факт, что имя типа `MyClass` было известно заранее и использовалось в операторе `typeof` для получения объекта класса `Type`, по отношению к которому осуществлялось косвенное или непосредственное обращение к методам рефлексии. В некоторых случаях такой подход может оказаться вполне пригодным, но истинные преимущества рефлексии проявляются лишь тогда, когда доступные в программе типы данных определяются динамически в результате анализа содержимого других сборок.

Как следует из главы 16, сборка несет в себе сведения о типах классов, структур и прочих элементов данных, которые в ней содержатся. Прикладной интерфейс Reflection API позволяет загрузить сборку, извлечь сведения о ней и получить экземпляры объектов любых открыто доступных в ней типов. Используя этот механизм, программа может выявлять свою среду и использовать те функциональные возможности, которые могут оказаться доступными без явного их определения во время компиляции. Это очень эффективный и привлекательный принцип. Представьте себе, например, программу, которая выполняет роль "браузера типов", отображая типы данных, доступные в системе, или же инструментальное средство разработки, позволяющее визуально составлять программы из различных типов данных, поддерживаемых в системе. А поскольку все сведения о типах могут быть извлечены и проверены, то ограничений на применение рефлексии практически не существует.

Для получения сведений о сборке сначала необходимо создать объект класса `Assembly`. В классе `Assembly` открытый конструктор не определяется. Вместо этого объект класса `Assembly` получается в результате вызова одного из его методов. Так, для загрузки сборки по заданному ее имени служит метод `LoadFrom()`. Ниже приведена его соответствующая форма:

```
static Assembly LoadFrom(string файл_сборки)
```

где *файл_сборки* обозначает конкретное имя файла сборки.

Как только будет получен объект класса `Assembly`, появится возможность обнаружить определенные в нем типы данных, вызвав для него метод `GetTypes()` в приведенной ниже общей форме.

```
Type[] GetTypes()
```

Этот метод возвращает массив типов, содержащихся в сборке.

Для того чтобы продемонстрировать порядок обнаружения типов в сборке, потребуются два исходных файла. Первый файл будет содержать ряд классов, обнаруживаемых в коде из второго файла. Создадим сначала файл `MyClasses.cs`, содержащий следующий код.

```
// Файл, содержащий три класса и носящий имя MyClasses.cs.

using System;

class MyClass {
  int x;
  int y;
```

```
  public MyClass(int i) {
    Console.WriteLine("Конструирование класса MyClass(int). ");
    x = y = i;
    Show();
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Конструирование класса MyClass(int, int). ");
    x = i;
    y = j;
    Show();
  }

  public int Sum() {
    return x+y;
  }

  public bool IsBetween(int i) {
    if((x < i) && (i < y)) return true;
    else return false;
  }

  public void Set(int a, int b) {
    Console.Write("В методе Set(int, int). ");
    x = a;
    y = b;
    Show();
  }

  // Перегрузить метод Set.
  public void Set(double a, double b) {
    Console.Write("В методе Set(double, double). ");
    x = (int) a;
    y = (int) b;
    Show();
  }

  public void Show() {
    Console.WriteLine("Значение x: {0}, значение y: {1}", x, y);
  }
}

class AnotherClass {
  string msg;
  public AnotherClass(string str) {
    msg = str;
  }

  public void Show() {
    Console.WriteLine(msg);
  }
}
```

```
class Demo {
  static void Main() {
    Console.WriteLine("Это заполнитель.");
  }
}
```

Этот файл содержит класс `MyClass`, неоднократно использовавшийся в предыдущих примерах. Кроме того, в файл добавлены второй класс `AnotherClass` и третий класс `Demo`. Следовательно, сборка, полученная из исходного кода, находящегося в этом исходном файле, будет содержать три класса. Затем этот файл компилируется, и из него формируется исполняемый файл `MyClasses.exe`. Именно эта сборка и будет опрашиваться программно.

Ниже приведена программа, в которой будут извлекаться сведения о файле сборки `MyClasses.exe`. Ее исходный текст составляет содержимое второго файла.

```
/* Обнаружить сборку, определить типы и создать объект
   с помощью рефлексии. */

using System;
using System.Reflection;

class ReflectAssemblyDemo {
  static void Main() {
    int val;

    // Загрузить сборку MyClasses.exe.
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    // Обнаружить типы, содержащиеся в сборке MyClasses.exe.
    Type[] alltypes = asm.GetTypes();
    foreach(Type temp in alltypes)
      Console.WriteLine("Найдено: " + temp.Name);

    Console.WriteLine();

    // Использовать первый тип, в данном случае — класс MyClass.
    Type t = alltypes[0]; // использовать первый найденный класс
    Console.WriteLine("Использовано: " + t.Name);

    // Получить сведения о конструкторе.
    ConstructorInfo[] ci = t.GetConstructors();

    Console.WriteLine("Доступные конструкторы: ");
    foreach(ConstructorInfo c in ci) {
      // Вывести возвращаемый тип и имя.
      Console.Write(" " + t.Name + "(");

      // Вывести параметры.
      ParameterInfo[] pi = c.GetParameters();
      for(int i=0; i < pi.Length; i++) {
        Console.Write(pi[i].ParameterType.Name + " " + pi[i].Name);
        if(i+1 < pi.Length) Console.Write(", ");
      }
```

```
      Console.WriteLine(")");
    }
    Console.WriteLine();

    // Найти подходящий конструктор.
    int x;
    for(x=0; x < ci.Length; x++) {
      ParameterInfo[] pi = ci[x].GetParameters();
      if(pi.Length == 2) break;
    }
    if(x == ci.Length) {
      Console.WriteLine("Подходящий конструктор не найден.");
      return;
    }
    else
      Console.WriteLine("Найден конструктор с двумя параметрами.\n");

    // Сконструировать объект.
    object[] consargs = new object[2];
    consargs[0] = 10;
    consargs[1] = 20;
    object reflectOb = ci[x].Invoke(consargs);

    Console.WriteLine("\nВызов методов для объекта reflectOb.");
    Console.WriteLine();
    MethodInfo[] mi = t.GetMethods();

    // Вызвать каждый метод.
    foreach(MethodInfo m in mi) {
      // Получить параметры.
      ParameterInfo[] pi = m.GetParameters();

      if(m.Name.CompareTo("Set")==0 &&
           pi[0].ParameterType == typeof(int)) {
        // Это метод Set(int, int).
        object[] args = new object[2];
        args[0] = 9;
        args[1] = 18;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Set")==0 &&
              pi[0].ParameterType == typeof(double)) {
        // Это метод Set(double, double).
        object[] args = new object[2];
        args[0] = 1.12;
        args[1] = 23.4;
        m.Invoke(reflectOb, args);
      }
      else if(m.Name.CompareTo("Sum")==0) {
        val = (int) m.Invoke(reflectOb, null);
        Console.WriteLine("Сумма равна " + val);
      }
      else if(m.Name.CompareTo("IsBetween")==0) {
        object[] args = new object[1];
```

```
        args[0] = 14;
        if((bool) m.Invoke(reflectOb, args))
        Console.WriteLine("Значение 14 находится между x и y");
      }
      else if(m.Name.CompareTo("Show")==0) {
       m.Invoke(reflectOb, null);
      }
    }
  }
}
```

При выполнении этой программы получается следующий результат.

```
Найдено: MyClass
Найдено: AnotherClass
Найдено: Demo

Использовано: MyClass

Доступные конструкторы:
   MyClass(Int32 i)
   MyClass(Int32 i, Int32 j)

Найден конструктор с двумя параметрами.

Конструирование класса MyClass(int, int)
Значение x: 10, значение y: 20

Вызов методов для объекта reflectOb

Сумма равна 30
Значение 14 находится между x и y
В методе Set(int, int). Значение x: 9, значение y: 18
В методе Set(double, double). Значение x: 1, значение y: 23
Значение x: 1, значение y: 23
```

Как следует из результата выполнения приведенной выше программы, обнаружены все три класса, содержащиеся в файле сборки `MyClasses.exe`. Первым среди них обнаружен класс `MyClass`, который затем был использован для получения экземпляра объекта и вызова соответствующих методов.

Отдельные типы обнаруживаются в сборке `MyClasses.exe` с помощью приведенной ниже последовательности кода, находящегося в самом начале метода `Main()`.

```
// Загрузить сборку MyClasses.exe.
Assembly asm = Assembly.LoadFrom("MyClasses.exe");

// Обнаружить типы, содержащиеся в сборке MyClasses.exe.
Type[] alltypes = asm.GetTypes();
foreach(Type temp in alltypes)
  Console.WriteLine("Найдено: " + temp.Name);
```

Этой последовательностью кода можно пользоваться всякий раз, когда требуется динамически загружать и опрашивать сборку.

Но сборка совсем не обязательно должна быть исполняемым файлом с расширением `.exe`. Сборки могут быть также в файлах динамически компонуемых библиотек

(DLL) с расширением .dll. Так, если скомпилировать исходный файл MyClasses.cs в следующей командной строке:

```
csc /t:library MyClasses.cs
```

то в итоге получится файл MyClasses.dll. Преимущество размещения кода в библиотеке DLL заключается, в частности, в том, что в этом случае метод Main() в исходном коде не нужен, тогда как всем исполняемым файлам требуется определенная точка входа, с которой должно начинаться выполнение программы. Именно поэтому класс Demo содержит метод Main() в качестве такой точки входа. А для библиотеки DLL метод Main() не требуется. Если же класс MyClass нужно превратить в библиотеку DLL, то в вызов метода LoadFrom() придется внести следующее изменение.

```
Assembly asm = Assembly.LoadFrom("MyClasses.dll");
```

Полностью автоматизированное обнаружение типов

Прежде чем завершить рассмотрение рефлексии, обратимся к еще одному поучительному примеру. Несмотря на то что в программе из предыдущего примера класс MyClass был полноценно использован без явного указания на его имя в программе, этот пример все же опирается на предварительную осведомленность о содержимом класса MyClass. Так, в программе были заранее известны имена методов Set и Sum из этого класса. Но с помощью рефлексии можно воспользоваться типом данных, ничего не зная о нем заранее. С этой целью придется извлечь все сведения, необходимые для конструирования объекта и формирования вызовов соответствующих методов. Такой подход может оказаться пригодным, например, при создании инструментального средства визуального проектирования, поскольку он позволяет использовать типы данных, имеющиеся в системе.

Рассмотрим следующий пример, демонстрирующий полностью автоматизированное обнаружение типов. В этом примере сначала загружается сборка MyClasses.exe, затем конструируется объект класса MyClass и далее вызываются все методы, объявленные в классе MyClass, причем о них ничего заранее неизвестно.

```
// Использовать класс MyClass, ничего не зная о нем заранее.

using System;
using System.Reflection;

class ReflectAssemblyDemo {
  static void Main() {
    int val;
    Assembly asm = Assembly.LoadFrom("MyClasses.exe");

    Type[] alltypes = asm.GetTypes();

    Type t = alltypes[0]; // использовать первый обнаруженный класс
    Console.WriteLine("Использовано: " + t.Name);

    ConstructorInfo[] ci = t.GetConstructors();

    // Использовать первый обнаруженный конструктор.
    ParameterInfo[] cpi = ci[0].GetParameters();
    object reflectOb;
```

```
if(cpi.Length > 0) {
  object[] consargs = new object[cpi.Length];

  // Инициализировать аргументы.
  for(int n=0; n < cpi.Length; n++)
    consargs[n] = 10 + n * 20;

  // Сконструировать объект.
  reflectOb = ci[0].Invoke(consargs);
} else
  reflectOb = ci[0].Invoke(null);

Console.WriteLine("\nВызов методов для объекта reflectOb.");
Console.WriteLine();

// Игнорировать наследуемые методы.
MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly |
                               BindingFlags.Instance |
                               BindingFlags.Public);

// Вызвать каждый метод.
foreach(MethodInfo m in mi) {
 Console.WriteLine("Вызов метода {0} ", m.Name);

// Получить параметры.
ParameterInfo[] pi = m.GetParameters();

// Выполнить методы.
switch(pi.Length) {
  case 0: // аргументы отсутствуют
    if(m.ReturnType == typeof(int)) {
      val = (int) m.Invoke(reflectOb, null);
      Console.WriteLine("Результат: " + val);
    }
    else if(m.ReturnType == typeof(void)) {
      m.Invoke(reflectOb, null);
    }
    break;
  case 1: // один аргумент
    if(pi[0].ParameterType == typeof(int)) {
      object[] args = new object[1];
      args[0] = 14;
      if((bool) m.Invoke(reflectOb, args))
        Console.WriteLine("Значение 14 находится между x и y");
      else
       Console.WriteLine("Значение 14 не находится между x и y");
    }
    break;
  case 2: // два аргумента
    if((pi[0].ParameterType == typeof(int)) &&
       (pi[1].ParameterType == typeof(int))) {
      object[] args = new object[2];
      args[0] = 9;
```

```
          args[1] = 18;
          m.Invoke(reflectOb, args);
        }
        else if((pi[0].ParameterType == typeof(double)) &&
                (pi[1].ParameterType == typeof(double))) {
          object[] args = new object[2];
          args[0] = 1.12;
          args[1] = 23.4;
          m.Invoke(reflectOb, args);
        }
        break;
      }
      Console.WriteLine();
    }
  }
}
```

Эта программа дает следующий результат.

```
Использовано: MyClass
Конструирование класса MyClass(int).
Значение x: 10, значение y: 10

Вызов методов для объекта reflectOb.

Вызов метода Sum
Результат: 20

Вызов метода IsBetween
Значение 14 не находится между x и y

Вызов метода Set
В методе Set(int, int). Значение x: 9, значение y: 18

Вызов метода Set
В методе Set(double, double). Значение x: 1, значение y: 23

Вызов метода Show
Значение x: 1, значение y: 23
```

Эта программа работает довольно просто, но все же требует некоторых пояснений. Во-первых, получаются и используются только те методы, которые явно объявлены в классе `MyClass`. Для этой цели служит форма `BindingFlags` метода `GetMethods()`, чтобы воспрепятствовать вызову методов, наследуемых от объекта. И во-вторых, количество параметров и возвращаемый тип каждого метода получаются динамически, а затем определяются и проверяются в операторе `switch`. На основании этой информации формируется вызов каждого метода.

Атрибуты

В C# разрешается вводить в программу информацию декларативного характера в форме *атрибута*, с помощью которого определяются дополнительные сведения (метаданные), связанные с классом, структурой, методом и т.д. Например, в програм-

ме можно указать атрибут, определяющий тип кнопки, которую должен отображать конкретный класс. Атрибуты указываются в квадратных скобках перед тем элементом, к которому они применяются. Следовательно, атрибут не является членом класса, но обозначает дополнительную информацию, присоединяемую к элементу.

Основы применения атрибутов

Атрибут поддерживается классом, наследующим от класса `System.Attribute`. Поэтому классы атрибутов должны быть подклассами класса `Attribute`. В классе `Attribute` определены основные функциональные возможности, но далеко не все они нужны для работы с атрибутами. В именах классов атрибутов принято употреблять суффикс `Attribute`. Например, `ErrorAttribute` — это имя класса атрибута, описывающего ошибку.

При объявлении класса атрибута перед его именем указывается атрибут `AttributeUsage`. Этот встроенный атрибут обозначает типы элементов, к которым может применяться объявляемый атрибут. Так, применение атрибута может ограничиваться одними методами.

Создание атрибута

В классе атрибута определяются члены, поддерживающие атрибут. Классы атрибутов зачастую оказываются довольно простыми и содержат небольшое количество полей или свойств. Например, атрибут может определять примечание, описывающее элемент, к которому присоединяется атрибут. Такой атрибут может принимать следующий вид.

```
  [AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // базовое поле свойства Remark

  public RemarkAttribute(string comment) {
    pri_remark = comment;
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}
```

Проанализируем этот класс атрибута построчно.

Объявляемый атрибут получает имя `RemarkAttribute`. Его объявлению предшествует встроенный атрибут `AttributeUsage`, указывающий на то, что атрибут `RemarkAttribute` может применяться ко всем типам элементов. С помощью встроенного атрибута `AttributeUsage` можно сузить перечень элементов, к которым может присоединяться объявляемый атрибут. Подробнее о его возможностях речь пойдет далее в этой главе.

Далее объявляется класс `RemarkAttribute`, наследующий от класса `Attribute`. В классе `RemarkAttribute` определяется единственное закрытое поле `pri_remark`, поддерживающее одно открытое и доступное для чтения свойство `Remark`. Это свойство содержит описание, связываемое с атрибутом. (Конечно, `Remark` можно было

бы объявить как автоматически реализуемое свойство с закрытым аксессором set, но ради наглядности данного примера выбрано свойство, доступное только для чтения.) В данном классе определен также один открытый конструктор, принимающий строковый аргумент и присваивающий его свойству Remark. Этим пока что ограничиваются функциональные возможности класса RemarkAttribute, готового к применению.

Присоединение атрибута

Как только класс атрибута будет определен, атрибут можно присоединить к элементу. Атрибут указывается перед тем элементом, к которому он присоединяется, и для этого его конструктор заключается в квадратные скобки. В качестве примера ниже показано, как атрибут RemarkAttribute связывается с классом.

```
[RemarkAttribute("В этом классе используется атрибут.")]
class UseAttrib {
  // ...
}
```

В этом фрагменте кода конструируется атрибут RemarkAttribute, содержащий комментарий "В этом классе используется атрибут." Данный атрибут затем связывается с классом UseAttrib.

Присоединяя атрибут, совсем не обязательно указывать суффикс Attribute. Например, приведенный выше класс может быть объявлен следующим образом.

```
[Remark("В этом классе используется атрибут.")]
class UseAttrib {
  // ...
}
```

В этом объявлении указывается только имя Remark. Такая сокращенная форма считается вполне допустимой, но все же надежнее указывать полное имя присоединяемого атрибута, чтобы избежать возможной путаницы и неоднозначности.

Получение атрибутов объекта

Как только атрибут будет присоединен к элементу, он может быть извлечен в других частях программы. Для извлечения атрибута обычно используется один из двух методов. Первый метод, GetCustomAttributes(), определяется в классе MemberInfo и наследуется классом Type. Он извлекает список всех атрибутов, присоединенных к элементу. Ниже приведена одна из его форм.

```
object[] GetCustomAttributes(bool наследование)
```

Если *наследование* имеет логическое значение true, то в список включаются атрибуты всех базовых классов, наследуемых по иерархической цепочке. В противном случае атрибуты извлекаются только из тех классов, которые определяются указанным типом.

Второй метод, GetCustomAttribute(), определяется в классе Attribute. Ниже приведена одна из его форм:

```
static Attribute GetCustomAttribute(MemberInfo элемент, Type тип_атрибута)
```

где *элемент* обозначает объект класса MemberInfo, описывающий тот элемент, для которого создаются атрибуты, тогда как *тип_атрибута* — требуемый атрибут. Данный метод используется в том случае, если имя получаемого атрибута известно заранее, что

зачастую и бывает. Так, если в классе `UseAttrib` имеется атрибут `RemarkAttribute`, то для получения ссылки на этот атрибут можно воспользоваться следующей последовательностью кода.

```
// Получить экземпляр объекта класса MemberInfo, связанного
// с классом, содержащим атрибут RemarkAttribute.
Type t = typeof(UseAttrib);

// Извлечь атрибут RemarkAttribute.
Type tRemAtt = typeof(RemarkAttribute);
RemarkAttribute ra = (RemarkAttribute)
      Attribute.GetCustomAttribute(t, tRemAtt);
```

Эта последовательность кода оказывается вполне работоспособной, поскольку класс `MemberInfo` является базовым для класса `Type`. Следовательно, `t` — это экземпляр объекта класса `MemberInfo`.

Имея ссылку на атрибут, можно получить доступ к его членам. Благодаря этому информация об атрибуте становится доступной для программы, использующей элемент, к которому присоединен атрибут. Например, в следующей строке кода выводится содержимое свойства `Remark`.

```
Console.WriteLine(ra.Remark);
```

Ниже приведена программа, в которой все изложенные выше особенности применения атрибутов демонстрируются на примере атрибута `RemarkAttribute`.

```
// Простой пример применения атрибута.

using System;
using System.Reflection;

  [AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // базовое поле свойства Remark

  public RemarkAttribute(string comment) {
    pri_remark = comment;
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}

[RemarkAttribute("В этом классе используется атрибут.")]
class UseAttrib {
  // ...
}

class AttribDemo {
  static void Main() {
     Type t = typeof(UseAttrib);

     Console.Write("Атрибуты в классе " + t.Name + ": ");
```

```
    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    Console.Write("Примечание: ");

    // Извлечь атрибут RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.WriteLine(ra.Remark);
  }
}
```

Эта программа дает следующий результат.

```
Атрибуты в классе UseAttrib: RemarkAttribute
Примечание: В этом классе используется атрибут.
```

Сравнение позиционных и именованных параметров

В предыдущем примере для инициализации атрибута `RemarkAttribute` его конструктору была передана символьная строка с помощью обычного синтаксиса конструктора. В этом случае параметр `comment` конструктора `RemarkAttribute()` называется *позиционным*. Этот термин отражает тот факт, что аргумент связан с параметром по его позиции в списке аргументов. Следовательно, первый аргумент передается первому параметру, второй аргумент — второму параметру и т.д.

Но для атрибута доступны также *именованные параметры*, которым можно присваивать первоначальные значения по их именам. В этом случае значение имеет имя, а не позиция параметра.

ПРИМЕЧАНИЕ

Несмотря на то что именованные параметры атрибутов, по существу, подобны именованным аргументам методов, они все же отличаются в деталях.

Именованный параметр поддерживается открытым полем или свойством, которое должно быть нестатическим и доступным только для записи. Любое поле или свойство подобного рода может автоматически использоваться в качестве именованного параметра. Значение присваивается именованному параметру с помощью соответствующего оператора, расположенного в списке аргументов при вызове конструктора атрибута. Ниже приведена общая форма объявления атрибута, включая именованные параметры.

```
[attrib(список_позиционных_параметров,
        именованный_параметр_1 = значение,
        именованный_параметр_2 = значение, ...)]
```

Первыми указываются позиционные параметры, если они существуют. Далее следуют именованные параметры с присваиваемыми значениями. Порядок следования

именованных параметров особого значения не имеет. Именованным параметрам не обязательно присваивать значение, и в этом случае используется значение, устанавливаемое по умолчанию.

Применение именованного параметра лучше всего показать на конкретном примере. Ниже приведен вариант класса RemarkAttribute, в который добавлено поле Supplement, предназначенное для хранения дополнительного примечания.

```
[AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // базовое поле свойства Remark

  // Это поле можно использовать в качестве именованного параметра.
  public string Supplement;

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    Supplement = "Отсутствует";
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}
```

Как видите, поле Supplement инициализируется в конструкторе символьной строкой "Отсутствует". Другого способа присвоить ему первоначальное значение в конструкторе не существует. Но поскольку поле Supplement является открытым в классе RemarkAttribute, его можно использовать в качестве именованного параметра, как показано ниже.

```
[RemarkAttribute("В этом классе используется атрибут.",
        Supplement = "Это дополнительная информация.")]
class UseAttrib {
  // ...
}
```

Обратите особое внимание на вызов конструктора класса RemarkAttribute. В этом конструкторе первым, как и прежде, указывается позиционный параметр, а за ним через запятую следует именованный параметр Supplement, которому присваивается конкретное значение. И наконец, закрывающая скобка,), завершает вызов конструктора. Таким образом, именованный параметр инициализируется в вызове конструктора. Этот синтаксис можно обобщить: позиционные параметры должны указываться в том порядке, в каком они определены в конструкторе, а именованные параметры — в произвольном порядке и вместе с присваиваемыми им значениями.

Ниже приведена программа, в которой демонстрируется применение поля Supplement в качестве именованного параметра атрибута.

```
// Использовать именованный параметр атрибута.

using System;
using System.Reflection;
```

```
  [AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // базовое поле свойства Remark

  public string Supplement; // это именованный параметр

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    Supplement = "Отсутствует";
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }
}

[RemarkAttribute("В этом классе используется атрибут.",
        Supplement = "Это дополнительная информация.")]
class UseAttrib {
  // ...
}

class NamedParamDemo {
  static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Атрибуты в классе " + t.Name + ": ");
    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }

    // Извлечь атрибут RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.Write("Примечание: ");
    Console.WriteLine(ra.Remark);

    Console.Write("Дополнение: ");
    Console.WriteLine(ra.Supplement);
  }
}
```

При выполнении этой программы получается следующий результат.

```
Атрибуты в классе UseAttrib: RemarkAttribute
Примечание: В этом классе используется атрибут.
Дополнение: Это дополнительная информация.
```

Прежде чем перейти к следующему вопросу, следует особо подчеркнуть, что поле `pri_remark` *нельзя* использовать в качестве именованного параметра, поскольку оно

закрыто в классе RemarkAttribute. Свойство Remark также *нельзя* использовать в качестве именованного параметра, потому что оно доступно только для чтения. Напомним, что в качестве именованных параметров могут служить только открытые поля и свойства.

Открытое и доступное только для чтения свойство может использоваться в качестве именованного параметра таким же образом, как и открытое поле. В качестве примера ниже показано, как автоматически реализуемое свойство Priority типа int вводится в класс RemarkAttribute.

```
// Использовать свойство в качестве именованного параметра атрибута.

using System;
using System.Reflection;

  [AttributeUsage(AttributeTargets.All)]
public class RemarkAttribute : Attribute {
  string pri_remark; // базовое поле свойства Remark

  public string Supplement; // это именованный параметр

  public RemarkAttribute(string comment) {
    pri_remark = comment;
    Supplement = "Отсутствует";
    Priority = 1;
  }

  public string Remark {
    get {
      return pri_remark;
    }
  }

  // Использовать свойство в качестве именованного параметра.
  public int Priority { get; set; }
}

[RemarkAttribute("В этом классе используется атрибут.",
        Supplement = " Это дополнительная информация.",
        Priority = 10)]
class UseAttrib {
  // ...
}

class NamedParamDemo {
  static void Main() {
    Type t = typeof(UseAttrib);

    Console.Write("Атрибуты в классе " + t.Name + ": ");

    object[] attribs = t.GetCustomAttributes(false);
    foreach(object o in attribs) {
      Console.WriteLine(o);
    }
```

```
    // Извлечь атрибут RemarkAttribute.
    Type tRemAtt = typeof(RemarkAttribute);
    RemarkAttribute ra = (RemarkAttribute)
          Attribute.GetCustomAttribute(t, tRemAtt);

    Console.Write("Примечание: ");
    Console.WriteLine(ra.Remark);

    Console.Write("Дополнение: ");
    Console.WriteLine(ra.Supplement);

    Console.WriteLine("Приоритет: " + ra.Priority);
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Атрибуты в классе UseAttrib: RemarkAttribute
Примечание: В этом классе используется атрибут.
Дополнение: Это дополнительная информация.
Приоритет: 10
```

В данном примере обращает на себя внимание порядок указания атрибутов перед классом `UseAttrib`, как показано ниже.

```
[RemarkAttribute("В этом классе используется атрибут.",
        Supplement = " Это дополнительная информация.",
        Priority = 10)]
class UseAttrib {
  // ...
}
```

Именованные параметры атрибутов `Supplement` и `Priority` *не* обязательно указывать в каком-то определенном порядке. Порядок их указания можно свободно изменить, не меняя сами атрибуты.

И последнее замечание: тип параметра атрибута (как позиционного, так и именованного) должен быть одним из встроенных простых типов, `object`, `Type`, перечислением или одномерным массивом одного из этих типов.

Встроенные атрибуты

В C# предусмотрено несколько встроенных атрибутов, но три из них имеют особое значение, поскольку они применяются в самых разных ситуациях. Это атрибуты `AttributeUsage`, `Conditional` и `Obsolete`, рассматриваемые далее по порядку.

Атрибут `AttributeUsage`

Как упоминалось ранее, атрибут `AttributeUsage` определяет типы элементов, к которым может быть применен объявляемый атрибут. `AttributeUsage` — это, по существу, еще одно наименование класса `System.AttributeUsageAttribute`. У него имеется следующий конструктор:

```
AttributeUsage(AttributeTargets validOn)
```

где *validOn* обозначает один или несколько элементов, к которым может быть применен объявляемый атрибут, тогда как `AttributeTargets` — перечисление, в котором определяются приведенные ниже значения.

`All`	`Assembly`	`Class`	`Constructor`
`Delegate`	`Enum`	`Event`	`Field`
`GenericParameter`	`Interface`	`Method`	`Module`
`Parameter`	`Property`	`ReturnValue`	`Struct`

Два этих значения или более можно объединить с помощью логической операции ИЛИ. Например, для указания атрибута, применяемого только к полям и свойствам, используются следующие значения.

```
AttributeTargets.Field | AttributeTargets.Property
```

В классе атрибута `AttributeUsage` поддерживаются два именованных параметра. Первым из них является параметр `AllowMultiple`, принимающий логическое значение. Если это значение истинно, то атрибут может быть применен к одному и тому же элементу неоднократно. Второй именованный параметр, `Inherited`, также принимает логическое значение. Если это значение истинно, то атрибут наследуется производными классами, а иначе он не наследуется. По умолчанию параметр `AllowMultiple` принимает ложное значение (`false`), а параметр `Inherited` — истинное значение (`true`).

В классе атрибута `AttributeUsage` определяется также доступное только для чтения свойство `ValidOn`. Оно возвращает значение типа `AttributeTargets`, определяющее типы элементов, к которым можно применять объявляемый атрибут. По умолчанию используется значение `AttributeTargets.All`.

Атрибут `Conditional`

Атрибут `Conditional` представляет, вероятно, наибольший интерес среди всех встроенных атрибутов. Ведь он позволяет создавать *условные методы*, которые вызываются только в том случае, если с помощью директивы `#define` определен конкретный идентификатор, а иначе метод пропускается. Следовательно, условный метод служит альтернативой условной компиляции по директиве `#if`.

`Conditional` — это, по существу, еще одно наименование класса `System.Diagnostics.ConditionalAttribute`. Для применения атрибута `Conditional` в исходный код программы следует включить пространство имен `System.Diagnostics`.

Рассмотрим применение данного атрибута на следующем примере программы.

```
// Продемонстрировать применение встроенного атрибута Conditional.

#define TRIAL

using System;
using System.Diagnostics;

class Test {
  [Conditional("TRIAL")]
  void Trial() {
    Console.WriteLine("Пробная версия, не " +
                      "предназначенная для распространения.");
  }
```

```
  [Conditional("RELEASE")]
  void Release() {
    Console.WriteLine("Окончательная рабочая версия.");
  }

  static void Main() {
    Test t = new Test();

    t.Trial(); //вызывается только в том случае, если
               // определен идентификатор TRIAL
    t.Release(); // вызывается только в том случае, если
                 // определен идентификатор RELEASE
  }
}
```

Эта программа дает следующий результат.

```
Пробная версия, не предназначенная для распространения.
```

Рассмотрим эту программу подробнее, чтобы стал понятнее результат ее выполнения. Прежде всего обратите внимание на то, что в этой программе определяется идентификатор `TRIAL`. Затем обратите внимание на определение методов `Trial()` и `Release()`. Каждому из них предшествует атрибут `Conditional`, общая форма которого приведена ниже:

```
[Conditional идентификатор]
```

где *идентификатор* обозначает конкретный идентификатор, определяющий условие выполнение метода. Данный атрибут может применяться только к методам. Если идентификатор определен, то метод выполняется, когда он вызывается. Если же идентификатор не определен, то метод не выполняется.

Оба метода, `Trial()` и `Release()`, вызываются в методе `Main()`. Но поскольку определен один лишь идентификатор `TRIAL`, то выполняется только метод `Trial()`, тогда как метод `Release()` игнорируется. Если же определить идентификатор `RELEASE`, то метод `Release()` будет также выполняться. А если удалить определение идентификатора `TRIAL`, то метод `Trial()` выполняться не будет.

Атрибут `Conditional` можно также применить в классе атрибута, т.е. в классе, наследующем от класса `Attribute`. Так, если идентификатор определен, то атрибут применяется, когда он встречается в ходе компиляции. В противном случае он не применяется.

На условные методы накладывается ряд ограничений. Во-первых, они должны возвращать значение типа `void`, а по существу, ничего не возвращать. Во-вторых, они должны быть членами класса или структуры, а не интерфейса. И в-третьих, они не могут предшествовать ключевому слову `override`.

Атрибут `Obsolete`

Атрибут `Obsolete` (сокращенное наименование класса `System.ObsoleteAttribute`) позволяет пометить элемент программы как устаревший. Ниже приведена общая форма этого атрибута:

```
[Obsolete("сообщение")]
```

где *сообщение* выводится при компилировании элемента программы, помеченного как устаревший. Ниже приведен краткий пример применения данного атрибута.

```
// Продемонстрировать применение атрибута Obsolete.

using System;

class Test {

  [Obsolete("Лучше использовать метод MyMeth2.")]
  public static int MyMeth(int a, int b) {
    return a / b;
  }

  // Усовершенствованный вариант метода MyMeth.
  public static int MyMeth2(int a, int b) {
    return b == 0 ? 0 : a /b;
  }

  static void Main() {
    // Для этого кода выводится предупреждение.
    Console.WriteLine("4 / 3 равно " + Test.MyMeth(4, 3));

    // А для этого кода предупреждение не выводится.
    Console.WriteLine("4 / 3 равно " + Test.MyMeth2(4, 3));
  }
}
```

Когда по ходу компиляции программы в методе `Main()` встречается вызов метода `MyMeth()`, формируется предупреждение, уведомляющее пользователя о том, что ему лучше воспользоваться методом `MyMeth2()`.

Ниже приведена вторая форма атрибута `Obsolete`:

```
[Obsolete("сообщение", ошибка)]
```

где *ошибка* обозначает логическое значение. Если это значение истинно (`true`), то при использовании устаревшего элемента формируется сообщение об ошибке компиляции вместо предупреждения. Эта форма отличается тем, что программа, содержащая подобную ошибку, не будет скомпилирована в исполняемом виде.

18 ГЛАВА

Обобщения

Эта глава посвящена *обобщениям* — одному из самых сложных и эффективных средств C#. Любопытно, что обобщения не вошли в первоначальную версию 1.0 и появились лишь в версии 2.0, но теперь они являются неотъемлемой частью языка C#. Не будет преувеличением сказать, что внедрение обобщений коренным образом изменило характер C#. Это нововведение не только означало появление нового элемента синтаксиса данного языка, но и открыло новые возможности для внесения многочисленных изменений и обновлений в библиотеку классов. И хотя после внедрения обобщений прошло уже несколько лет, последствия этого важного шага до сих пор сказываются на развитии C# как языка программирования.

Обобщения как языковое средство очень важны потому, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты для обработки разнотипных данных с соблюдением типовой безопасности. Как вам должно быть известно, многие алгоритмы очень похожи по своей логике независимо от типа данных, к которым они применяются. Например, механизм, поддерживающий очередь, остается одинаковым независимо от того, предназначена ли очередь для хранения элементов типа `int`, `string`, `object` или для класса, определяемого пользователем. До появления обобщений для обработки данных разных типов приходилось создавать различные варианты одного и того же алгоритма. А благодаря обобщениям можно сначала выработать единое решение независимо от конкретного типа данных, а затем применить его к обработке данных самых разных типов без каких-либо дополнительных усилий.

В этой главе описываются синтаксис, теория и практика применения обобщений, а также показывается, каким образом обобщения обеспечивают типовую безопасность в ряде случаев, которые раньше считались сложными. После прочтения настоящей главы у вас невольно возникнет желание ознакомиться с материалом главы 25, посвященной коллекциям, так как в ней приведено немало примеров применения обобщений в классах обобщенных коллекций.

Что такое обобщения

Термин *обобщение*, по существу, означает *параметризированный тип*. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется *обобщенным*, как, например, *обобщенный класс* или *обобщенный метод*.

Следует особо подчеркнуть, что в C# всегда имелась возможность создавать обобщенный код, оперируя ссылками типа `object`. А поскольку класс `object` является базовым для всех остальных классов, то по ссылке типа `object` можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа `object`.

Но дело в том, что в таком коде трудно было соблюсти типовую безопасность, поскольку для преобразования типа `object` в конкретный тип данных требовалось приведение типов. А это служило потенциальным источником ошибок из-за того, что приведение типов могло быть неумышленно выполнено неверно. Это затруднение позволяют преодолеть обобщения, обеспечивая типовую безопасность, которой раньше так недоставало. Кроме того, обобщения упрощают весь процесс, поскольку исключают необходимость выполнять приведение типов для преобразования объекта или другого типа обрабатываемых данных. Таким образом, обобщения расширяют возможности повторного использования кода и позволяют делать это надежно и просто.

ПРИМЕЧАНИЕ

Программирующим на C++ и Java необходимо иметь в виду, что обобщения в C# не следует путать с шаблонами в C++ и обобщениями в Java, поскольку это разные, хотя и похожие средства. В действительности между этими тремя подходами к реализации обобщений существуют коренные различия. Если вы имеете некоторый опыт программирования на C++ или Java, то постарайтесь на основании этого опыта не делать никаких далеко идущих выводов о том, как обобщения действуют в C#.

Простой пример обобщений

Начнем рассмотрение обобщений с простого примера обобщенного класса. В приведенной ниже программе определяются два класса. Первым из них является обобщенный класс `Gen`, вторым — класс `GenericsDemo`, в котором используется класс `Gen`.

```
// Простой пример обобщенного класса.

using System;

// В приведенном ниже классе Gen параметр типа T заменяется
// реальным типом данных при создании объекта типа Gen.
class Gen<T> {
  T ob; // объявить переменную типа T

  // Обратите внимание на то, что у этого конструктора имеется параметр типа T.
  public Gen(T o) {
    ob = o;
  }

  // Возвратить переменную экземпляра ob, которая относится к типу T.
  public T GetOb() {
    return ob;
  }

  // Показать тип T.
  public void ShowType() {
    Console.WriteLine("К типу T относится " + typeof(T));
  }
}

// Продемонстрировать применение обобщенного класса.
class GenericsDemo {
  static void Main() {
    // Создать переменную ссылки на объект Gen типа int.
    Gen<int> iOb;

    // Создать объект типа Gen<int> и присвоить ссылку на него переменной iOb.
    iOb = new Gen<int>(102);

    // Показать тип данных, хранящихся в переменной iOb.
    iOb.ShowType();

    // Получить значение переменной iOb.
    int v = iOb.GetOb();
    Console.WriteLine("Значение: " + v);

    Console.WriteLine();

    // Создать объект типа Gen для строк.
    Gen<string> strOb = new Gen<string>("Обобщения повышают эффективность.");

    // Показать тип данных, хранящихся в переменной strOb.
    strOb.ShowType();

    // Получить значение переменной strOb.
    string str = strOb.GetOb();
    Console.WriteLine("Значение: " + str);
  }
}
```

Эта программа дает следующий результат.

```
К типу T относится System.Int32
Значение: 102

К типу T относится System.String
Значение: Обобщения повышают эффективность.
```

Внимательно проанализируем эту программу. Прежде всего обратите внимание на объявление класса `Gen` в приведенной ниже строке кода:

```
class Gen<T> {
```

где `T` — это имя *параметра типа*. Это имя служит в качестве метки-заполнителя конкретного типа, который указывается при создании объекта класса `Gen`. Следовательно, имя `T` используется в классе `Gen` всякий раз, когда требуется параметр типа. Обратите внимание на то, что имя `T` заключается в угловые скобки (< >). Этот синтаксис можно обобщить: всякий раз, когда объявляется параметр типа, он указывается в угловых скобках. А поскольку параметр типа используется в классе `Gen`, то такой класс считается *обобщенным*.

В объявлении класса `Gen` можно указывать любое имя параметра типа, но по традиции выбирается имя `T`. К числу других наиболее употребительных имен параметров типа относятся `V` и `E`. Вы, конечно, вольны использовать и более описательные имена, например `TValue` или `TKey`. Но в этом случае первой в имени параметра типа принято указывать прописную букву `T`.

Далее имя `T` используется для объявления переменной `ob`, как показано в следующей строке кода.

```
T ob; // объявить переменную типа T
```

Как пояснялось выше, имя параметра типа `T` служит меткой-заполнителем конкретного типа, указываемого при создании объекта класса `Gen`. Поэтому переменная `ob` будет иметь тип, *привязываемый* к `T` при получении экземпляра объекта класса `Gen`. Так, если вместо `T` указывается тип `string`, то в экземпляре данного объекта переменная `ob` будет иметь тип `string`.

А теперь рассмотрим конструктор класса `Gen`.

```
public Gen(T o) {
  ob = o;
}
```

Как видите, параметр `o` этого конструктора относится к типу `T`. Это означает, что конкретный тип параметра `o` определяется типом, привязываемым к `T` при создании объекта класса `Gen`. А поскольку параметр `o` и переменная экземпляра `ob` относятся к типу `T`, то после создания объекта класса `Gen` их конкретный тип окажется одним и тем же.

С помощью параметра типа `T` можно также указывать тип, возвращаемый методом, как показано ниже на примере метода `GetOb()`.

```
public T GetOb() {
  return ob;
}
```

Переменная `ob` также относится к типу `T`, поэтому ее тип совпадает с типом, возвращаемым методом `GetOb()`.

Метод ShowType() отображает тип параметра T, передавая его оператору typeof. Но поскольку реальный тип подставляется вместо T при создании объекта класса Gen, то оператор typeof получит необходимую информацию о конкретном типе.

В классе GenericsDemo демонстрируется применение обобщенного класса Gen. Сначала в нем создается вариант класса Gen для типа int.

```
Gen<int> iOb;
```

Внимательно проанализируем это объявление. Прежде всего обратите внимание на то, что тип int указывается в угловых скобках после имени класса Gen. В этом случае int служит *аргументом типа*, привязанным к параметру типа T в классе Gen. В данном объявлении создается вариант класса Gen, в котором тип T заменяется типом int везде, где он встречается. Следовательно, после этого объявления int становится типом переменной ob и возвращаемым типом метода GetOb().

В следующей строке кода переменной iOb присваивается ссылка на экземпляр объекта класса Gen для варианта типа int.

```
iOb = new Gen<int>(102);
```

Обратите внимание на то, что при вызове конструктора класса Gen указывается также аргумент типа int. Это необходимо потому, что переменная (в данном случае — iOb), которой присваивается ссылка, относится к типу Gen<int>. Поэтому ссылка, возвращаемая оператором new, также должна относиться к типу Gen<int>. В противном случае во время компиляции возникнет ошибка. Например, приведенное ниже присваивание станет причиной ошибки во время компиляции.

```
iOb = new Gen<double>(118.12); // Ошибка!
```

Переменная iOb относится к типу Gen<int> и поэтому не может использоваться для ссылки на объект типа Gen<double>. Такой контроль типов относится к одним из главных преимуществ обобщений, поскольку он обеспечивает типовую безопасность.

Затем в программе отображается тип переменной ob в объекте iOb — тип System.Int32. Это структура .NET, соответствующая типу int. Далее значение переменной ob получается в следующей строке кода.

```
int v = iOb.GetOb();
```

Возвращаемым для метода GetOb() является тип T, который был заменен на тип int при объявлении переменной iOb, и поэтому метод GetOb() возвращает значение того же типа int. Следовательно, данное значение может быть присвоено переменной v типа int.

Далее в классе GenericsDemo объявляется объект типа Gen<string>.

```
Gen<string> strOb = new Gen<string>("Обобщения повышают эффективность.");
```

В этом объявлении указывается аргумент типа string, поэтому в объекте класса Gen вместо T подставляется тип string. В итоге создается вариант класса Gen для типа string, как демонстрируют остальные строки кода рассматриваемой здесь программы.

Прежде чем продолжить изложение, следует дать определение некоторым терминам. Когда для класса Gen указывается аргумент типа, например int или string, то создается так называемый в C# *закрыто сконструированный тип*. В частности, Gen<int> является закрыто сконструированным типом. Ведь, по существу, такой обобщенный тип, как Gen<T>, является абстракцией. И только после того, как будет сконструирован конкретный вариант, например Gen<int>, создается конкретный тип. А конструк-

ция, подобная Gen<T>, называется в C# *открыто сконструированным типом*, поскольку в ней указывается параметр типа T, но не такой конкретный тип, как int.

В C# чаще определяются такие понятия, как *открытый* и *закрытый типы*. Открытым типом считается такой параметр типа или любой обобщенный тип, для которого аргумент типа является параметром типа или же включает его в себя. А любой тип, не относящийся к открытому, считается закрытым. *Сконструированным типом* считается такой обобщенный тип, для которого предоставлены все аргументы типов. Если все эти аргументы относятся к закрытым типам, то такой тип считается закрыто сконструированным. А если один или несколько аргументов типа относятся к открытым типам, то такой тип считается открыто сконструированным.

Различение обобщенных типов по аргументам типа

Что касается обобщенных типов, то следует иметь в виду, что ссылка на один конкретный вариант обобщенного типа не совпадает по типу с другим вариантом того же самого обобщенного типа. Так, если ввести в приведенную выше программу следующую строку кода, то она не будет скомпилирована.

```
iOb = strOb; // Неверно!
```

Несмотря на то что обе переменные, iOb и strOb, относятся к типу Gen<T>, они ссылаются на разные типы, поскольку у них разные аргументы.

Повышение типовой безопасности с помощью обобщений

В связи с изложенным выше возникает следующий резонный вопрос: если аналогичные функциональные возможности обобщенного класса Gen можно получить и без обобщений, просто указав объект как тип данных и выполнив надлежащее приведение типов, то какая польза от того, что класс Gen делается обобщенным? Ответ на этот вопрос заключается в том, что обобщения автоматически обеспечивают типовую безопасность всех операций, затрагивающих класс Gen. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

Для того чтобы стали более понятными преимущества обобщений, рассмотрим сначала программу, в которой создается необобщенный аналог класса Gen.

```
// Класс NonGen является полным функциональным аналогом
// класса Gen, но без обобщений.

using System;

class NonGen {
  object ob; // переменная ob теперь относится к типу object

  // Передать конструктору ссылку на объект типа object.
  public NonGen(object o) {
    ob = o;
  }

  // Возвратить объект типа object.
  public object GetOb() {
    return ob;
```

```
  }

  // Показать тип переменной ob.
  public void ShowType() {
    Console.WriteLine("Тип переменной ob: " + ob.GetType());
  }
}

// Продемонстрировать применение необобщенного класса.
class NonGenDemo {
  static void Main() {
    NonGen iOb;

    // Создать объект класса NonGen.
    iOb = new NonGen(102);

    // Показать тип данных, хранящихся в переменной iOb.
    iOb.ShowType();

    // Получить значение переменной iOb.
    // На этот раз потребуется приведение типов.
    int v = (int) iOb.GetOb();
    Console.WriteLine("Значение: " + v);

    Console.WriteLine();

    // Создать еще один объект класса NonGen и
    // сохранить строку в переменной it.
    NonGen strOb = new NonGen("Тест на необобщенность");

   // Показать тип данных, хранящихся в переменной strOb.
    strOb.ShowType();

    // Получить значение переменной strOb.
    // И в этом случае требуется приведение типов.
    String str = (string) strOb.GetOb();
    Console.WriteLine("Значение: " + str);

    // Этот код компилируется, но он принципиально неверный!
    iOb = strOb;

    // Следующая строка кода приводит к исключительной
    // ситуации во время выполнения.
    // v = (int) iOb.GetOb(); // Ошибка при выполнении!
  }
}
```

При выполнении этой программы получается следующий результат.

```
Тип переменной ob: System.Int32
Значение: 102

Тип переменной ob: System.String
Значение: Тест на необобщенность
```

Как видите, результат выполнения этой программы такой же, как и у предыдущей программы.

В этой программе обращает на себя внимание ряд любопытных моментов. Прежде всего, тип T заменен везде, где он встречается в классе NonGen. Благодаря этому в классе NonGen может храниться объект любого типа, как и в обобщенном варианте этого класса. Но такой подход оказывается непригодным по двум причинам. Во-первых, для извлечения хранящихся данных требуется явное приведение типов. И во-вторых, многие ошибки несоответствия типов не могут быть обнаружены вплоть до момента выполнения программы. Рассмотрим каждую из этих причин более подробно.

Начнем со следующей строки кода.

```
int v = (int) iOb.GetOb();
```

Теперь возвращаемым типом метода GetOb() является object, а следовательно, для распаковки значения, возвращаемого методом GetOb(), и его последующего сохранения в переменной v требуется явное приведение к типу int. Если исключить приведение типов, программа не будет скомпилирована. В обобщенной версии этой программы приведение типов не требовалось, поскольку тип int указывался в качестве аргумента типа при создании объекта iOb. А в необобщенной версии этой программы потребовалось явное приведение типов. Но это не только неудобно, но и чревато ошибками.

А теперь рассмотрим следующую последовательность кода в конце анализируемой здесь программы.

```
// Этот код компилируется, но он принципиально неверный!
iOb = strOb;

// Следующая строка кода приводит к исключительной
// ситуации во время выполнения.
// v = (int) iOb.GetOb(); // Ошибка при выполнении!
```

В этом коде значение переменной strOb присваивается переменной iOb. Но переменная strOb ссылается на объект, содержащий символьную строку, а не целое значение. Такое присваивание оказывается верным с точки зрения синтаксиса, поскольку все ссылки на объекты класса NonGen одинаковы, а значит, по ссылке на один объект класса NonGen можно обращаться к любому другому объекту класса NonGen. Тем не менее такое присваивание неверно с точки зрения семантики, как показывает следующая далее закомментированная строка кода. В этой строке тип, возвращаемый методом GetOb(), приводится к типу int, а затем предпринимается попытка присвоить полученное в итоге значение переменной int. К сожалению, в отсутствие обобщений компилятор не сможет выявить подобную ошибку. Вместо этого возникнет исключительная ситуация во время выполнения, когда будет предпринята попытка приведения к типу int. Для того чтобы убедиться в этом, удалите символы комментария в начале данной строки кода, скомпилируйте, а затем выполните программу. При ее выполнении возникнет ошибка.

Упомянутая выше ситуация не могла бы возникнуть, если бы в программе использовались обобщения. Компилятор выявил бы ошибку в приведенной выше последовательности кода, если бы она была включена в обобщенную версию программы, и сообщил бы об этой ошибке, предотвратив тем самым серьезный сбой, приводящий к исключительной ситуации при выполнении программы. Возможность создавать типизированный код, в котором ошибки несоответствия типов выявляются во время

компиляции, является главным преимуществом обобщений. Несмотря на то что в C# всегда имелась возможность создавать "обобщенный" код, используя ссылки на объекты, такой код не был типизированным, т.е. не обеспечивал типовую безопасность, а его неправильное применение могло привести к исключительным ситуациям во время выполнения. Подобные ситуации исключаются благодаря обобщениям. По существу, обобщения переводят ошибки при выполнении в разряд ошибок при компиляции. В этом и заключается основная польза от обобщений.

В рассматриваемой здесь необобщенной версии программы имеется еще один любопытный момент. Обратите внимание на то, как тип переменной `ob` экземпляра класса `NonGen` создается с помощью метода `ShowType()` в следующей строке кода.

```
Console.WriteLine("Тип переменной ob: " + ob.GetType());
```

Как пояснялось в главе 11, в классе `object` определен ряд методов, доступных для всех типов данных. Одним из них является метод `GetType()`, возвращающий объект класса `Type`, который описывает тип вызывающего объекта во время выполнения. Следовательно, конкретный тип объекта, на который ссылается переменная `ob`, становится известным во время выполнения, несмотря на то, что тип переменной `ob` указан в исходном коде как `object`. Именно поэтому в среде CLR будет сгенерировано исключение при попытке выполнить неверное приведение типов во время выполнения программы.

Обобщенный класс с двумя параметрами типа

В классе обобщенного типа можно указать два или более параметра типа. В этом случае параметры типа указываются списком через запятую. В качестве примера ниже приведен класс `TwoGen`, являющийся вариантом класса `Gen` с двумя параметрами типа.

```
// Простой обобщенный класс с двумя параметрами типа T и V.

using System;

class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Обратите внимание на то, что в этом конструкторе
  // указываются параметры типа T и V.
  public TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Показать типы T и V.
  public void showTypes() {
    Console.WriteLine("К типу T относится " + typeof(T));
    Console.WriteLine("К типу V относится " + typeof(V));
  }

  public T getob1() {
```

```
    return ob1;
  }

  public V GetObj2() {
    return ob2;
  }
}

// Продемонстрировать применение обобщенного класса с двумя параметрами типа.
class SimpGen {
  static void Main() {

    TwoGen<int, string> tgObj =
      new TwoGen<int, string>(119, "Альфа Бета Гамма");

    // Показать типы.
    tgObj.showTypes();

    // Получить и вывести значения.
    int v = tgObj.getob1();
    Console.WriteLine("Значение: " + v);
    string str = tgObj.GetObj2();
    Console.WriteLine("Значение: " + str);
  }
}
```

Эта программа дает следующий результат.

```
К типу T относится System.Int32
К типу V относится System.String
Значение: 119
Значение: Альфа Бета Гамма
```

Обратите внимание на то, как объявляется класс `TwoGen`.

```
class TwoGen<T, V> {
```

В этом объявлении указываются два параметра типа `T` и `V`, разделенные запятой. А поскольку у класса `TwoGen` два параметра типа, то при создании объекта этого класса необходимо указывать два соответствующих аргумента типа, как показано ниже.

```
TwoGen<int, string> tgObj =
  new TwoGen<int, string>(119, "Альфа Бета Гамма");
```

В данном случае вместо `T` подставляется тип `int`, а вместо `V` — тип `string`.

В представленном выше примере указываются аргументы разного типа, но они могут быть и одного типа. Например, следующая строка кода считается вполне допустимой.

```
TwoGen<string, string> x =
  new TwoGen<string, string>("Hello", "Goodbye");
```

В этом случае оба типа, `T` и `V`, заменяются одним и тем же типом `string`. Ясно, что если бы аргументы были одного и того же типа, то два параметра типа были бы не нужны.

Общая форма обобщенного класса

Синтаксис обобщений, представленных в предыдущих примерах, может быть сведен к общей форме. Ниже приведена общая форма объявления обобщенного класса.

```
class имя_класса<список_параметров_типа> { // ...
```

А вот как выглядит синтаксис объявления ссылки на обобщенный класс.

```
имя_класса<список_аргументов_типа> имя_переменной =
    new имя_класса<список_параметров_типа> (список_аргументов_конструктора);
```

Ограниченные типы

В предыдущих примерах параметры типа можно было заменить любым типом данных. Например, в следующей строке кода объявляется любой тип, обозначаемый как `T`.

```
class Gen<T> {
```

Это означает, что вполне допустимо создавать объекты класса `Gen`, в которых тип `T` заменяется типом `int`, `double`, `string`, `FileStream` или любым другим типом данных. Во многих случаях отсутствие ограничений на указание аргументов типа считается вполне приемлемым, но иногда оказывается полезно ограничить круг типов, которые могут быть указаны в качестве аргумента типа.

Допустим, что требуется создать метод, оперирующий содержимым потока, включая объекты типа `FileStream` или `MemoryStream`. На первый взгляд, такая ситуация идеально подходит для применения обобщений, но при этом нужно каким-то образом гарантировать, что в качестве аргументов типа будут использованы только типы потоков, но не `int` или любой другой тип. Кроме того, необходимо как-то уведомить компилятор о том, что методы, определяемые в классе потока, будут доступны для применения. Так, в обобщенном коде должно быть каким-то образом известно, что в нем может быть вызван метод `Read()`.

Для выхода из подобных ситуаций в C# предусмотрены *ограниченные типы*. Указывая параметр типа, можно наложить определенное ограничение на этот параметр. Это делается с помощью оператора `where` при указании параметра типа:

```
class имя_класса<параметр_типа> where параметр_типа : ограничения { // ...
```

где *ограничения* указываются списком через запятую.

В C# предусмотрен ряд ограничений на типы данных.

- *Ограничение на базовый класс*, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является *неприкрытое ограничение типа*, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- *Ограничение на интерфейс*, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

- *Ограничение на конструктор*, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора `new()`.
- *Ограничение ссылочного типа*, требующее указывать аргумент ссылочного типа с помощью оператора `class`.
- *Ограничение типа значения*, требующее указывать аргумент типа значения с помощью оператора `struct`.

Среди всех этих ограничений чаще всего применяются ограничения на базовый класс и интерфейс, хотя все они важны в равной степени. Каждое из этих ограничений рассматривается далее по порядку.

Применение ограничения на базовый класс

Ограничение на базовый класс позволяет указывать базовый класс, который должен наследоваться аргументом типа. Ограничение на базовый класс служит двум главным целям. Во-первых, оно позволяет использовать в обобщенном классе те члены базового класса, на которые указывает данное ограничение. Это дает, например, возможность вызвать метод или обратиться к свойству базового класса. В отсутствие ограничения на базовый класс компилятору ничего не известно о типе членов, которые может иметь аргумент типа. Накладывая ограничение на базовый класс, вы тем самым даете компилятору знать, что все аргументы типа будут иметь члены, определенные в этом базовом классе.

И во-вторых, ограничение на базовый класс гарантирует использование только тех аргументов типа, которые поддерживают указанный базовый класс. Это означает, что для любого ограничения, накладываемого на базовый класс, аргумент типа должен обозначать сам базовый класс или производный от него класс. Если же попытаться использовать аргумент типа, не соответствующий указанному базовому классу или не наследующий его, то в результате возникнет ошибка во время компиляции.

Ниже приведена общая форма наложения ограничения на базовый класс, в которой используется оператор `where`:

```
where T : имя_базового_класса
```

где *T* обозначает имя параметра типа, а *имя_базового_класса* — конкретное имя ограничиваемого базового класса. Одновременно в этой форме ограничения может быть указан только один базовый класс.

В приведенном ниже простом примере демонстрируется механизм наложения ограничения на базовый класс.

```
// Простой пример, демонстрирующий механизм наложения
// ограничения на базовый класс.

using System;

class A {
  public void Hello() {
    Console.WriteLine("Hello");
  }
}
```

```
// Класс B наследует класс A.
class B : A { }

  // Класс C не наследует класс A.
class C { }

// В силу ограничения на базовый класс во всех аргументах типа,
// передаваемых классу Test, должен присутствовать базовый класс A.
class Test<T> where T : A {
  T obj;

  public Test(T o) {
    obj = o;
  }

  public void SayHello() {
    // Метод Hello() вызывается, поскольку он объявлен в базовом классе A.
    obj.Hello();
  }
}

class BaseClassConstraintDemo {
  static void Main() {
    A a = new A();
    B b = new B();
    C c = new C();

    // Следующий код вполне допустим, поскольку класс A указан как базовый.
    Test<A> t1 = new Test<A>(a);

    t1.SayHello();

    // Следующий код вполне допустим, поскольку класс B наследует от класса A.
    Test<B> t2 = new Test<B>(b);

    t2.SayHello();

    // Следующий код недопустим, поскольку класс C не наследует от класса A.
// Test<C> t3 = new Test<C>(c); // Ошибка!
// t3.SayHello(); // Ошибка!
  }
}
```

В данном примере кода класс `A` наследуется классом `B`, но не наследуется классом `C`. Обратите также внимание на то, что в классе `A` объявляется метод `Hello()`, а класс `Test` объявляется как обобщенный следующим образом.

```
class Test<T> where T : A {
```

Оператор `where` в этом объявлении накладывает следующее ограничение: любой аргумент, указываемый для типа `T`, должен иметь класс `A` в качестве базового.

А теперь обратите внимание на то, что в классе `Test` объявляется метод `SayHello()`, как показано ниже.

```
public void SayHello() {
  // Метод Hello() вызывается, поскольку он объявлен в базовом классе A.
  obj.Hello();
}
```

Этот метод вызывает в свою очередь метод `Hello()` для объекта `obj` типа `T`. Любопытно, что единственным основанием для вызова метода `Hello()` служит следующее требование ограничения на базовый класс: любой аргумент типа, привязанный к типу `T`, должен относиться к классу `A` или наследовать от класса `A`, в котором объявлен метод `Hello()`. Следовательно, любой допустимый тип `T` будет также определять метод `Hello()`. Если бы данное ограничение на базовый класс не было наложено, то компилятору ничего не было бы известно о том, что метод `Hello()` может быть вызван для объекта типа `T`. Убедитесь в этом сами, удалив оператор `where` из объявления обобщенного класса `Test`. В этом случае программа не подлежит компиляции, поскольку теперь метод `Hello()` неизвестен.

Помимо разрешения доступа к членам базового класса, ограничение на базовый класс гарантирует, что в качестве аргументов типа могут быть переданы только те типы данных, которые наследуют базовый класс. Именно поэтому приведенные ниже строки кода закомментированы.

```
// Test<C> t3 = new Test<C>(c); // Ошибка!
// t3.SayHello(); // Ошибка!
```

Класс `C` не наследует от класса `A`, и поэтому он не может использоваться в качестве аргумента типа при создании объекта типа `Test`. Убедитесь в этом сами, удалив символы комментария и попытавшись перекомпилировать этот код.

Прежде чем продолжить изложение дальше, рассмотрим вкратце два последствия наложения ограничения на базовый класс. Во-первых, это ограничение разрешает доступ к членам базового класса из обобщенного класса. И во-вторых, оно гарантирует допустимость только тех аргументов типа, которые удовлетворяют данному ограничению, обеспечивая тем самым типовую безопасность.

В предыдущем примере показано, как накладывается ограничение на базовый класс, но из него не совсем ясно, зачем это вообще нужно. Для того чтобы особое значение ограничения на базовый класс стало понятнее, рассмотрим еще один, более практический пример. Допустим, что требуется реализовать механизм управления списками телефонных номеров, чтобы пользоваться разными категориями таких списков, в частности отдельными списками для друзей, поставщиков, клиентов и т.д. Для этой цели можно сначала создать класс `PhoneNumber`, в котором будут храниться имя абонента и номер его телефона. Такой класс может иметь следующий вид.

```
// Базовый класс, в котором хранятся имя абонента и номер его телефона.
class PhoneNumber {
  public PhoneNumber(string n, string num) {
    Name = n;
    Number = num;
  }

  // Автоматически реализуемые свойства, в которых
  // хранятся имя абонента и номер его телефона.
  public string Number { get; set; }
  public string Name { get; set; }
}
```

Далее создадим классы, наследующие класс PhoneNumber: Friend и Supplier. Эти классы приведены ниже.

```
// Класс для телефонных номеров друзей.
class Friend : PhoneNumber {

  public Friend(string n, string num, bool wk) :
    base(n, num)
  {
    IsWorkNumber = wk;
   }

  public bool IsWorkNumber { get; private set; }
    // ...
  }

// Класс для телефонных номеров поставщиков.
class Supplier : PhoneNumber {
  public Supplier(string n, string num) :
    base(n, num) { }

  // ...
}
```

Обратите внимание на то, что в класс Friend введено свойство IsWorkNumber, возвращающее логическое значение true, если номер телефона является рабочим.

Для управления списками телефонных номеров создадим еще один класс под названием PhoneList. Его следует сделать обобщенным, поскольку он должен служить для управления любым списком телефонных номеров. В функции такого управления должен, в частности, входить поиск телефонных номеров по заданным именам и наоборот, поэтому на данный класс необходимо наложить ограничение по типу, требующее, чтобы объекты, сохраняемые в списке, были экземплярами класса, производного от класса PhoneNumber.

```
// Класс PhoneList способен управлять любым видом списка телефонных
// номеров, при условии, что он является производным от класса PhoneNumber.

class PhoneList<T> where T : PhoneNumber {
  T[] phList;
  int end;

  public PhoneList() {
    phList = new T[10];
    end = 0;
  }

  // Добавить элемент в список.
  public bool Add(T newEntry) {
    if(end == 10) return false;
    phList[end] = newEntry;
    end++;
    return true;
  }
```

```
  // Найти и возвратить сведения о телефоне по заданному имени.
  public T FindByName(string name) {
    for(int i=0; i<end; i++) {
      // Имя может использоваться, потому что его свойство Name
      // относится к членам класса PhoneNumber, который является
      // базовым по накладываемому ограничению.
      if(phList[i].Name == name)
        return phList[i];
    }

   // Имя отсутствует в списке.
   throw new NotFoundException();
  }

  // Найти и возвратить сведения о телефоне по заданному номеру.
  public T FindByNumber(string number) {
    for(int i=0; i<end; i++) {
      // Номер телефона также может использоваться, поскольку
      // его свойство Number относится к членам класса PhoneNumber,
      // который является базовым по накладываемому ограничению.
      if(phList[i].Number == number)
        return phList[i];
    }

  // Номер телефона отсутствует в списке.
  throw new NotFoundException();
  }

  // ...
}
```

Ограничение на базовый класс разрешает коду в классе `PhoneList` доступ к свойствам `Name` и `Number` для управления любым видом списка телефонных номеров. Оно гарантирует также, что для построения объекта класса `PhoneList` будут использоваться только доступные типы. Обратите внимание на то, что в классе `PhoneList` генерируется исключение `NotFoundException`, если имя или номер телефона не найдены. Это специальное исключение, объявляемое ниже.

```
class NotFoundException : Exception {
/* Реализовать все конструкторы класса Exception.
   Эти конструкторы выполняют вызов конструктора базового класса.
   Класс NotFoundException ничем не дополняет класс Exception и
   поэтому не требует никаких дополнительных действий. */

  public NotFoundException() : base() { }
  public NotFoundException(string str) : base(str) { }
  public NotFoundException(
    string str, Exception inner) : base(str, inner) { }
  protected NotFoundException(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
      base(si, sc) { }
}
```

В данном примере используется только конструктор, вызываемый по умолчанию, но ради наглядности этого примера в классе исключения `NotFoundException` реализуются все конструкторы, определенные в классе `Exception`. Обратите внимание на то, что эти конструкторы вызывают эквивалентный конструктор базового класса, определенный в классе `Exception`. А поскольку класс исключения `NotFoundException` ничем не дополняет базовый класс `Exception`, то для любых дополнительных действий нет никаких оснований.

В приведенной ниже программе все рассмотренные выше фрагменты кода объединяются вместе, а затем демонстрируется применение класса `PhoneList`. Кроме того, в ней создается класс `EmailFriend`. Этот класс не наследует от класса `PhoneNumber`, а следовательно, он *не* может использоваться для создания объектов класса `PhoneList`.

```
// Более практический пример, демонстрирующий применение
// ограничения на базовый класс.

using System;

// Специальное исключение, генерируемое в том случае,
// если имя или номер телефона не найдены.
class NotFoundException : Exception {
  /* Реализовать все конструкторы класса Exception.
     Эти конструкторы выполняют вызов конструктора базового класса.
     Класс NotFoundException ничем не дополняет класс Exception и
     поэтому не требует никаких дополнительных действий. */

    public NotFoundException() : base() { }
    public NotFoundException(string str) : base(str) { }
    public NotFoundException(
      string str,Exception inner) : base(str, inner) { }
    protected NotFoundException(
     System.Runtime.Serialization.SerializationInfo si,
     System.Runtime.Serialization.StreamingContext sc) :
       base(si, sc) { }
}

// Базовый класс, в котором хранятся имя абонента и номер его телефона.
class PhoneNumber {
  public PhoneNumber(string n, string num) {
    Name = n;
    Number = num;
  }

  public string Number { get; set; }
  public string Name { get; set; }
  }

  // Класс для телефонных номеров друзей.
class Friend : PhoneNumber {
  public Friend(string n, string num, bool wk) :
    base(n, num)
  {
    IsWorkNumber = wk;
```

```
  }

  public bool IsWorkNumber { get; private set; }

    // ...
}

// Класс для телефонных номеров поставщиков.
class Supplier : PhoneNumber {
  public Supplier(string n, string num) :
    base(n, num) { }

    // ...
}

// Этот класс не наследует от класса PhoneNumber.
class EmailFriend {
  // ...
}

// Класс PhoneList способен управлять любым видом списка телефонных номеров,
// при условии, что он является производным от класса PhoneNumber.
class PhoneList<T> where T : PhoneNumber {
  T[] phList;
  int end;

  public PhoneList() {
  phList = new T[10];
    end = 0;
  }

  // Добавить элемент в список.
  public bool Add(T newEntry) {
    if(end == 10) return false;

    phList[end] = newEntry;
    end++;
   return true;
  }

  // Найти и возвратить сведения о телефоне по заданному имени.
  public T FindByName(string name) {

    for(int i=0; i<end; i++) {
      // Имя может использоваться, потому что его свойство Name
      // относится к членам класса PhoneNumber, который является
      // базовым по накладываемому ограничению.
      if(phList[i].Name == name)
        return phList[i];
  }

    // Имя отсутствует в списке.
    throw new NotFoundException();
  }
```

```
  // Найти и возвратить сведения о телефоне по заданному номеру.
  public T FindByNumber(string number) {
  for(int i=0; i<end; i++) {
      // Номер телефона также может использоваться, поскольку
      // его свойство Number относится к членам класса PhoneNumber,
      // который является базовым по накладываемому ограничению.
      if(phList[i].Number == number)
        return phList[i];
  }

    // Номер телефона отсутствует в списке.
    throw new NotFoundException();
  }

  // ...
}

// Продемонстрировать наложение ограничений на базовый класс.
class UseBaseClassConstraint {
  static void Main() {
    // Следующий код вполне допустим, поскольку
    // класс Friend наследует от класса PhoneNumber.
    PhoneList<Friend> plist = new PhoneList<Friend>();
    plist.Add(new Friend("Том", "555-1234", true));
    plist.Add(new Friend("Гари", "555-6756", true));
    plist.Add(new Friend("Матт", "555-9254", false));

    try {
      // Найти номер телефона по заданному имени друга.
      Friend frnd = plist.FindByName("Гари");

      Console.Write(frnd.Name + ": " + frnd.Number);

      if(frnd.IsWorkNumber)
        Console.WriteLine(" (рабочий)");
      else
        Console.WriteLine();
    } catch(NotFoundException) {
      Console.WriteLine("Не найдено");
    }

    Console.WriteLine();

    // Следующий код также допустим, поскольку
    // класс Supplier наследует от класса PhoneNumber.
    PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
    plist2.Add(new Supplier("Фирма Global Hardware", "555-8834"));
    plist2.Add(new Supplier("Агентство Computer Warehouse", "555-9256"));
    plist2.Add(new Supplier("Компания NetworkCity", "555-2564"));
    try {
      // Найти наименование поставщика по заданному номеру телефона.
      Supplier sp = plist2.FindByNumber("555-2564");
      Console.WriteLine(sp.Name + ": " + sp.Number);
```

```
    } catch(NotFoundException) {
      Console.WriteLine("Не найдено");
    }

    // Следующее объявление недопустимо, поскольку
    // класс EmailFriend НЕ наследует от класса PhoneNumber.
// PhoneList<EmailFriend> plist3 =
// new PhoneList<EmailFriend>(); // Ошибка!
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Гари: 555-6756 (рабочий)
Компания NetworkCity: 555-2564
```

Поэкспериментируйте с этой программой. В частности, попробуйте составить разные виды списков телефонных номеров или воспользоваться свойством `IsWorkNumber` в классе `PhoneList`. Вы сразу же обнаружите, что компилятор не позволит вам этого сделать, потому что свойство `IsWorkNumber` определено в классе `Friend`, а не в классе `PhoneNumber`, а следовательно, оно неизвестно в классе `PhoneList`.

Применение ограничения на интерфейс

Ограничение на интерфейс позволяет указывать интерфейс, который должен быть реализован аргументом типа. Это ограничение служит тем же основным целям, что и ограничение на базовый класс. Во-первых, оно позволяет использовать члены интерфейса в обобщенном классе. И во-вторых, оно гарантирует использование только тех аргументов типа, которые реализуют указанный интерфейс. Это означает, что для любого ограничения, накладываемого на интерфейс, аргумент типа должен обозначать сам интерфейс или же тип, реализующий этот интерфейс.

Ниже приведена общая форма наложения ограничения на интерфейс, в которой используется оператор `where`:

```
where T : имя_интерфейса
```

где *T* — это имя параметра типа, а *имя_интерфейса* — конкретное имя ограничиваемого интерфейса. В этой форме ограничения может быть указан список интерфейсов через запятую. Если ограничение накладывается одновременно на базовый класс и интерфейс, то первым в списке должен быть указан базовый класс.

Ниже приведена программа, демонстрирующая наложение ограничения на интерфейс и представляющая собой переработанный вариант предыдущего примера программы, управляющей списками телефонных номеров. В этом варианте класс `PhoneNumber` преобразован в интерфейс `IPhoneNumber`, который реализуется в классах `Friend` и `Supplier`.

```
// Применить ограничение на интерфейс.

using System;

// Специальное исключение, генерируемое в том случае,
// если имя или номер телефона не найдены.
class NotFoundException : Exception {
```

```
  /* Реализовать все конструкторы класса Exception.
     Эти конструкторы выполняют вызов конструктора базового класса.
     Класс NotFoundException ничем не дополняет класс Exception и
     поэтому не требует никаких дополнительных действий. */

    public NotFoundException() : base() { }
    public NotFoundException(string str) : base(str) { }
    public NotFoundException(
      string str,Exception inner) : base(str, inner) { }
    protected NotFoundException(
     System.Runtime.Serialization.SerializationInfo si,
     System.Runtime.Serialization.StreamingContext sc) :
       base(si, sc) { }
}

// Интерфейс, поддерживающий имя и номер телефона.
public interface IPhoneNumber {

  string Number {
    get;
    set;
  }

  string Name {
    get;
    set;
  }
}

// Класс для телефонных номеров друзей.
// В нем реализуется интерфейс IPhoneNumber.
class Friend : IPhoneNumber {

  public Friend(string n, string num, bool wk) {
    Name = n;
    Number = num;

    IsWorkNumber = wk;
  }

  public bool IsWorkNumber { get; private set; }

  // Реализовать интерфейс IPhoneNumber.
  public string Number { get; set; }
  public string Name { get; set; }

  // ...
}

// Класс для телефонных номеров поставщиков.
class Supplier : IPhoneNumber {
  public Supplier(string n, string num) {
    Name = n;
    Number = num;
```

```
  }

  // Реализовать интерфейс IPhoneNumber.
  public string Number { get; set; }
  public string Name { get; set; }

  // ...
}

// В этом классе интерфейс IPhoneNumber не реализуется.
class EmailFriend {
  // ...
}

// Класс PhoneList способен управлять любым видом списка телефонных
// номеров, при условии, что он реализует интерфейс PhoneNumber.
class PhoneList<T> where T : IPhoneNumber {
  T[] phList;
  int end;

  public PhoneList() {
  phList = new T[10];
  end = 0;
  }

  public bool Add(T newEntry) {
    if(end == 10) return false;

    phList[end] = newEntry;
    end++;
    return true;
  }

  // Найти и возвратить сведения о телефоне по заданному имени.
  public T FindByName(string name) {

    for(int i=0; i<end; i++) {
      // Имя может использоваться, потому что его свойство Name
      // относится к членам интерфейса IPhoneNumber, на который
      // накладывается ограничение.
      if(phList[i].Name == name)
        return phList[i];
    }

    // Имя отсутствует в списке.
    throw new NotFoundException();
  }

  // Найти и возвратить сведения о телефоне по заданному номеру.
  public T FindByNumber(string number) {
    for(int i=0; i<end; i++) {
      // Номер телефона также может использоваться, поскольку его
      // свойство Number относится к членам интерфейса IPhoneNumber,
      // на который накладывается ограничение.
```

```
      if(phList[i].Number == number)
        return phList[i];
    }

    // Номер телефона отсутствует в списке.
    throw new NotFoundException();
  }

  // ...
}

// Продемонстрировать наложение ограничения на интерфейс.
class UseInterfaceConstraint {
  static void Main() {

    // Следующий код вполне допустим, поскольку
    // в классе Friend реализуется интерфейс IPhoneNumber.
    PhoneList<Friend> plist = new PhoneList<Friend>();
    plist.Add(new Friend("Том", "555-1234", true));
    plist.Add(new Friend("Гари", "555-6756", true));
    plist.Add(new Friend("Матт", "555-9254", false));

    try {
      // Найти номер телефона по заданному имени друга.
      Friend frnd = plist.FindByName("Гари");

      Console.Write(frnd.Name + ": " + frnd.Number);

      if(frnd.IsWorkNumber)
        Console.WriteLine(" (рабочий)");
      else
        Console.WriteLine();
    } catch(NotFoundException) {
      Console.WriteLine("Не найдено");
    }

    Console.WriteLine();

    // Следующий код также допустим, поскольку в классе Supplier
    // также реализуется интерфейс IPhoneNumber.
    PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
    plist2.Add(new Supplier("Фирма Global Hardware", "555-8834"));
    plist2.Add(new Supplier("Агентство Computer Warehouse", "555-9256"));
    plist2.Add(new Supplier("Компания NetworkCity", "555-2564"));

    try {
      // Найти наименование поставщика по заданному номеру телефона.
      Supplier sp = plist2.FindByNumber("555-2564");
      Console.WriteLine(sp.Name + ": " + sp.Number);
    } catch(NotFoundException) {
      Console.WriteLine("Не найдено");
    }

    // Следующее объявление недопустимо, поскольку
```

```
    // в классе EmailFriend НЕ реализуется интерфейс IPhoneNumber.
// PhoneList<EmailFriend> plist3 =
// new PhoneList<EmailFriend>(); // Ошибка!
  }
}
```

В этой версии программы ограничение на интерфейс, указываемое в классе `PhoneList`, требует, чтобы аргумент типа реализовал интерфейс `IPhoneList`. А поскольку этот интерфейс реализуется в обоих классах, `Friend` и `Supplier`, то они относятся к допустимым типам, привязываемым к типу `T`. В то же время интерфейс не реализуется в классе `EmailFriend`, и поэтому этот класс не может быть привязан к типу `T`. Для того чтобы убедиться в этом, удалите символы комментария в двух последних строках кода в методе `Main()`. Вы сразу же обнаружите, что программа не компилируется.

Применение ограничения new() на конструктор

Ограничение `new()` на конструктор позволяет получать экземпляр объекта обобщенного типа. Как правило, создать экземпляр параметра обобщенного типа не удается. Но это положение изменяет ограничение `new()`, поскольку оно требует, чтобы аргумент типа предоставил конструктор без параметров. Им может быть конструктор, вызываемый по умолчанию и предоставляемый автоматически, если явно определяемый конструктор отсутствует или же конструктор без параметров явно объявлен пользователем. Накладывая ограничение `new()`, можно вызывать конструктор без параметров для создания объекта.

Ниже приведен простой пример, демонстрирующий наложение ограничения `new()`.

```
// Продемонстрировать наложение ограничения new() на конструктор.

using System;

class MyClass {

  public MyClass() {
    // ...
  }

  //...
}

class Test<T> where T : new() {
  T obj;
  public Test() {

    // Этот код работоспособен благодаря наложению ограничения new().
    obj = new T(); // создать объект типа T
  }

  // ...
}

class ConsConstraintDemo {
```

```
  static void Main() {
    Test<MyClass> x = new Test<MyClass>();
  }
}
```

Прежде всего обратите внимание на объявление класса `Test`.

```
class Test<T> where T : new() {
```

В силу накладываемого ограничения `new()` любой аргумент типа должен предоставлять конструктор без параметров.

Далее проанализируем приведенный ниже конструктор класса `Test`.

```
public Test() {

  // Этот код работоспособен благодаря наложению ограничения new().
  obj = new T(); // создать объект типа T
}
```

В этом фрагменте кода создается объект типа `T`, и ссылка на него присваивается переменной экземпляра `obj`. Такой код допустим только потому, что ограничение `new()` требует наличия конструктора. Для того чтобы убедиться в этом, попробуйте сначала удалить ограничение `new()`, а затем попытайтесь перекомпилировать программу. В итоге вы получите сообщение об ошибке во время компиляции.

В методе `Main()` получается экземпляр объекта типа `Test`, как показано ниже.

```
Test<MyClass> x = new Test<MyClass>();
```

Обратите внимание на то, что аргументом типа в данном случае является класс `MyClass` и что в этом классе определяется конструктор без параметров. Следовательно, этот класс допускается использовать в качестве аргумента типа для класса `Test`. Следует особо подчеркнуть, что в классе `MyClass` совсем не обязательно определять конструктор без параметров явным образом. Его используемый по умолчанию конструктор вполне удовлетворяет накладываемому ограничению. Но если классу потребуются другие конструкторы, помимо конструктора без параметров, то придется объявить явным образом и вариант без параметров.

Что касается применения ограничения `new()`, то следует обратить внимание на три других важных момента. Во-первых, его можно использовать вместе с другими ограничениями, но последним по порядку. Во-вторых, ограничение `new()` позволяет конструировать объект, используя только конструктор без параметров, — даже если доступны другие конструкторы. Иными словами, передавать аргументы конструктору параметра типа не разрешается. И в-третьих, ограничение `new()` нельзя использовать одновременно с ограничением типа значения, рассматриваемым далее.

Ограничения ссылочного типа и типа значения

Два других ограничения позволяют указать на то, что аргумент, обозначающий тип, должен быть либо ссылочного типа, либо типа значения. Эти ограничения оказываются полезными в тех случаях, когда для обобщенного кода важно провести различие между ссылочным типом и типом значения. Ниже приведена общая форма ограничения ссылочного типа.

```
where T : class
```

В этой форме с оператором `where` ключевое слово `class` указывает на то, что аргумент *T* должен быть ссылочного типа. Следовательно, всякая попытка использовать тип значения, например `int` или `bool`, вместо *T* приведет к ошибке во время компиляции.

Ниже приведена общая форма ограничения типа значения.

```
where T : struct
```

В этой форме ключевое слово `struct` указывает на то, что аргумент *T* должен быть типа значения. (Напомним, что структуры относятся к типам значений.) Следовательно, всякая попытка использовать ссылочный тип, например `string`, вместо *T* приведет к ошибке во время компиляции. Но если имеются дополнительные ограничения, то в любом случае `class` или `struct` должно быть первым по порядку накладываемым ограничением.

Ниже приведен пример, демонстрирующий наложение ограничения ссылочного типа.

```
// Продемонстрировать наложение ограничения ссылочного типа.

using System;

class MyClass {
  //...
}

// Наложить ограничение ссылочного типа.
class Test<T> where T : class {
  T obj;

  public Test() {
    // Следующий оператор допустим только потому, что
    // аргумент T гарантированно относится к ссылочному
    // типу, что позволяет присваивать пустое значение.
    obj = null;
  }

  // ...
}

class ClassConstraintDemo {
  static void Main() {

    // Следующий код вполне допустим, поскольку MyClass является классом.
    Test<MyClass> x = new Test<MyClass>();

    // Следующая строка кода содержит ошибку, поскольку
    // int относится к типу значения.
// Test<int> y = new Test<int>();
  }
}
```

Обратите внимание на следующее объявление класса `Test`.

```
class Test<T> where T : class {
```

Ограничение `class` требует, чтобы любой аргумент `T` был ссылочного типа. В данном примере кода это необходимо для правильного выполнения операции присваивания в конструкторе класса `Test`.

```
public Test() {
  // Следующий оператор допустим только потому, что
  // аргумент T гарантированно относится к ссылочному
  // типу, что позволяет присваивать пустое значение.
  obj = null;
}
```

В этом фрагменте кода переменной `obj` типа `T` присваивается пустое значение. Такое присваивание допустимо только для ссылочных типов. Как правило, пустое значение нельзя присвоить переменной типа значения. (Исключением из этого правила является *обнуляемый тип*, который представляет собой специальный тип структуры, инкапсулирующий тип значения и допускающий пустое значение (`null`). Подробнее об этом — в главе 20.) Следовательно, в отсутствие ограничения такое присваивание было бы недопустимым, и код не подлежал бы компиляции. Это один из тех случаев, когда для обобщенного кода может оказаться очень важным различие между типами значений и ссылочными типами.

Ограничение типа значения является дополнением ограничения ссылочного типа. Оно просто гарантирует, что любой аргумент, обозначающий тип, должен быть типа значения, в том числе `struct` и `enum`. (В данном случае обнуляемый тип не относится к типу значения.) Ниже приведен пример наложения ограничения типа значения.

```
// Продемонстрировать наложение ограничения типа значения.

using System;

struct MyStruct {
  //...
}

class MyClass {
  // ...
}

class Test<T> where T : struct {
  T obj;

  public Test(T x) {
    obj = x;
  }

  // ...
}

class ValueConstraintDemo {
  static void Main() {

    // Оба следующих объявления вполне допустимы.
    Test<MyStruct> x = new Test<MyStruct>(new MyStruct());
```

```
    Test<int> y = new Test<int>(10);

    // А следующее объявление недопустимо!
// Test<MyClass> z = new Test<MyClass>(new MyClass());
  }
}
```

В этом примере кода класс `Test` объявляется следующим образом.

```
class Test<T> where T : struct {
```

На параметр типа `T` в классе `Test` накладывается ограничение `struct`, и поэтому к нему могут быть привязаны только аргументы типа значения. Это означает, что объявления `Test<MyStruct>` и `Test<int>` вполне допустимы, тогда как объявление `Test<MyClass>` недопустимо. Для того чтобы убедиться в этом, удалите символы комментария в начале последней строки приведенного выше кода и перекомпилируйте его. В итоге вы получите сообщение об ошибке во время компиляции.

Установление связи между двумя параметрами типа с помощью ограничения

Существует разновидность ограничения на базовый класс, позволяющая установить связь между двумя параметрами типа. В качестве примера рассмотрим следующее объявление обобщенного класса.

```
class Gen<T, V> where V : T {
```

В этом объявлении оператор `where` уведомляет компилятор о том, что аргумент типа, привязанный к параметру типа `V`, должен быть таким же, как и аргумент типа, привязанный к параметру типа `T`, или же наследовать от него. Если подобная связь отсутствует при объявлении объекта типа `Gen`, то во время компиляции возникнет ошибка. Такое ограничение на параметр типа называется *неприкрытым ограничением типа*. В приведенном ниже примере демонстрируется наложение этого ограничения.

```
// Установить связь между двумя параметрами типа.

using System;

class A {
  //...
}

class B : A {
  // ...
}

// Здесь параметр типа V должен наследовать от параметра типа T.
class Gen<T, V> where V : T {
  // ...
}

class NakedConstraintDemo {
  static void Main() {

    // Это объявление вполне допустимо, поскольку
```

```
    // класс B наследует от класса A.
    Gen<A, B> x = new Gen<A, B>();

    // А это объявление недопустимо, поскольку
    // класс A не наследует от класса B.
// Gen<B, A> y = new Gen<B, A>();

  }
}
```

Обратите внимание на то, что класс `B` наследует от класса `A`. Проанализируем далее оба объявления объектов класса `Gen` в методе `Main()`. Как следует из комментария к первому объявлению

```
Gen<A, B> x = new Gen<A, B>();
```

оно вполне допустимо, поскольку класс `B` наследует от класса `A`. Но второе объявление

```
// Gen<B, A> y = new Gen<B, A>();
```

недопустимо, поскольку класс `A` не наследует от класса `B`.

Применение нескольких ограничений

С параметром типа может быть связано несколько ограничений. В этом случае ограничения указываются списком через запятую. В этом списке первым должно быть указано ограничение `class` либо `struct`, если оно присутствует, или же ограничение на базовый класс, если оно накладывается. Указывать ограничения `class` или `struct` одновременно с ограничением на базовый класс не разрешается. Далее по списку должно следовать ограничение на интерфейс, а последним по порядку — ограничение `new()`. Например, следующее объявление считается вполне допустимым.

```
class Gen<T> where T : MyClass, IMyInterface, new() {
  // ...
```

В данном случае параметр типа `T` должен быть заменен аргументом типа, наследующим от класса `MyClass`, реализующим интерфейс `IMyInterface` и использующим конструктор без параметра.

Если же в обобщении используются два или более параметра типа, то ограничения на каждый из них накладываются с помощью отдельного оператора `where`, как в приведенном ниже примере.

```
// Использовать несколько операторов where.

using System;

// У класса Gen имеются два параметра типа, и на оба накладываются
// ограничения с помощью отдельных операторов where.
class Gen<T, V> where T : class
                where V : struct {
  T ob1;
  V ob2;

  public Gen(T t, V v) {
    ob1 = t;
```

```
    ob2 = v;
  }
}

class MultipleConstraintDemo {
  static void Main() {

    // Эта строка кода вполне допустима, поскольку
    // string — это ссылочный тип, а int — тип значения.
    Gen<string, int> obj = new Gen<string, int>("тест", 11);
    // А следующая строка кода недопустима, поскольку
    // bool не относится к ссылочному типу.
// Gen<bool, int> obj = new Gen<bool, int>(true, 11);
  }
}
```

В данном примере класс `Gen` принимает два аргумента с ограничениями, накладываемыми с помощью отдельных операторов `where`. Обратите особое внимание на объявление этого класса.

```
class Gen<T, V> where T : class
                where V : struct {
```

Как видите, один оператор `where` отделяется от другого только пробелом. Другие знаки препинания между ними не нужны и даже недопустимы.

Получение значения, присваиваемого параметру типа по умолчанию

Как упоминалось выше, при написании обобщенного кода иногда важно провести различие между типами значений и ссылочными типами. Такая потребность возникает, в частности, в том случае, если переменной параметра типа должно быть присвоено значение по умолчанию. Для ссылочных типов значением по умолчанию является `null`, для неструктурных типов значений — `0` или логическое значение `false`, если это тип `bool`, а для структур типа `struct` — объект соответствующей структуры с полями, установленными по умолчанию. В этой связи возникает вопрос: какое значение следует присваивать по умолчанию переменной параметра типа: `null`, `0` или нечто другое?

Например, если в следующем объявлении класса `Test`:

```
class Test<T> {
  T obj;
  // ...
```

переменной `obj` требуется присвоить значение по умолчанию, то какой из двух вариантов

```
obj = null; // подходит только для ссылочных типов
```

или

```
obj = 0; // подходит только для числовых типов и
         // перечислений, но не для структур
```

следует выбрать? Для разрешения этой дилеммы можно воспользоваться еще одной формой оператора `default`, приведенной ниже.

```
default(тип)
```

Эта форма оператора `default` пригодна для всех аргументов типа, будь то типы значений или ссылочные типы.

Ниже приведен короткий пример, демонстрирующий данную форму оператора `default`.

```
// Продемонстрировать форму оператора default.

using System;

class MyClass {
  //...
}

// Получить значение, присваиваемое параметру типа T по умолчанию.
class Test<T> {
  public T obj;

  public Test() {
    // Следующий оператор годится только для ссылочных типов.
// obj = null; // не годится

    // Следующий оператор годится только для типов значений.
// obj = 0; // не годится

    // А этот оператор годится как для ссылочных типов,
    // так и для типов значений.
    obj = default(T); // Годится!
  }

  // ...
}

class DefaultDemo {
  static void Main() {

    // Сконструировать объект класса Test, используя ссылочный тип.
    Test<MyClass> x = new Test<MyClass>();

    if(x.obj == null)
     Console.WriteLine("Переменная x.obj имеет пустое значение <null>.");

    // Сконструировать объект класса Test, используя тип значения.
    Test<int> y = new Test<int>();

    if(y.obj == 0)
      Console.WriteLine("Переменная y.obj имеет значение 0.");
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Переменная x.obj имеет пустое значение <null>.
Переменная y.obj имеет значение 0.
```

Обобщенные структуры

В C# разрешается создавать обобщенные структуры. Синтаксис для них такой же, как и для обобщенных классов. В качестве примера ниже приведена программа, в которой создается обобщенная структура XY для хранения координат X, Y.

```
// Продемонстрировать применение обобщенной структуры.

using System;

// Эта структура является обобщенной.
struct XY<T> {
  T x;
  T y;

  public XY(T a, T b) {
    x = a;
    y = b;
  }

  public T X {
    get { return x; }
    set { x = value; }
  }

  public T Y {
    get { return y; }
    set { y = value; }
  }
}

class StructTest {
  static void Main() {
    XY<int> xy = new XY<int>(10, 20);
    XY<double> xy2 = new XY<double>(88.0, 99.0);

     Console.WriteLine(xy.X + ", " + xy.Y);

     Console.WriteLine(xy2.X + ", " + xy2.Y);
  }
}
```

При выполнении этой программы получается следующий результат.

```
10, 20
88, 99
```

Как и на обобщенные классы, на обобщенные структуры могут накладываться ограничения. Например, на аргументы типа в приведенном ниже варианте структуры XY накладывается ограничение типа значения.

```
struct XY<T> where T : struct {
// ...
```

Создание обобщенного метода

Как следует из приведенных выше примеров, в методах, объявляемых в обобщенных классах, может использоваться параметр типа из данного класса, а следовательно, такие методы автоматически становятся обобщенными по отношению к параметру типа. Но помимо этого имеется возможность объявить обобщенный метод со своими собственными параметрами типа и даже создать обобщенный метод, заключенный в необобщенном классе.

Рассмотрим для начала простой пример. В приведенной ниже программе объявляется необобщенный класс `ArrayUtils`, а в нем — статический обобщенный метод `CopyInsert()`. Этот метод копирует содержимое одного массива в другой, вводя по ходу дела новый элемент в указанном месте. Метод `CopyInsert()` можно использовать вместе с массивами любого типа.

```
// Продемонстрировать применение обобщенного метода.

using System;

// Класс обработки массивов. Этот класс не является обобщенным.
class ArrayUtils {

  // Копировать массив, вводя по ходу дела новый элемент.
  // Этот метод является обобщенным.
  public static bool CopyInsert<T>(T e, uint idx,
                                   T[] src, T[] target) {

    // Проверить, насколько велик массив.
    if(target.Length < src.Length+1)
      return false;

    // Скопировать содержимое массива src в целевой массив,
    // попутно введя значение e по индексу idx.
    for(int i=0, j=0; i < src.Length; i++, j++) {
      if(i == idx) {
        target[j] = e;
        j++;
      }
      target[j] = src[i];
    }

    return true;
  }
}

class GenMethDemo {
  static void Main() {
    int[] nums = { 1, 2, 3 };
    int[] nums2 = new int[4];

    // Вывести содержимое массива nums.
    Console.Write("Содержимое массива nums: ");
    foreach(int x in nums)
```

```
      Console.Write(x + " ");

    Console.WriteLine();

    // Обработать массив типа int.
    ArrayUtils.CopyInsert(99, 2, nums, nums2);

    // Вывести содержимое массива nums2.
    Console.Write("Содержимое массива nums2: ");
    foreach(int x in nums2)
      Console.Write(x + " ");

    Console.WriteLine();

    // А теперь обработать массив строк, используя метод copyInsert.
    string[] strs = {"Обобщения", "весьма", "эффективны."};
    string[] strs2 = new string[4];

    // Вывести содержимое массива strs.
    Console.Write("Содержимое массива strs: ");
    foreach(string s in strs)
      Console.Write(s + " ");

    Console.WriteLine();

    // Ввести элемент в массив строк.
    ArrayUtils.CopyInsert("в C#", 1, strs, strs2);

    // Вывести содержимое массива strs2.
    Console.Write("Содержимое массива strs2: ");
    foreach(string s in strs2)
      Console.Write(s + " ");

    Console.WriteLine();

    // Этот вызов недопустим, поскольку первый аргумент
    // относится к типу double, а третий и четвертый
    // аргументы обозначают элементы массивов типа int.
// ArrayUtils.CopyInsert(0.01, 2, nums, nums2);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Содержимое массива nums: 1 2 3
Содержимое массива nums2: 1 2 99 3
Содержимое массива strs: Обобщения весьма эффективны.
Содержимое массива strs2: Обобщения в C# весьма эффективны.
```

Внимательно проанализируем метод `CopyInsert()`. Прежде всего обратите внимание на объявление этого метода в следующей строке кода.

```
public static bool CopyInsert<T>(T e, uint idx,
                                 T[] src, T[] target) {
```

Параметр типа объявляется *после* имени метода, но *перед* списком его параметров. Обратите также внимание на то, что метод `CopyInsert()` является статическим, что позволяет вызывать его независимо от любого объекта. Следует, однако, иметь в виду, что обобщенные методы могут быть либо статическими, либо нестатическими. В этом отношении для них не существует никаких ограничений.

Далее обратите внимание на то, что метод `CopyInsert()` вызывается в методе `Main()` с помощью обычного синтаксиса и без указания аргументов типа. Дело в том, что типы аргументов различаются автоматически, а тип `T` соответственно подстраивается. Этот процесс называется *выводимостью типов*. Например, в первом вызове данного метода

```
ArrayUtils.CopyInsert(99, 2, nums, nums2);
```

тип `T` становится типом `int`, поскольку числовое значение `99` и элементы массивов `nums` и `nums2` относятся к типу `int`. А во втором вызове данного метода используются строковые типы, и поэтому тип `T` заменяется типом `string`.

А теперь обратите внимание на приведенную ниже закомментированную строку кода.

```
// ArrayUtils.CopyInsert(0.01, 2, nums, nums2);
```

Если удалить символы комментария в начале этой строки кода и затем попытаться перекомпилировать программу, то будет получено сообщение об ошибке. Дело в том, что первый аргумент в данном вызове метода `CopyInsert()` относится к типу `double`, а третий и четвертый аргументы обозначают элементы массивов `nums` и `nums2` типа `int`. Но все эти аргументы типа должны заменить один и тот же параметр типа `T`, а это приведет к несоответствию типов и, как следствие, к ошибке во время компиляции. Подобная возможность соблюдать типовую безопасность относится к одним из самых главных преимуществ обобщенных методов.

Синтаксис объявления метода `CopyInsert()` может быть обобщен. Ниже приведена общая форма объявления обобщенного метода.

```
возвращаемый_тип имя_метода<список_параметров_типа>(список_параметров) {// ...
```

В любом случае *список_параметров_типа* обозначает разделяемый запятой список параметров типа. Обратите внимание на то, что в объявлении обобщенного метода список параметров типа следует *после* имени метода.

Вызов обобщенного метода с явно указанными аргументами типа

В большинстве случаев неявной выводимости типов оказывается достаточно для вызова обобщенного метода, тем не менее аргументы типа могут быть указаны явным образом. Для этого достаточно указать аргументы типа после имени метода при его вызове. В качестве примера ниже приведена строка кода, в которой метод `CopyInsert()` вызывается с явно указываемым аргументом типа `string`.

```
ArrayUtils.CopyInsert<string>("B C#", 1, strs, strs2);
```

Тип передаваемых аргументов необходимо указывать явно в том случае, если компилятор не сможет вывести тип параметра `T` или если требуется отменить выводимость типов.

Применение ограничений в обобщенных методах

На аргументы обобщенного метода можно наложить ограничения, указав их после списка параметров. В качестве примера ниже приведен вариант метода `CopyInsert()` для обработки данных только ссылочных типов.

```
public static bool CopyInsert<T>(T e, uint idx,
                                 T[] src, T[] target) where T : class {
```

Если попробовать применить этот вариант в предыдущем примере программы обработки массивов, то приведенный ниже вызов метода `CopyInsert()` не будет скомпилирован, поскольку `int` является типом значения, а не ссылочным типом.

```
// Теперь неправильно, поскольку параметр T должен быть ссылочного типа!
ArrayUtils.CopyInsert(99, 2, nums, nums2); // Теперь недопустимо!
```

Обобщенные делегаты

Как и методы, делегаты также могут быть обобщенными. Ниже приведена общая форма объявления обобщенного делегата.

```
delegate возвращаемый_тип имя_делегата<список_параметров_типа>(список_аргументов);
```

Обратите внимание на расположение списка параметров типа. Он следует непосредственно после имени делегата. Преимущество обобщенных делегатов заключается в том, что их допускается определять в типизированной обобщенной форме, которую можно затем согласовать с любым совместимым методом.

В приведенном ниже примере программы демонстрируется применение делегата `SomeOp` с одним параметром типа `T`. Этот делегат возвращает значение типа `T` и принимает аргумент типа `T`.

```
// Простой пример обобщенного делегата.

using System;

// Объявить обобщенный делегат.
delegate T SomeOp<T>(T v);

class GenDelegateDemo {
  // Возвратить результат суммирования аргумента.
  static int Sum(int v) {
    int result = 0;
    for(int i=v; i>0; i--)
      result += i;

    return result;
  }

  // Возвратить строку, содержащую обратное значение аргумента.
  static string Reflect(string str) {
    string result = "";

    foreach(char ch in str)
```

```
      result = ch + result;

    return result;
  }

  static void Main() {
    // Сконструировать делегат типа int.
    SomeOp<int> intDel = Sum;
    Console.WriteLine(intDel(3));

    // Сконструировать делегат типа string.
    SomeOp<string> strDel = Reflect;
    Console.WriteLine(strDel("Привет"));
  }
}
```

Эта программа дает следующий результат.

```
6
тевирП
```

Рассмотрим эту программу более подробно. Прежде всего обратите внимание на следующее объявление делегата `SomeOp`.

```
delegate T SomeOp<T>(T v);
```

Как видите, тип `T` может служить в качестве возвращаемого типа, несмотря на то, что параметр типа `T` указывается после имени делегата `SomeOp`.

Далее в классе `GenDelegateDemo` объявляются методы `Sum()` и `Reflect()`, как показано ниже.

```
static int Sum(int v) {

static string Reflect(string str) {
```

Метод `Sum()` возвращает результат суммирования целого значения, передаваемого в качестве аргумента, а метод `Reflect()` — символьную строку, которая получается обращенной по отношению к строке, передаваемой в качестве аргумента.

В методе `Main()` создается экземпляр `intDel` делегата, которому присваивается ссылка на метод `Sum()`.

```
SomeOp<int> intDel = Sum;
```

Метод `Sum()` принимает аргумент типа `int` и возвращает значение типа `int`, поэтому он совместим с целочисленным экземпляром делегата `SomeOp`.

Аналогичным образом создается экземпляр `strDel` делегата, которому присваивается ссылка на метод `Reflect()`.

```
SomeOp<string> strDel = Reflect;
```

Метод `Reflect()` принимает аргумент типа `string` и возвращает результат типа `string`, поэтому он совместим со строковым экземпляром делегата `SomeOp`.

В силу присущей обобщениям типовой безопасности обобщенным делегатам нельзя присваивать несовместимые методы. Так, следующая строка кода оказалась бы ошибочной в рассматриваемой здесь программе.

```
SomeOp<int> intDel = Reflect; // Ошибка!
```

Ведь метод `Reflect()` принимает аргумент типа `string` и возвращает результат типа `string`, а следовательно, он несовместим с целочисленным экземпляром делегата `SomeOp`.

Обобщенные интерфейсы

Помимо обобщенных классов и методов, в C# допускаются обобщенные интерфейсы. Такие интерфейсы указываются аналогично обобщенным классам. Ниже приведен измененный вариант примера из главы 12, демонстрирующего интерфейс `ISeries`. (Напомним, что `ISeries` является интерфейсом для класса, генерирующего последовательный ряд числовых значений.) Тип данных, которым оперирует этот интерфейс, теперь определяется параметром типа.

```
// Продемонстрировать применение обобщенного интерфейса.

using System;

public interface ISeries<T> {
  T GetNext();// возвратить следующее по порядку число
  void Reset(); // генерировать ряд последовательных чисел с самого начала
  void SetStart(T v); // задать начальное значение
}

// Реализовать интерфейс ISeries.
class ByTwos<T> : ISeries<T> {
  T start;
  T val;

  // Этот делегат определяет форму метода, вызываемого для генерирования
  // очередного элемента в ряду последовательных значений.
  public delegate T IncByTwo(T v);
  // Этой ссылке на делегат будет присвоен метод,
  // передаваемый конструктору класса ByTwos.
  IncByTwo incr;
  public ByTwos(IncByTwo incrMeth) {
    start = default(T);
    val = default(T);
    incr = incrMeth;
  }

  public T GetNext() {
    val = incr(val);
    return val;
  }

  public void Reset() {
    val = start;
  }

  public void SetStart(T v) {
    start = v;
    val = start;
```

```
  }
}

class ThreeD {
  public int x, y, z;
    public ThreeD(int a, int b, int c) {
    x = a;
    y = b;
    z = c;
  }
}

class GenIntfDemo {
  // Определить метод увеличения на два каждого
  // последующего значения типа int.
  static int IntPlusTwo(int v) {
  return v + 2;
  }

  // Определить метод увеличения на два каждого
  // последующего значения типа double.
  static double DoublePlusTwo(double v) {
  return v + 2.0;
  }

  // Определить метод увеличения на два каждого
  // последующего значения координат объекта типа ThreeD.
  static ThreeD ThreeDPlusTwo(ThreeD v) {
    if(v==null) return new ThreeD(0, 0, 0);
    else return new ThreeD(v.x + 2, v.y + 2, v.z + 2);
  }

  static void Main() {

    // Продемонстрировать генерирование
    // последовательного ряда значений типа int.
    ByTwos<int> intBT = new ByTwos<int>(IntPlusTwo);

    for(int i=0; i < 5; i++)
      Console.Write(intBT.GetNext() + " ");

    Console.WriteLine();

    // Продемонстрировать генерирование
    // последовательного ряда значений типа double.
    ByTwos<double> dblBT =
                  new ByTwos<double>(DoublePlusTwo);
    dblBT.SetStart(11.4);

    for(int i=0; i < 5; i++)
      Console.Write(dblBT.GetNext() + " ");

    Console.WriteLine();
```

```
    // Продемонстрировать генерирование последовательного ряда
    // значений координат объекта типа ThreeD.
    ByTwos<ThreeD> ThrDBT = new ByTwos<ThreeD>(ThreeDPlusTwo);

    ThreeD coord;

    for(int i=0; i < 5; i++) {
      coord = ThrDBT.GetNext();
      Console.Write(coord.x + "," +
                    coord.y + "," +
                    coord.z + " ");
    }

    Console.WriteLine();
  }
}
```

Этот код выдает следующий результат.

```
2 4 6 8 10
13.4 15.4 17.4 19.4 21.4
0,0,0 2,2,2 4,4,4 6,6,6 8,8,8
```

В данном примере кода имеется ряд любопытных моментов. Прежде всего обратите внимание на объявление интерфейса `ISeries` в следующей строке кода.

```
public interface ISeries<T> {
```

Как упоминалось выше, для объявления обобщенного интерфейса используется такой же синтаксис, что и для объявления обобщенного класса.

А теперь обратите внимание на следующее объявление класса `ByTwos`, реализующего интерфейс `Iseries`.

```
class ByTwos<T> : ISeries<T> {
```

Параметр типа `T` указывается не только при объявлении класса `ByTwos`, но и при объявлении интерфейса `ISeries`. И это очень важно. Ведь класс, реализующий обобщенный вариант интерфейса, сам должен быть обобщенным. Так, приведенное ниже объявление недопустимо, поскольку параметр типа `T` не определен.

```
class ByTwos : ISeries<T> { // Неверно!
```

Аргумент типа, требующийся для интерфейса `ISeries`, должен быть передан классу `ByTwos`. В противном случае интерфейс никак не сможет получить аргумент типа.

Далее переменные, хранящие текущее значение в последовательном ряду (`val`) и его начальное значение (`start`), объявляются как объекты обобщенного типа `T`. После этого объявляется делегат `IncByTwo`. Этот делегат определяет форму метода, используемого для увеличения на два значения, хранящегося в объекте типа `T`. Для того чтобы в классе `ByTwos` могли обрабатываться данные любого типа, необходимо каким-то образом определить порядок увеличения на два значения каждого типа данных. Для этого конструктору класса `ByTwos` передается ссылка на метод, выполняющий увеличение на два. Эта ссылка хранится в переменной экземпляра делегата `incr`. Когда требуется сгенерировать следующий элемент в последовательном ряду, этот метод вызывается с помощью делегата `incr`.

А теперь обратите внимание на класс ThreeD. В этом классе инкапсулируются координаты трехмерного пространства (X,Z,Y). Его назначение — продемонстрировать обработку данных типа класса в классе ByTwos.

Далее в классе GenIntfDemo объявляются три метода увеличения на два для объектов типа int, double и ThreeD. Все эти методы передаются конструктору класса ByTwos при создании объектов соответствующих типов. Обратите особое внимание на приведенный ниже метод ThreeDPlusTwo().

```
// Определить метод увеличения на два каждого
// последующего значения координат объекта типа ThreeD.
static ThreeD ThreeDPlusTwo(ThreeD v) {
  if(v==null) return new ThreeD(0, 0, 0);
  else return new ThreeD(v.x + 2, v.y + 2, v.z + 2);
}
```

В этом методе сначала проверяется, содержит ли переменная экземпляра v пустое значение (null). Если она содержит это значение, то метод возвращает новый объект типа ThreeD со всеми обнуленными полями координат. Ведь дело в том, что переменной v по умолчанию присваивается значение типа default(T) в конструкторе класса ByTwos. Это значение оказывается по умолчанию нулевым для типов значений и пустым для типов ссылок на объекты. Поэтому если предварительно не был вызван метод SetStart(), то перед первым увеличением на два переменная v будет содержать пустое значение вместо ссылки на объект. Это означает, что для первого увеличения на два требуется новый объект.

На параметр типа в обобщенном интерфейсе могут накладываться ограничения таким же образом, как и в обобщенном классе. В качестве примера ниже приведен вариант объявления интерфейса ISeries с ограничением на использование только ссылочных типов.

```
public interface ISeries<T> where T : class {
```

Если реализуется именно такой вариант интерфейса ISeries, в реализующем его классе следует указать то же самое ограничение на параметр типа T, как показано ниже.

```
class ByTwos<T> : ISeries<T> where T : class {
```

В силу ограничения ссылочного типа этот вариант интерфейса ISeries нельзя применять к типам значений. Поэтому если реализовать его в рассматриваемом здесь примере программы, то допустимым окажется только объявление ByTwos<ThreeD>, но не объявления ByTwos<int> и ByTwos<double>.

Сравнение экземпляров параметра типа

Иногда возникает потребность сравнить два экземпляра параметра типа. Допустим, что требуется написать обобщенный метод IsIn(), возвращающий логическое значение true, если в массиве содержится некоторое значение. Для этой цели сначала можно попробовать сделать следующее.

```
// Не годится!
public static bool IsIn<T>(T what, T[] obs) {
  foreach(T v in obs)
```

```
    if(v == what) // Ошибка!
      return true;

  return false;
}
```

К сожалению, эта попытка не пройдет. Ведь параметр `T` относится к обобщенному типу, и поэтому компилятору не удастся выяснить, как сравнивать два объекта. Требуется ли для этого поразрядное сравнение или же только сравнение отдельных полей? А возможно, сравнение ссылок? Вряд ли компилятор сможет найти ответы на эти вопросы. Правда, из этого положения все же имеется выход.

Для сравнения двух объектов параметра обобщенного типа они должны реализовывать интерфейс `IComparable` или `IComparable<T>` и/или интерфейс `IEquatable<T>`. В обоих вариантах интерфейса `IComparable` для этой цели определен метод `CompareTo()`, а в интерфейсе `IEquatable<T>` — метод `Equals()`. Разновидности интерфейса `IComparable` предназначены для применения в тех случаях, когда требуется определить относительный порядок следования двух объектов. А интерфейс `IEquatable` служит для определения равенства двух объектов. Все эти интерфейсы определены в пространстве имен `System` и реализованы во встроенных в C# типах данных, включая `int`, `string` и `double`. Но их нетрудно реализовать и для собственных создаваемых классов. Итак, начнем с обобщенного интерфейса `IEquatable<T>`.

Интерфейс `IEquatable<T>` объявляется следующим образом.

```
public interface IEquatable<T>
```

Сравниваемый тип данных передается ему в качестве аргумента типа `T`. В этом интерфейсе определяется метод `Equals()`, как показано ниже.

```
bool Equals(T other)
```

В этом методе сравниваются вызывающий объект и другой объект, определяемый параметром *other*. В итоге возвращается логическое значение `true`, если оба объекта равны, а иначе — логическое значение `false`.

В ходе реализации интерфейса `IEquatable<T>` обычно требуется также переопределять методы `GetHashCode()` и `Equals(Object)`, определенные в классе `Object`, чтобы они оказались совместимыми с конкретной реализацией метода `Equals()`. Ниже приведен пример программы, в которой демонстрируется исправленный вариант упоминавшегося ранее метода `IsIn()`.

```
// Требуется обобщенный интерфейс IEquatable<T>.
public static bool IsIn<T>(T what, T[] obs) where T : IEquatable<T> {
  foreach(T v in obs)
    if(v.Equals(what)) // Применяется метод Equals().
      return true;

    return false;
}
```

Обратите внимание в приведенном выше примере на применение следующего ограничения.

```
where T : IEquatable<T>
```

Это ограничение гарантирует, что только те типы, в которых реализован интерфейс `IEquatable`, являются действительными аргументами типа для метода `IsIn()`. Вну-

три этого метода применяется метод `Equals()`, который определяет равенство одного объекта другому.

Для определения относительного порядка следования двух элементов применяется интерфейс `IComparable`. У этого интерфейса имеются две формы: обобщенная и необобщенная. Обобщенная форма данного интерфейса обладает преимуществом обеспечения типовой безопасности, и поэтому мы рассмотрим здесь именно ее. Обобщенный интерфейс `IComparable<T>` объявляется следующим образом.

```
public interface IComparable<T>
```

Сравниваемый тип данных передается ему в качестве аргумента типа `T`. В этом интерфейсе определяется метод `CompareTo()`, как показано ниже.

```
int CompareTo(T other)
```

В этом методе сравниваются вызывающий объект и другой объект, определяемый параметром *other*. В итоге возвращается нуль, если вызывающий объект оказывается больше, чем объект *other*; и отрицательное значение, если вызывающий объект оказывается меньше, чем объект *other*.

Для того чтобы воспользоваться методом `CompareTo()`, необходимо указать ограничение, которое требуется наложить на аргумент типа для реализации обобщенного интерфейса `IComparable<T>`. А затем достаточно вызвать метод `CompareTo()`, чтобы сравнить два экземпляра параметра типа.

Ниже приведен пример применения обобщенного интерфейса `IComparable<T>`. В этом примере вызывается метод `InRange()`, возвращающий логическое значение `true`, если объект оказывается среди элементов отсортированного массива.

```
// Требуется обобщенный интерфейс IComparable<T>. В данном методе
// предполагается, что массив отсортирован. Он возвращает логическое
// значение true, если значение параметра what оказывается среди элементов
// массива, передаваемых параметру obs.
public static bool InRange<T>(T what, T[] obs) where T : IComparable<T> {
  if(what.CompareTo(obs[0]) < 0 ||
     what.CompareTo(obs[obs.Length-1]) > 0) return false;
  return true;
}
```

В приведенном ниже примере программы демонстрируется применение обоих методов `IsIn()` и `InRange()` на практике.

```
// Продемонстрировать применение обобщенных
// интерфейсов IComparable<T> и IEquatable<T>.

using System;

// Теперь в классе MyClass реализуются обобщенные
// интерфейсы IComparable<T> и IEquatable<T>.
class MyClass : IComparable<MyClass>, IEquatable<MyClass> {
  public int Val;

  public MyClass(int x) { Val = x; }

  // Реализовать обобщенный интерфейс IComparable<T>.
  public int CompareTo(MyClass other) {
```

```
    return Val - other.Val; // Now, no cast is needed.
  }

  // Реализовать обобщенный интерфейс IEquatable<T>.
  public bool Equals(MyClass other) {
    return Val == other.Val;
  }

  // Переопределить метод Equals(Object).
  public override bool Equals(Object obj) {
    if(obj is MyClass)
      return Equals((MyClass) obj);
    return false;
  }

  // Переопределить метод GetHashCode().
  public override int GetHashCode() {
    return Val.GetHashCode();
  }
}

class CompareDemo {

  // Требуется обобщенный интерфейс IEquatable<T>.
  public static bool IsIn<T>(T what, T[] obs) where T : IEquatable<T> {
    foreach(T v in obs)
      if(v.Equals(what)) // Применяется метод Equals()
        return true;

    return false;
  }

  // Требуется обобщенный интерфейс IComparable<T>. В данном методе
  // предполагается, что массив отсортирован. Он возвращает логическое
  // значение true, если значение параметра what оказывается среди элементов
  // массива, передаваемых параметру obs.
  public static bool InRange<T>(T what, T[] obs) where T : IComparable<T> {
    if(what.CompareTo(obs[0]) < 0 ||
      what.CompareTo(obs[obs.Length-1]) > 0) return false;
    return true;
  }

  // Продемонстрировать операции сравнения.
  static void Main() {

    // Применить метод IsIn() к данным типа int.
    int[] nums = { 1, 2, 3, 4, 5 };

    if(IsIn(2, nums))
      Console.WriteLine("Найдено значение 2.");

     if(IsIn(99, nums))
       Console.WriteLine("Не подлежит выводу.");
```

```
    // Применить метод IsIn() к объектам класса MyClass.
    MyClass[] mcs = { new MyClass(1), new MyClass(2),
                      new MyClass(3), new MyClass(4) };

    if(IsIn(new MyClass(3), mcs))
      Console.WriteLine("Найден объект MyClass(3).");

    if(IsIn(new MyClass(99), mcs))
      Console.WriteLine("Не подлежит выводу.");

    // Применить метод InRange() к данным типа int.
    if(InRange(2, nums))
      Console.WriteLine("Значение 2 находится в границах массива nums.");
    if(InRange(1, nums))
      Console.WriteLine("Значение 1 находится в границах массива nums.");
    if(InRange(5, nums))
      Console.WriteLine("Значение 5 находится в границах массива nums.");
    if(!InRange(0, nums))
      Console.WriteLine("Значение 0 НЕ находится в границах массива nums.");
    if(!InRange(6, nums))
      Console.WriteLine("Значение 6 НЕ находится в границах массива nums.");

    // Применить метод InRange() к объектам класса MyClass.
    if(InRange(new MyClass(2), mcs))
      Console.WriteLine("Объект MyClass(2) находится в границах массива nums.");
    if(InRange(new MyClass(1), mcs))
      Console.WriteLine("Объект MyClass(1) находится " +
                        "в границах массива nums.");
    if(InRange(new MyClass(4), mcs))
      Console.WriteLine("Объект MyClass(4) находится " +
                        "в границах массива nums.");
    if(!InRange(new MyClass(0), mcs))
      Console.WriteLine("Объект MyClass(0) НЕ " +
                        "находится в границах массива nums.");
    if(!InRange(new MyClass(5), mcs))
      Console.WriteLine("Объект MyClass(5) НЕ " +
                        "находится в границах массива nums.");
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Найдено значение 2.
Найден объект MyClass(3).
Значение 2 находится в границах массива nums.
Значение 1 находится в границах массива nums.
Значение 5 находится в границах массива nums.
Значение 0 НЕ находится в границах массива nums
Значение 6 НЕ находится в границах массива nums
Объект MyClass(2) находится в границах массива nums.
Объект MyClass(1) находится в границах массива nums.
Объект MyClass(4) находится в границах массива nums.
Объект MyClass(0) НЕ находится в границах массива nums.
Объект MyClass(5) НЕ находится в границах массива nums.
```

ПРИМЕЧАНИЕ

Если параметр типа обозначает ссылку или ограничение на базовый класс, то к экземплярам объектов, определяемых таким параметром типа, можно применять операторы == и !=, хотя они проверяют на равенство только ссылки. А для сравнения значений придется реализовать интерфейс `IComparable` или же обобщенные интерфейсы `IComparable<T>` и `IEquatable<T>`.

Иерархии обобщенных классов

Обобщенные классы могут входить в иерархию классов аналогично необобщенным классам. Следовательно, обобщенный класс может действовать как базовый или производный класс. Главное отличие между иерархиями обобщенных и необобщенных классов заключается в том, что в первом случае аргументы типа, необходимые обобщенному базовому классу, должны передаваться всеми производными классами вверх по иерархии аналогично передаче аргументов конструктора.

Применение обобщенного базового класса

Ниже приведен простой пример иерархии, в которой используется обобщенный базовый класс.

```
// Простая иерархия обобщенных классов.
using System;

// Обобщенный базовый класс.
class Gen<T> {
  T ob;
  public Gen(T o) {
    ob = o;
  }

  // Возвратить значение переменной ob.
  public T GetOb() {
    return ob;
  }
}

// Класс, производный от класса Gen.
class Gen2<T> : Gen<T> {
  public Gen2(T o) : base(o) {
   // ...
  }
}

class GenHierDemo {
  static void Main() {
    Gen2<string> g2 = new Gen2<string>("Привет");
      Console.WriteLine(g2.GetOb());
  }
}
```

В этой иерархии класс Gen2 наследует от обобщенного класса Gen. Обратите внимание на объявление класса Gen2 в следующей строке кода.

```
class Gen2<T> : Gen<T> {
```

Параметр типа T указывается в объявлении класса Gen2 и в то же время передается классу Gen. Это означает, что любой тип, передаваемый классу Gen2, будет передаваться также классу Gen. Например, в следующем объявлении:

```
Gen2<string> g2 = new Gen2<string>("Привет");
```

параметр типа string передается классу Gen. Поэтому переменная ob в той части класса Gen2, которая относится к классу Gen, будет иметь тип string.

Обратите также внимание на то, что в классе Gen2 параметр типа T не используется, а только передается вверх по иерархии базовому классу Gen. Это означает, что в производном классе следует непременно указывать параметры типа, требующиеся его обобщенному базовому классу, даже если этот производный класс не обязательно должен быть обобщенным.

Разумеется, в производный класс можно свободно добавлять его собственные параметры типа, если в этом есть потребность. В качестве примера ниже приведен вариант предыдущей иерархии классов, где в класс Gen2 добавлен собственный параметр типа.

```
// Пример добавления собственных параметров типа в производный класс.

using System;

// Обобщенный базовый класс.
class Gen<T> {
  T ob; // объявить переменную типа T

  // Передать конструктору ссылку типа T.
  public Gen(T o) {
    ob = o;
  }

  // Возвратить значение переменной ob.
  public T GetOb() {
    return ob;
  }
}

//  Класс, производный от класса Gen. В этом классе
// определяется второй параметр типа V.
class Gen2<T, V> : Gen<T> {
  V ob2;

  public Gen2(T o, V o2) : base(o) {
    ob2 = o2;
  }

  public V GetObj2() {
    return ob2;
  }
```

```
}

// Создать объект класса Gen2.
class GenHierDemo2 {
  static void Main() {

    // Создать объект класса Gen2 с параметрами
    // типа string и int.
    Gen2<string, int> x =
      new Gen2<string, int>("Значение равно: ", 99);
    Console.Write(x.GetOb());
    Console.WriteLine(x.GetObj2());
  }
}
```

Обратите внимание на приведенное ниже объявление класса `Gen2` в данном варианте иерархии классов.

```
class Gen2<T, V> : Gen<T> {
```

В этом объявлении `T` — это тип, передаваемый базовому классу `Gen`; а `V` — тип, характерный только для производного класса `Gen2`. Он служит для объявления объекта `ob2` и в качестве типа, возвращаемого методом `GetObj2()`. В методе `Main()` создается объект класса `Gen2` с параметром `T` типа `string` и параметром `V` типа `int`. Поэтому код из приведенного выше примера дает следующий результат.

```
Значение равно: 99
```

Обобщенный производный класс

Необобщенный класс может быть вполне законно базовым для обобщенного производного класса. В качестве примера рассмотрим следующую программу.

```
// Пример необобщенного класса в качестве базового для
// обобщенного производного класса.

using System;

// Необобщенный базовый класс.
class NonGen {
  int num;

  public NonGen(int i) {
    num = i;
  }

  public int GetNum() {
    return num;
  }
}

// Обобщенный производный класс.
class Gen<T> : NonGen {
  T ob;
```

```
  public Gen(T o, int i) : base (i) {
    ob = o;
  }

  // Возвратить значение переменной ob.
  public T GetOb() {
    return ob;
  }
}

// Создать объект класса Gen.
class HierDemo3 {
  static void Main() {

    // Создать объект класса Gen с параметром типа string.
    Gen<String> w = new Gen<String>("Привет", 47);

    Console.Write(w.GetOb() + " ");
    Console.WriteLine(w.GetNum());
  }
}
```

Эта программа дает следующий результат.

```
Привет 47
```

В данной программе обратите внимание на то, как класс `Gen` наследует от класса `NonGen` в следующем объявлении.

```
class Gen<T> : NonGen {
```

Класс `NonGen` не является обобщенным, и поэтому аргумент типа для него не указывается. Это означает, что параметр `T`, указываемый в объявлении обобщенного производного класса `Gen`, не требуется для указания базового класса `NonGen` и даже не может в нем использоваться. Следовательно, класс `Gen` наследует от класса `NonGen` обычным образом, т.е. без выполнения каких-то особых условий.

Переопределение виртуальных методов в обобщенном классе

В обобщенном классе виртуальный метод может быть переопределен таким же образом, как и любой другой метод. В качестве примера рассмотрим следующую программу, в которой переопределяется виртуальный метод `GetOb()`.

```
// Пример переопределения виртуального метода в обобщенном классе.

using System;

// Обобщенный базовый класс.
class Gen<T> {
  protected T ob;

  public Gen(T o) {
    ob = o;
  }

  // Возвратить значение переменной ob. Этот метод является виртуальным.
```

```
  public virtual T GetOb() {
    Console.Write("Метод GetOb() из класса Gen" + " возвращает результат: ");
    return ob;
  }
}

// Класс, производный от класса Gen. В этом классе
// переопределяется метод GetOb().
class Gen2<T> : Gen<T> {

  public Gen2(T o) : base(o) { }

  // Переопределить метод GetOb().
  public override T GetOb() {
    Console.Write("Метод GetOb() из класса Gen2" + " возвращает результат: ");
    return ob;
  }
}

// Продемонстрировать переопределение метода в обобщенном классе.
class OverrideDemo {
  static void Main() {

    // Создать объект класса Gen с параметром типа int.
    Gen<int> iOb = new Gen<int>(88);

    // Здесь вызывается вариант метода GetOb() из класса Gen.
    Console.WriteLine(iOb.GetOb());

    // А теперь создать объект класса Gen2 и присвоить
    // ссылку на него переменной iOb типа Gen<int>.
    iOb = new Gen2<int>(99);

    // Здесь вызывается вариант метода GetOb() из класса Gen2.
    Console.WriteLine(iOb.GetOb());
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Метод GetOb() из класса Gen возвращает результат: 88
Метод GetOb() из класса Gen2 возвращает результат: 99
```

Как следует из результата выполнения приведенной выше программы, переопределяемый вариант метода `GetOb()` вызывается для объекта типа `Gen2`, а его вариант из базового класса вызывается для объекта типа `Gen`.

Обратите внимание на следующую строку кода.

```
iOb = new Gen2<int>(99);
```

Такое присваивание вполне допустимо, поскольку `iOb` является переменной типа `Gen<int>`. Следовательно, она может ссылаться на любой объект типа `Gen<int>` или же объект класса, производного от `Gen<int>`, включая и `Gen2<int>`. Разумеется, переменную `iOb` нельзя использовать, например, для ссылки на объект типа `Gen2<int>`, поскольку это может привести к несоответствию типов.

Перегрузка методов с несколькими параметрами типа

Методы, параметры которых объявляются с помощью параметров типа, могут быть перегружены. Но правила их перегрузки упрощаются по сравнению с методами без параметров типа. Как правило, метод, в котором параметр типа служит для указания типа данных параметра этого метода, может быть перегружен при условии, что сигнатуры обоих его вариантов отличаются. Это означает, что оба варианта перегружаемого метода должны отличаться по типу или количеству их параметров. Но типовые различия должны определяться не по параметру обобщенного типа, а исходя из аргумента типа, подставляемого вместо параметра типа при конструировании объекта этого типа. Следовательно, метод с параметрами типа может быть перегружен таким образом, что он окажется пригодным не для всех возможных случаев, хотя и будет выглядеть верно.

В качестве примера рассмотрим следующий обобщенный класс.

```
// Пример неоднозначности, к которой может привести
// перегрузка методов с параметрами типа.
//
// Этот код не подлежит компиляции.

using System;

// Обобщенный класс, содержащий метод Set(), перегрузка
// которого может привести к неоднозначности.
class Gen<T, V> {
  T ob1;
  V ob2;

  // ...
  // В некоторых случаях эти два метода не будут
  // отличаться своими параметрами типа.
  public void Set(T o) {
    ob1 = o;
  }

  public void Set(V o) {
    ob2 = o;
  }
}

class AmbiguityDemo {
  static void Main() {
    Gen<int, double> ok = new Gen<int, double>();
    Gen<int, int> notOK = new Gen<int, int>();
    ok.Set(10); // верно, поскольку аргументы типа отличаются
    notOK.Set(10); // неоднозначно, поскольку аргументы ничем не отличаются!
  }
}
```

Рассмотрим приведенный выше код более подробно. Прежде всего обратите внимание на то, что класс `Gen` объявляется с двумя параметрами типа: `T` и `V`. В классе `Gen` метод `Set()` перегружается по параметрам типа `T` и `V`, как показано ниже.

```
public void Set(T o) {
  ob1 = o;
}

public void Set(V o) {
  ob2 = o;
}
```

Такой подход кажется вполне обоснованным, поскольку типы `T` и `V` ничем внешне не отличаются. Но подобная перегрузка таит в себе потенциальную неоднозначность.

При таком объявлении класса `Gen` не соблюдается никаких требований к различению типов `T` и `V`. Например, нет ничего принципиально неправильного в том, что объект класса `Gen` будет сконструирован так, как показано ниже.

```
Gen<int, int> notOK = new Gen<int, int>();
```

В данном случае оба типа, `T` и `V`, заменяются типом `int`. В итоге оба варианта метода `Set()` оказываются совершенно одинаковыми, что, разумеется, приводит к ошибке. Следовательно, при последующей попытке вызвать метод `Set()` для объекта `notOK` в методе `Main()` появится сообщение об ошибке вследствие неоднозначности во время компиляции.

Как правило, методы с параметрами типа перегружаются при условии, что объект конструируемого типа не приводит к конфликту. Следует, однако, иметь в виду, что ограничения на типы не учитываются при разрешении конфликтов, возникающих при перегрузке методов. Поэтому ограничения на типы нельзя использовать для исключения неоднозначности. Конструкторы, операторы и индексаторы с параметрами типа могут быть перегружены аналогично конструкторам по тем же самым правилам.

Ковариантность и контравариантность в параметрах обобщенного типа

В главе 15 ковариантность и контравариантность были рассмотрены в связи с необобщенными делегатами. Эта форма ковариантности и контравариантности по-прежнему поддерживается в C#, поскольку она очень полезна. Но в версии C# 4.0 возможности ковариантности и контравариантности были расширены до параметров обобщенного типа, применяемых в обобщенных интерфейсах и делегатах. Ковариантность и контравариантность применяется, главным образом, для рационального разрешения особых ситуаций, возникающих в связи с применением обобщенных интерфейсов и делегатов, определенных в среде .NET Framework. И поэтому некоторые интерфейсы и делегаты, определенные в библиотеке, были обновлены, чтобы использовать ковариантность и контравариантность параметров типа. Разумеется, преимуществами ковариантности и контравариантности можно также воспользоваться в интерфейсах и делегатах, создаваемых собственными силами. В этом разделе механизмы ковариантности и контравариантности параметров типа поясняются на конкретных примерах.

Применение ковариантности в обобщенном интерфейсе

Применительно к обобщенному интерфейсу ковариантность служит средством, разрешающим методу возвращать тип, производный от класса, указанного в параметре типа. В прошлом возвращаемый тип должен был в точности соответствовать

параметру типа в силу строгой проверки обобщений на соответствие типов. Ковариантность смягчает это строгое правило таким образом, чтобы обеспечить типовую безопасность. Параметр ковариантного типа объявляется с помощью ключевого слова `out`, которое предваряет имя этого параметра.

Для того чтобы стали понятнее последствия применения ковариантности, обратимся к конкретному примеру. Ниже приведен очень простой интерфейс `IMyCoVarGenIF`, в котором применяется ковариантность.

```
// В этом обобщенном интерфейсе поддерживается ковариантность.
public interface IMyCoVarGenIF<out T> {
  T GetObject();
}
```

Обратите особое внимание на то, как объявляется параметр обобщенного типа `T`. Его имени предшествует ключевое слово `out`. В данном контексте ключевое слово `out` обозначает, что обобщенный тип `T` является ковариантным. А раз он ковариантный, то метод `GetObject()` может возвращать ссылку на обобщенный тип `T` или же ссылку на любой класс, производный от типа `T`.

Несмотря на свою ковариантность по отношению к обобщенному типу `T`, интерфейс `IMyCoVarGenIF` реализуется аналогично любому другому обобщенному интерфейсу. Ниже приведен пример реализации этого интерфейса в классе `MyClass`.

```
// Реализовать интерфейс IMyCoVarGenIF.
class MyClass<T> : IMyCoVarGenIF<T> {
  T obj;

  public MyClass(T v) { obj = v; }

  public T GetObject() { return obj; }
}
```

Обратите внимание на то, что ключевое слово `out` не указывается еще раз в выражении, объявляющем реализацию данного интерфейса в классе `MyClass`. Это не только не нужно, но и вредно, поскольку всякая попытка еще раз указать ключевое слово `out` будет расцениваться компилятором как ошибка.

А теперь рассмотрим следующую простую реализацию иерархии классов.

```
// Создать простую иерархию классов.
class Alpha {
  string name;

  public Alpha(string n) { name = n; }

  public string GetName() { return name; }
    // ...
  }

class Beta : Alpha {
  public Beta(string n) : base(n) { }
  // ...
}
```

Как видите, класс `Beta` является производным от класса `Alpha`.

С учетом всего изложенного выше, следующая последовательность операций будет считаться вполне допустимой.

```
// Создать ссылку из интерфейса IMyCoVarGenIF на объект типа MyClass<Alpha>.
// Это вполне допустимо как при наличии ковариантности, так и без нее.
IMyCoVarGenIF<Alpha> AlphaRef =
              new MyClass<Alpha>(new Alpha("Alpha #1"));

Console.WriteLine("Имя объекта, на который ссылается переменная AlphaRef: " +
                  AlphaRef.GetObject().GetName());

// А теперь создать объект MyClass<Beta> и присвоить его переменной AlphaRef.
// *** Эта строка кода вполне допустима благодаря ковариантности. ***
AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));

Console.WriteLine("Имя объекта, на который теперь ссылается " +
                  "переменная AlphaRef: " + AlphaRef.GetObject().GetName());
```

Прежде всего, переменной `AlphaRef` типа `IMyCoVarGenIF<Alpha>` в этом фрагменте кода присваивается ссылка на объект типа `MyClass<Alpha>`. Это вполне допустимая операция, поскольку в классе `MyClass` реализуется интерфейс `IMyCoVarGenIF`, причем и в том, и в другом в качестве аргумента типа указывается `Alpha`. Далее имя объекта выводится на экран при вызове метода `GetName()` для объекта, возвращаемого методом `GetObject()`. И эта операция вполне допустима, поскольку `Alpha` — это и тип, возвращаемый методом `GetName()`, и обобщенный тип `T`. После этого переменной `AlphaRef` присваивается ссылка на экземпляр объекта типа `MyClass<Beta>`, что также допустимо, потому что класс `Beta` является производным от класса `Alpha`, а обобщенный тип `T` — ковариантным в интерфейсе `IMyCoVarGenIF`. Если бы любое из этих условий не выполнялось, данная операция оказалась бы недопустимой.

Ради большей наглядности примера вся рассмотренная выше последовательность операций собрана ниже в единую программу.

```
// Продемонстрировать ковариантность в обобщенном интерфейсе.
using System;

// Этот обобщенный интерфейс поддерживает ковариантность.
public interface IMyCoVarGenIF<out T> {
  T GetObject();
}

// Реализовать интерфейс IMyCoVarGenIF.
class MyClass<T> : IMyCoVarGenIF<T> {
  T obj;

  public MyClass(T v) { obj = v; }

  public T GetObject() { return obj; }
}

// Создать простую иерархию классов.
class Alpha {
  string name;

  public Alpha(string n) { name = n; }
```

```
  public string GetName() { return name; }
  // ...
}

class Beta : Alpha {
  public Beta(string n) : base(n) { }
  // ...
}

class VarianceDemo {
  static void Main() {
    // Создать ссылку из интерфейса IMyCoVarGenIF на объект типа
MyClass<Alpha>.
    // Это вполне допустимо как при наличии ковариантности, так и без нее.
    IMyCoVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>(new Alpha("Alpha #1"));

    Console.WriteLine("Имя объекта, на который ссылается переменная " +
                      "AlphaRef: " + AlphaRef.GetObject().GetName());

    // А теперь создать объект MyClass<Beta> и присвоить его
    // переменной AlphaRef.
    // *** Эта строка кода вполне допустима благодаря ковариантности. ***
    AlphaRef = new MyClass<Beta>(new Beta("Beta #1"));

    Console.WriteLine("Имя объекта, на который теперь ссылается переменная " +
                      "AlphaRef: " + AlphaRef.GetObject().GetName());
  }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
Имя объекта, на который ссылается переменная AlphaRef: Alpha #1
Имя объекта, на который теперь ссылается переменная AlphaRef: Beta #1
```

Следует особо подчеркнуть, что переменной `AlphaRef` можно присвоить ссылку на объект типа `MyClass<Beta>` благодаря только тому, что обобщенный тип `T` указан как ковариантный в интерфейсе `IMyCoVarGenIF`. Для того чтобы убедиться в этом, удалите ключевое слово `out` из объявления параметра обобщенного типа `T` в интерфейсе `IMyCoVarGenIF` и попытайтесь скомпилировать данную программу еще раз. Компиляция завершится неудачно, поскольку строгая проверка на соответствие типов не разрешит теперь подобное присваивание.

Один обобщенный интерфейс может вполне наследовать от другого. Иными словами, обобщенный интерфейс с параметром ковариантного типа можно расширить, как показано ниже.

```
public interface IMyCoVarGenIF2<out T> : IMyCoVarGenIF<T> {
  // ...
}
```

Обратите внимание на то, что ключевое слово `out` указано только в объявлении расширенного интерфейса. Указывать его в объявлении базового интерфейса не только не нужно, но и не допустимо. И последнее замечание: обобщенный тип `T` допускается не указывать как ковариантный в объявлении интерфейса `IMyCoVarGenIF2`. Но при этом исключается ковариантность, которую может обеспечить расширенный интерфейс

IMyCoVarGetIF. Разумеется, возможность сделать интерфейс IMyCoVarGenIF2 инвариантным может потребоваться в некоторых случаях его применения.

На применение ковариантности накладываются некоторые ограничения. Ковариантность параметра типа может распространяться только на тип, возвращаемый методом. Следовательно, ключевое слово out нельзя применять в параметре типа, служащем для объявления параметра метода. Ковариантность оказывается пригодной только для ссылочных типов. Ковариантный тип нельзя использовать в качестве ограничения в интерфейсном методе. Так, следующий интерфейс считается недопустимым.

```
public interface IMyCoVarGenIF2<out T> {
   void M<V>() where V:T; // Ошибка, ковариантный тип T нельзя
                          // использовать как ограничение
}
```

Применение контравариантности в обобщенном интерфейсе

Применительно к обобщенному интерфейсу контравариантность служит средством, разрешающим методу использовать аргумент, тип которого относится к базовому классу, указанному в соответствующем параметре типа. В прошлом тип аргумента метода должен был в точности соответствовать параметру типа в силу строгой проверки обобщений на соответствие типов. Контравариантность смягчает это строгое правило таким образом, чтобы обеспечить типовую безопасность. Параметр контравариантного типа объявляется с помощью ключевого слова in, которое предваряет имя этого параметра.

Для того чтобы стали понятнее последствия применения ковариантности, вновь обратимся к конкретному примеру. Ниже приведен обобщенный интерфейс IMyContraVarGenIF контравариантного типа. В нем указывается контравариантный параметр обобщенного типа T, который используется в объявлении метода Show().

```
// Это обобщенный интерфейс, поддерживающий контравариантность.
public interface IMyContraVarGenIF<in T> {
  void Show(T obj);
}
```

Как видите, обобщенный тип T указывается в данном интерфейсе как контравариантный с помощью ключевого слова in, предшествующего имени его параметра. Обратите также внимание на то, что T является параметром типа для аргумента obj в методе Show().

Далее интерфейс IMyContraVarGenIF реализуется в классе MyClass, как показано ниже.

```
// Реализовать интерфейс IMyContraVarGenIF.
class MyClass<T> : IMyContraVarGenIF<T> {
  public void Show(T x) { Console.WriteLine(x); }
}
```

В данном случае метод Show() просто выводит на экран строковое представление переменной x, получаемое в результате неявного обращения к методу ToString() из метода WriteLine().

После этого объявляется иерархия классов, как показано ниже.

```
// Создать простую иерархию классов.
class Alpha {
```

```
  public override string ToString() {
    return "Это объект класса Alpha.";
  }
  // ...
}

class Beta : Alpha {
  public override string ToString() {
    return "Это объект класса Beta.";
  }
  // ...
}
```

Ради большей наглядности классы `Alpha` и `Beta` несколько отличаются от аналогичных классов из предыдущего примера применения ковариантности. Обратите также внимание на то, что метод `ToString()` переопределяется таким образом, чтобы возвращать тип объекта.

С учетом всего изложенного выше, следующая последовательность операций будет считаться вполне допустимой.

```
// Создать ссылку из интерфейса IMyContraVarGenIF<Alpha>
// на объект типа MyClass<Alpha>.
// Это вполне допустимо как при наличии контравариантности, так и без нее.
IMyContraVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>();

// Создать ссылку из интерфейса IMyContraVarGenIF<beta>
// на объект типа MyClass<Beta>.
// И это вполне допустимо как при наличии контравариантности, так и без нее.
IMyContraVarGenIF<Beta> BetaRef = new MyClass<Beta>();

// Создать ссылку из интерфейса IMyContraVarGenIF<beta>
// на объект типа MyClass<Alpha>.
// *** Это вполне допустимо благодаря контравариантности. ***
IMyContraVarGenIF<Beta> BetaRef2 = new MyClass<Alpha>();

// Этот вызов допустим как при наличии контравариантности, так и без нее.
BetaRef.Show(new Beta());

// Присвоить переменную AlphaRef переменной BetaRef.
// *** Это вполне допустимо благодаря контравариантности. ***
BetaRef = AlphaRef;

BetaRef.Show(new Beta());
```

Прежде всего, обратите внимание на создание двух переменных ссылочного типа `IMyContraVarGenIF`, которым присваиваются ссылки на объекты класса `MyClass`, где параметры типа совпадают с аналогичными параметрами в интерфейсных ссылках. В первом случае используется параметр типа `Alpha`, а во втором — параметр типа `Beta`. Эти объявления не требуют контравариантности и допустимы в любом случае.

Далее создается переменная ссылочного типа `IMyContraVarGenIF<Beta>`, но на этот раз ей присваивается ссылка на объект класса `MyClass<Alpha>`. Эта операция вполне допустима, поскольку обобщенный тип `T` объявлен как контравариантный.

Как и следовало ожидать, следующая строка, в которой вызывается метод `BetaRef.Show()` с аргументом `Beta`, является вполне допустимой. Ведь `Beta` — это обобщенный тип `T` в классе `MyClass<Beta>` и в то же время аргумент в методе `Show()`.

В следующей строке переменная `AlphaRef` присваивается переменной `BetaRef`. Эта операция вполне допустима лишь в силу контравариантности. В данном случае переменная относится к типу `MyClass<Beta>`, а переменная `AlphaRef` — к типу `MyClass<Alpha>`. Но поскольку `Alpha` является базовым классом для класса `Beta`, то такое преобразование типов оказывается допустимым благодаря контравариантности. Для того чтобы убедиться в необходимости контравариантности в рассматриваемом здесь примере, попробуйте удалить ключевое слово `in` из объявления обобщенного типа `T` в интерфейсе `IMyContraVarGenIF`, а затем попытайтесь скомпилировать приведенный выше код еще раз. В результате появятся ошибки компиляции.

Ради большей наглядности примера вся рассмотренная выше последовательность операций собрана ниже в единую программу.

```
// Продемонстрировать контравариантность в обобщенном интерфейсе.
using System;

// Это обобщенный интерфейс, поддерживающий контравариантность.
public interface IMyContraVarGenIF<in T> {
  void Show(T obj);
}

// Реализовать интерфейс IMyContraVarGenIF.
class MyClass<T> : IMyContraVarGenIF<T> {
  public void Show(T x) { Console.WriteLine(x); }
}

// Создать простую иерархию классов.
class Alpha {
  public override string ToString() {
    return "Это объект класса Alpha.";
  }
  // ...
}

class Beta : Alpha {
  public override string ToString() {
    return "Это объект класса Beta.";
  }
  // ...
}

class VarianceDemo {
  static void Main() {
    // Создать ссылку из интерфейса IMyContraVarGenIF<Alpha>
    // на объект типа MyClass<Alpha>.
    // Это вполне допустимо как при наличии контравариантности, так и без нее.
    IMyContraVarGenIF<Alpha> AlphaRef = new MyClass<Alpha>();

    // Создать ссылку из интерфейса IMyContraVarGenIF<beta>
    // на объект типа MyClass<Beta>.
    // И это вполне допустимо как при наличии контравариантности,
```

```
    // так и без нее.
    IMyContraVarGenIF<Beta> BetaRef = new MyClass<Beta>();

    // Создать ссылку из интерфейса IMyContraVarGenIF<beta>
    // на объект типа MyClass<Alpha>.
    // *** Это вполне допустимо благодаря контравариантности. ***
    IMyContraVarGenIF<Beta> BetaRef2 = new MyClass<Alpha>();

    // Этот вызов допустим как при наличии контравариантности, так и без нее.
    BetaRef.Show(new Beta());

    // Присвоить переменную AlphaRef переменной BetaRef.
    // *** Это вполне допустимо благодаря контравариантности. ***
    BetaRef = AlphaRef;

    BetaRef.Show(new Beta());
  }
}
```

Выполнение этой программы дает следующий результат.

```
Это объект класса Beta.
Это объект класса Beta.
```

Контравариантный интерфейс может быть расширен аналогично описанному выше расширению ковариантного интерфейса. Для достижения контравариантного характера расширенного интерфейса в его объявлении должен быть указан такой же параметр обобщенного типа, как и у базового интерфейса, но с ключевым словом `in`, как показано ниже.

```
public interface IMyContraVarGenIF2<in T> : IMyContraVarGenIF<T> {
// ...
}
```

Следует иметь в виду, что указывать ключевое слово `in` в объявлении базового интерфейса не только не нужно, но и не допустимо. Более того, сам расширенный интерфейс `IMyContraVarGenIF2` не обязательно должен быть контравариантным. Иными словами, обобщенный тип `T` в интерфейсе `IMyContraVarGenIF2` не требуется модифицировать ключевым словом `in`. Разумеется, все преимущества, которые сулит контравариантность в интерфейсе `IMyContraVarGen`, при этом будут утрачены в интерфейсе `IMyContraVarGenIF2`.

Контравариантность оказывается пригодной только для ссылочных типов, а параметр контравариантного типа можно применять только к аргументам методов. Следовательно, ключевое слово `in` нельзя указывать в параметре типа, используемом в качестве возвращаемого типа.

Вариантные делегаты

Как пояснялось в главе 15, ковариантность и контравариантность поддерживается в необобщенных делегатах в отношении типов, возвращаемых методами, и типов, указываемых при объявлении параметров. Начиная с версии C# 4.0, возможности ковариантности и контравариантности были распространены и на обобщенные делегаты. Подобные возможности действуют таким же образом, как было описано выше в отношении обобщенных интерфейсов.

Ниже приведен пример контравариантного делегата.

```
// Объявить делегат, контравариантный по отношению к обобщенному типу T.
delegate bool SomeOp<in T>(T obj);
```

Этому делегату можно присвоить метод с параметром обобщенного типа T или же класс, производный от типа T.

А вот пример ковариантного делегата.

```
// Объявить делегат, ковариантный по отношению к обобщенному типу T.
delegate T AnotherOp<out T, V>(V obj);
```

Этому делегату можно присвоить метод, возвращающий обобщенный тип T, или же класс, производный от типа T. В данном случае V оказывается просто параметром инвариантного типа.

В следующем примере программы демонстрируется применение обоих разновидностей вариантных делегатов на практике.

```
// Продемонстрировать конвариантность и контравариантность
// в обобщенных делегатах.

using System;

// Объявить делегат, контравариантный по отношению к обобщенному типу T.
delegate bool SomeOp<in T>(T obj);

// Объявить делегат, ковариантный по отношению к обобщенному типу T.
delegate T AnotherOp<out T, V>(V obj);

class Alpha {
  public int Val { get; set; }

  public Alpha(int v) { Val = v; }
}

class Beta : Alpha {
  public Beta(int v) : base(v) { }
}

class GenDelegateVarianceDemo {
  // Возвратить логическое значение true, если значение
  // переменной obj.Val окажется четным.
  static bool IsEven(Alpha obj) {
    if((obj.Val % 2) == 0) return true;
    return false;
  }

  static Beta ChangeIt(Alpha obj) {
    return new Beta(obj.Val +2);
  }

  static void Main() {
    Alpha objA = new Alpha(4);
    Beta objB = new Beta(9);

    // Продемонстрировать сначала контравариантность.
     // Объявить делегат SomeOp<Alpha> и задать для него метод IsEven.
     SomeOp<Alpha> checkIt = IsEven;
```

```
    // Объявить делегат SomeOp<Beta>.
    SomeOp<Beta> checkIt2;

    // А теперь присвоить делегат SomeOp<Alpha> делегату SomeOp<Beta>.
    // *** Это допустимо только благодаря контравариантности. ***
    checkIt2 = checkIt;

    // Вызвать метод через делегат.
    Console.WriteLine(checkIt2(objB));

    // Далее, продемонстрировать контравариантность.

    // Объявить сначала два делегата типа AnotherOp.
    // Здесь возвращаемым типом является класс Beta,
    // а параметром типа — класс Alpha.
    // Обратите внимание на то, что для делегата modifyIt
    // задается метод ChangeIt.
    AnotherOp<Beta, Alpha> modifyIt = ChangeIt;

    // Здесь возвращаемым типом является класс Alpha,
    // а параметром типа — тот же класс Alpha.
    AnotherOp<Alpha, Alpha> modifyIt2;

    // А теперь присвоить делегат modifyIt делегату modifyIt2.
    // *** Это допустимо только благодаря ковариантности. ***
    modifyIt2 = modifyIt;

    // Вызвать метод и вывести результаты на экран.
    objA = modifyIt2(objA);
    Console.WriteLine(objA.Val);
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
False
6
```

Каждая операция достаточно подробно поясняется в комментариях к данной программе. Следует особо подчеркнуть, для успешной компиляции программы в объявлении обоих типов делегатов `SomeOp` and `AnotherOp` должны быть непременно указаны ключевые слова `in` и `out` соответственно. Без этих модификаторов компиляция программы будет выполнена с ошибками из-за отсутствия неявных преобразований типов в означенных строках кода.

Создание экземпляров объектов обобщенных типов

Когда приходится иметь дело с обобщениями, то нередко возникает вопрос: не приведет ли применение обобщенного класса к неоправданному раздуванию кода? Ответ на этот вопрос прост: не приведет. Дело в том, что в C# обобщения реализованы весьма эффективным образом: новые объекты конструируемого типа создаются лишь по мере надобности. Этот процесс описывается ниже.

Когда обобщенный класс компилируется в псевдокод MSIL, он сохраняет все свои параметры типа в их обобщенной форме. А когда конкретный экземпляр класса потребуется во время выполнения программы, то JIT-компилятор сконструирует конкретный вариант этого класса в исполняемом коде, в котором параметры типа заменяются аргументами типа. В каждом экземпляре с теми же самыми аргументами типа будет использоваться один и тот же вариант данного класса в исполняемом коде.

Так, если имеется некоторый обобщенный класс `Gen<T>`, то во всех объектах типа `Gen<T>` будет использоваться один и тот же исполняемый код данного класса. Следовательно, раздувание кода исключается благодаря тому, что в программе создаются только те варианты класса, которые действительно требуются. Когда же возникает потребность сконструировать объект другого типа, то компилируется новый вариант класса в исполняемом коде.

Как правило, новый исполняемый вариант обобщенного класса создается для каждого объекта конструируемого типа, в котором аргумент имеет тип значения, например `int` или `double`. Следовательно, в каждом объекте типа `Gen<int>` будет использоваться один исполняемый вариант класса `Gen`, а в каждом объекте типа `Gen<double>` — другой вариант класса `Gen`, причем каждый вариант приспосабливается к конкретному типу значения. Но во всех случаях, когда аргумент оказывается ссылочного типа, используется *только один вариант* обобщенного класса, поскольку все ссылки имеют одинаковую длину (в байтах). Такая оптимизация также исключает раздувание кода.

Некоторые ограничения, присущие обобщениям

Ниже перечислен ряд ограничений, которые следует иметь в виду при использовании обобщений.

- Свойства, операторы, индексаторы и события не могут быть обобщенными. Но эти элементы могут использоваться в обобщенном классе, причем с параметрами обобщенного типа этого класса.
- К обобщенному методу нельзя применять модификатор `extern`.
- Типы указателей нельзя использовать в аргументах типа.
- Если обобщенный класс содержит поле типа `static`, то в объекте *каждого* конструируемого типа должна быть *своя* копия этого поля. Это означает, что во всех экземплярах объектов *одного* конструируемого типа совместно используется одно и то же поле типа `static`. Но в экземплярах объектов *другого* конструируемого типа совместно используется другая копия этого поля. Следовательно, поле типа `static` не может совместно использоваться объектами *всех* конструируемых типов.

Заключительные соображения относительно обобщений

Обобщения являются весьма эффективным дополнением C#, поскольку они упрощают создание типизированного, повторно используемого кода. Несмотря на несколько усложненный, на первый взгляд, синтаксис обобщений, их применение быстро входит в привычку. Аналогично, умение применять ограничения к месту требует некоторой практики и со временем не вызывает особых затруднений. Обобщения теперь стали неотъемлемой частью программирования на C#. Поэтому освоение этого важного языкового средства стоит затраченных усилий.

19 ГЛАВА
LINQ

Без сомнения, LINQ относится к одним из самых интересных средств языка C#. Эти средства были внедрены в версии C# 3.0 и явились едва ли не самым главным его дополнением, которое состояло не только во внесении совершенно нового элемента в синтаксис C#, добавлении нескольких ключевых слов и предоставлении больших возможностей, но и в значительном расширении рамок данного языка программирования и круга задач, которые он позволяет решать. Проще говоря, внедрение LINQ стало поворотным моментом в истории развития C#.

Аббревиатура LINQ означает Language-Integrated Query, т.е. *язык интегрированных запросов*. Это понятие охватывает ряд средств, позволяющих извлекать информацию из источника данных. Как вам должно быть известно, извлечение данных составляет важную часть многих программ. Например, программа может получать информацию из списка заказчиков, искать информацию в каталоге продукции или получать доступ к учетному документу, заведенному на работника. Как правило, такая информация хранится в базе данных, существующей отдельно от приложения. Так, каталог продукции может храниться в реляционной базе данных. В прошлом для взаимодействия с такой базой данных приходилось формировать запросы на языке структурированных запросов (SQL). А для доступа к другим источникам данных, например в формате XML, требовался отдельный подход. Следовательно, до версии 3.0 поддержка подобных запросов в C# отсутствовала. Но это положение изменилось после внедрения LINQ.

LINQ дополняет C# средствами, позволяющими формировать запросы для любого LINQ-совместимого источника

данных. При этом синтаксис, используемый для формирования запросов, остается неизменным, независимо от типа источника данных. Это, в частности, означает, что синтаксис, требующийся для формирования запроса к реляционной базе данных, практически ничем не отличается от синтаксиса запроса данных, хранящихся в массиве. Для этой цели теперь не нужно прибегать к средствам SQL или другого внешнего по отношению к C# механизма извлечения данных из источника. Возможности формировать запросы отныне полностью интегрированы в язык C#.

Помимо SQL, LINQ можно использовать вместе с XML-файлами и наборами данных ADO.NET Dataset. Не менее важным является применение LINQ вместе с массивами и коллекциями в C# (подробнее рассматриваемыми в главе 25). Таким образом, средства LINQ предоставляют, в целом, единообразный доступ к данным. И хотя такой принцип уже сам по себе является весьма эффективным и новаторским, преимущества LINQ этим не ограничиваются. LINQ предлагает осмыслить иначе и подойти по-другому к решению многих видов задач программирования, помимо традиционной организации доступа к базам данных. И в конечном итоге многие решения могут быть выработаны на основе LINQ.

LINQ поддерживается целым рядом взаимосвязанных средств, включая внедренный в C# синтаксис запросов, лямбда-выражения, анонимные типы и методы расширения. О лямбда-выражениях речь уже шла в главе 15, а остальные средства рассматриваются в этой главе.

ПРИМЕЧАНИЕ

LINQ в C# — это, по сути, язык в языке. Поэтому предмет рассмотрения LINQ довольно обширен и включает в себя многие средства, возможности и альтернативы. Несмотря на то что в этой главе дается подробное описание средств LINQ, рассмотреть здесь все их возможности, особенности и области применения просто невозможно. Для этого потребовалась бы отдельная книга. В связи с этим в настоящей главе основное внимание уделяется главным элементам LINQ, применение которых демонстрируется на многочисленных примерах. А в долгосрочной перспективе LINQ представляет собой подсистему, которую придется изучать самостоятельно и достаточно подробно.

Основы LINQ

В основу LINQ положено понятие *запроса*, в котором определяется информация, получаемая из источника данных. Например, запрос списка рассылки почтовых сообщений заказчикам может потребовать предоставления адресов всех заказчиков, проживающих в конкретном городе; запрос базы данных товарных запасов — список товаров, запасы которых исчерпались на складе; а запрос журнала, регистрирующего интенсивность использования Интерента, — список наиболее часто посещаемых вебсайтов. И хотя все эти запросы отличаются в деталях, их можно выразить, используя одни и те же синтаксические элементы LINQ.

Как только запрос будет сформирован, его можно выполнить. Это делается, в частности, в цикле `foreach`. В результате выполнения запроса выводятся его результаты. Поэтому использование запроса может быть разделено на две главные стадии. На первой стадии запрос формируется, а на второй — выполняется. Таким образом, при формировании запроса определяется, что *именно* следует извлечь из источника данных. А при выполнении запроса выводятся конкретные *результаты*.

Для обращения к источнику данных по запросу, сформированному средствами LINQ, в этом источнике должен быть реализован интерфейс `IEnumerable`. Он имеет две формы: обобщенную и необобщенную. Как правило, работать с источником данных легче, если в нем реализуется обобщенная форма `IEnumerable<T>`, где `T` обозначает обобщенный тип перечисляемых данных. Здесь и далее предполагается, что в источнике данных реализуется форма интерфейса `IEnumerable<T>`. Этот интерфейс объявляется в пространстве имен `System.Collections.Generic`. Класс, в котором реализуется форма интерфейса `IEnumerable<T>`, поддерживает перечисление, а это означает, что его содержимое может быть получено по очереди или в определенном порядке. Форма интерфейса `IEnumerable<T>` поддерживается всеми массивами в C#. Поэтому на примере массивов можно наглядно продемонстрировать основные принципы работы LINQ. Следует, однако, иметь в виду, что применение LINQ не ограничивается одними массивами.

Простой запрос

А теперь самое время обратиться к простому примеру использования LINQ. В приведенной ниже программе используется запрос для получения положительных значений, содержащихся в массиве целых значений.

```
// Сформировать простой запрос LINQ.

using System;
using System.Linq;

class SimpQuery {
  static void Main() {

    int[] nums = { 1, -2, 3, 0, -4, 5 };

    // Сформировать простой запрос на получение только положительных значений.
    var posNums = from n in nums
                  where n > 0
                  select n;

    Console.Write("Положительные значения из массива nums: ");

    // Выполнить запрос и отобразить его результаты.
    foreach(int i in posNums) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

Эта программа дает следующий результат.

```
Положительные значения из массива nums: 1 3 5
```

Как видите, в конечном итоге отображаются только положительные значения, хранящиеся в массиве `nums`. Несмотря на всю свою простоту, этот пример наглядно демонстрирует основные возможности LINQ. Поэтому рассмотрим его более подробно.

Прежде всего обратите внимание на применение в данном примере программы следующего оператора.

```
using System.Linq;
```

Для применения средств LINQ в исходный текст программы следует включить пространство имен `System.Linq`.

Затем в программе объявляется массив `nums` типа `int`. Все массивы в C# неявным образом преобразуются в форму интерфейса `IEnumerable<T>`. Благодаря этому любой массив в C# может служить в качестве источника данных, извлекаемых по запросу LINQ.

Далее объявляется запрос, по которому из массива `nums` извлекаются элементы только с положительными значениями.

```
var posNums = from n in nums
              where n > 0
              select n;
```

Переменная `posNums` называется *переменной запроса*. В ней хранится ссылка на ряд правил, определяемых в запросе. Обратите внимание на применение ключевого слова `var` для объявления переменной `posNums` неявным образом. Как вам должно быть уже известно, благодаря этому переменная `posNums` становится неявно типизированной. Такими переменными удобно пользоваться в запросах, хотя их тип можно объявить и явным образом (это должна быть одна из форм интерфейса `IEnumerable<T>`). Объявляемой переменной `posNums` в итоге присваивается выражение запроса.

Все запросы начинаются с оператора `from`, определяющего два элемента. Первым из них является *переменная диапазона*, принимающая элементы из источника данных. В рассматриваемом здесь примере эту роль выполняет переменная `n`. Вторым элементом является источник данных (в данном случае — массив `nums`). Тип переменной диапазона выводится из источника данных. Поэтому переменная `n` относится к типу `int`. Ниже приведена общая форма оператора `from`.

```
from переменная_диапазона in источник_данных
```

Далее следует оператор `where`, обозначающий условие, которому должен удовлетворять элемент в источнике данных, чтобы его можно было получить по запросу. Ниже приведена общая форма синтаксиса оператора `where`.

```
where булево_выражение
```

В этой форме *булево_выражение* должно давать результат типа `bool`. Такое выражение иначе называется *предикатом*. В запросе можно указывать несколько операторов `where`. В данном примере программы используется следующий оператор `where`.

```
where n > 0
```

Этот оператор будет давать истинный результат только для тех элементов массива, значения которых оказываются больше нуля. Выражение `n > 0` будет вычисляться для каждого из `n` элементов массива `n` при выполнении запроса. В итоге будут получены только те значения, которые удовлетворяют этому условию. Иными словами, оператор `where` выполняет роль своеобразного фильтра, отбирая лишь определенные элементы.

Все запросы оканчиваются оператором `select` или `group`. В данном примере используется оператор `select`, точно определяющий, что именно должно быть получено по запросу. В таких простых примерах запросов, как рассматриваемый здесь, выбирается конкретное значение диапазона. Поэтому по данному запросу возвращаются только те целые значения, которые удовлетворяют условию, указанному в операторе `where`. В более сложных запросах можно дополнительно уточнять, что именно

следует выбирать. Например, по запросу списка рассылки может быть получена лишь фамилия адресата вместо его полного адреса. Обратите внимание на то, что оператор `select` завершается точкой с запятой, поскольку это последний оператор в запросе. А другие его операторы не оканчиваются точкой с запятой.

Итак, переменная запроса `posNums` создана, но результаты запроса пока еще не получены. Дело в том, что сам запрос определяет лишь ряд конкретных правил, а результаты будут только после выполнения запроса. Кроме того, один и тот же запрос может быть выполнен два раза или больше, причем с разными результатами, если в промежутке между последовательно производимыми попытками выполнить один и тот же запрос изменяется базовый источник данных. Поэтому одного лишь объявления переменной запроса `posNums` совершенно недостаточно для того, чтобы она содержала результаты запроса.

Для выполнения запроса в данном примере программы организуется следующий цикл.

```
foreach(int i in posNums) Console.WriteLine(i + " ");
```

В этом цикле переменная `posNums` указывается в качестве коллекции, к которой происходит обращение на каждом шаге цикла. В цикле `foreach` соблюдаются правила, определенные в запросе и доступные по ссылке из переменной `posNums`. На каждом шаге цикла возвращается очередной элемент, полученный из массива. Этот процесс завершается, когда запрашиваемых элементов в массиве больше не обнаружено. В данном примере тип `int` переменной шага цикла `i` указывается явно, поскольку по запросу извлекаются элементы именно этого типа. Явное указание типа переменной шага цикла вполне допустимо в тех случаях, когда заранее известен тип значения, выбираемого по запросу. Но в более сложных случаях оказывается проще, а иногда даже нужно, указывать тип переменной шага цикла неявным образом с помощью ключевого слова `var`.

Неоднократное выполнение запросов

Итак, в запросе определяются правила, по которым извлекаются данные, но этого явно недостаточно для получения результатов, поскольку запрос должен быть выполнен, причем это может быть сделано несколько раз. Если же в промежутке между последовательно производимыми попытками выполнить один и тот же запрос источник данных изменяется, то получаемые результаты могут отличаться. Поэтому как только запрос определен, его выполнение будет всегда давать только самые последние результаты. Обратимся к конкретному примеру. Ниже приведен другой вариант рассматриваемой здесь программы, где содержимое массива `nums` изменяется в промежутке между двумя последовательно производимыми попытками выполнить один и тот же запрос, хранящийся в переменной `posNums`.

```
// Сформировать простой запрос.

using System;
using System.Linq;
using System.Collections.Generic;

class SimpQuery {
  static void Main() {
    int[] nums = { 1, -2, 3, 0, -4, 5 };
```

```
    // Сформировать простой запрос на получение только положительных значений.
    var posNums = from n in nums
                  where n > 0
                  select n;

    Console.Write("Положительные значения из массива nums: ");

    // Выполнить запрос и отобразить его результаты.
    foreach(int i in posNums) Console.Write(i + " ");

    Console.WriteLine();

    // Внести изменения в массив nums.
    Console.WriteLine("\nЗадать значение 99 для элемента массива nums[1].");
    nums[1] = 99;

    Console.Write("Положительные значения из массива nums\n" +
                  "после изменений в нем: ");

    // Выполнить запрос второй раз.
    foreach(int i in posNums) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Положительные значения из массива nums: 1 3 5

Задать значение 99 для элемента массива nums[1].
Положительные значения из массива nums
после изменений в нем: 1 99 3 5
```

Как следует из результата выполнения приведенной выше программы, значение элемента массива `nums[1]` изменилось с –2 на 99, что и отражают результаты повторного выполнения запроса. Этот важный момент следует подчеркнуть особо. Каждая попытка выполнить запрос приносит свои результаты, получаемые при перечислении текущего содержимого источника данных. Поэтому если источник данных претерпевает изменения, то могут измениться и результаты выполнения запроса. Преимущества такого подхода к обработке запросов весьма значительны. Так, если по запросу получается список необработанных заказов в Интернет-магазине, то при каждой попытке выполнить запрос желательно получить сведения обо всех заказах, включая и только что введенные.

Связь между типами данных в запросе

Как показывает предыдущий пример, запрос включает в себя переменные, типы которых связаны друг с другом. К их числу относятся переменная запроса, переменная диапазона и источник данных. Соблюсти соответствие этих типов данных очень важно, но в то же время нелегко — по крайней мере, так кажется на первый взгляд, поэтому данный вопрос заслуживает более пристального внимания.

Тип переменной диапазона должен соответствовать типу элементов, хранящихся в источнике данных. Следовательно, тип переменной диапазона зависит от типа источника данных. Как правило, тип переменной диапазона может быть выведен средствами C#. Но выводимость типов может быть осуществлена при условии, что в источнике данных реализована форма интерфейса `IEnumerable<T>`, где `T` обозначает тип элементов в источнике данных. (Как упоминалось выше, форма интерфейса `IEnumerable<T>` реализуется во всех массивах, как, впрочем, и во многих других источниках данных.) Но если в источнике данных реализован необобщенный вариант интерфейса `IEnumerable`, то тип переменной диапазона придется указывать явно. И это делается в операторе `from`. Ниже приведен пример явного объявления типа `int` переменной диапазона `n`.

```
var posNums = from int n in nums
  // ...
```

Очевидно, что явное указание типа здесь не требуется, поскольку все массивы неявно преобразуются в форму интерфейса `IEnumerable<T>`, которая позволяет вывести тип переменной диапазона.

Тип объекта, возвращаемого по запросу, представляет собой экземпляр интерфейса `IEnumerable<T>`, где `T` — тип получаемых элементов. Следовательно, тип переменной запроса должен быть экземпляром интерфейса `IEnumerable<T>`, а значение `T` должно определяться типом значения, указываемым в операторе `select`. В предыдущих примерах значению `T` соответствовал тип `int`, поскольку переменная `n` имела тип `int`. (Как пояснялось выше, переменная `n` относится к типу `int`, потому что элементы именно этого типа хранятся в массиве `nums`.) С учетом явного указания типа `IEnumerable<int>` упомянутый выше запрос можно было бы составить следующим образом.

```
IEnumerable<int> posNums = from n in nums
                           where n > 0
                           select n;
```

Следует иметь в виду, что тип элемента, выбираемого оператором `select`, должен соответствовать типу аргумента, передаваемого форме интерфейса `IEnumerable<T>`, используемой для объявления переменной запроса. Зачастую при объявлении переменных запроса используется ключевое слово `var` вместо явного указания ее типа, поскольку это дает компилятору возможность самому вывести соответствующий тип данной переменной из оператора `select`. Как будет показано далее в этой главе, такой подход оказывается особенно удобным в тех случаях, когда оператор `select` возвращает из источника данных нечто более существенное, чем отдельный элемент.

Когда запрос выполняется в цикле `foreach`, тип переменной шага цикла должен быть таким же, как и тип переменной диапазона. В предыдущих примерах тип этой переменной указывался явно как `int`. Но имеется и другая возможность: предоставить компилятору самому вывести тип данной переменной, и для этого достаточно указать ее тип как `var`. Как будет показано далее в этой главе, ключевое слово `var` приходится использовать и в тех случаях, когда тип данных просто неизвестен.

Общая форма запроса

У всех запросов имеется общая форма, основывающаяся на ряде приведенных ниже контекстно-зависимых ключевых слов.

Ascending	by	descending	equals
from	group	in	into
join	let	on	orderby
select	where		

Среди них лишь приведенные ниже ключевые слова используются в начале операторов запроса.

from	group	join	let
orderby	select	where	

Запрос должен начинаться с ключевого слова `from` и оканчиваться ключевым словом `select` или `group`. Оператор `select` определяет тип значения, перечисляемого по запросу, а оператор `group` возвращает данные группами, причем каждая группа может перечисляться по отдельности. Как следует из приведенных выше примеров, в операторе `where` указываются критерии, которым должен удовлетворять искомый элемент, чтобы быть полученным по запросу. А остальные операторы позволяют уточнить запрос. Все они рассматриваются далее по порядку.

Отбор запрашиваемых значений с помощью оператора `where`

Как пояснялось выше, оператор `where` служит для отбора данных, возвращаемых по запросу. В предыдущих примерах этот оператор был продемонстрирован в своей простейшей форме, в которой для отбора данных используется единственное условие. Однако для более тщательного отбора данных можно задать несколько условий и, в частности, в нескольких операторах `where`. В качестве примера рассмотрим следующую программу, в которой из массива выводятся только те значения, которые положительны и меньше 10.

```
// Использовать несколько операторов where.

using System;
using System.Linq;

class TwoWheres {
  static void Main() {

    int[] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };

    // Сформировать запрос на получение положительных значений меньше 10.
    var posNums = from n in nums
                  where n > 0
                  where n < 10
                  select n;

    Console.Write("Положительные значения меньше 10: ");

    // Выполнить запрос и вывести его результаты.
    foreach(int i in posNums) Console.Write (i + " ");
    Console.WriteLine();
  }
}
```

Эта программа дает следующий результат.

```
Положительные значения меньше 10: 1 3 6 9
```

Как видите, по данному запросу извлекаются только положительные значения меньше 10. Этот результат достигается благодаря двум следующим операторам `where`.

```
where n > 0
where n < 10
```

Условие в первом операторе `where` требует, чтобы элемент массива был больше нуля. А условие во втором операторе `where` требует, чтобы элемент массива был меньше 10. Следовательно, запрашиваемый элемент массива должен находиться в пределах от 1 до 9 (включительно), чтобы удовлетворять обоим условиям.

В таком применении двух операторов `where` для отбора данных нет ничего дурного, но аналогичного эффекта можно добиться с помощью более компактно составленного условия в единственном операторе `where`. Ниже приведен тот же самый запрос, переформированный по этому принципу.

```
var posNums = from n in nums
              where n > 0 && n < 10
              select n;
```

Как правило, в условии оператора `where` разрешается использовать любое допустимое в C# выражение, дающее булев результат. Например, в приведенной ниже программе определяется массив символьных строк. В ряде этих строк содержатся адреса Интернета. По запросу в переменой `netAddrs` извлекаются только те строки, которые содержат более четырех символов и оканчиваются на ".net". Следовательно, по данному запросу обнаруживаются строки, содержащие адреса Интернета с именем `.net` домена самого верхнего уровня.

```
// Продемонстрировать применение еще одного оператора where.

using System;
using System.Linq;

class WhereDemo2 {
  static void Main() {

    string[] strs = { ".com", ".net", "hsNameA.com",
                      "hsNameB.net", "test", ".network",
                      "hsNameC.net", "hsNameD.com" };

    // Сформировать запрос на получение адресов
    // Интернета, оканчивающихся на .net.
    var netAddrs = from addr in strs
                   where addr.Length > 4 && addr.EndsWith(".net",
                         StringComparison.Ordinal)
                   select addr;

    // Выполнить запрос и вывести его результаты.
    foreach(var str in netAddrs) Console.WriteLine(str);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
hsNameB.net
hsNameC.net
```

Обратите внимание на то, что в операторе `where` данной программы используется один из методов обработки символьных строк под названием `EndsWith()`. Он возвращает логическое значение `true`, если вызывающая его строка оканчивается последовательностью символов, указываемой в качестве аргумента этого метода.

Сортировка результатов запроса с помощью оператора `orderby`

Зачастую результаты запроса требуют сортировки. Допустим, что требуется получить список просроченных счетов по порядку остатка на счету: от самого большого до самого малого или же список имен заказчиков в алфавитном порядке. Независимо от преследуемой цели, результаты запроса можно очень легко отсортировать, используя такое средство LINQ, как оператор `orderby`.

Оператор `orderby` можно использовать для сортировки результатов запроса по одному или нескольким критериям. Рассмотрим для начала самый простой случай сортировки по одному элементу. Ниже приведена общая форма оператора `orderby` для сортировки результатов запроса по одному критерию:

```
orderby элемент порядок
```

где *элемент* обозначает конкретный элемент, по которому проводится сортировка. Это может быть весь элемент, хранящийся в источнике данных, или только часть одного поля в данном элементе. А *порядок* обозначает порядок сортировки по нарастающей или убывающей с обязательным добавлением ключевого слова `ascending` или `descending` соответственно. По умолчанию сортировка проводится по нарастающей, и поэтому ключевое слово `ascending`, как правило, не указывается.

Ниже приведен пример программы, в которой оператор `orderby` используется для извлечения значений из массива типа `int` по нарастающей.

```
// Продемонстрировать применение оператора orderby.

using System;
using System.Linq;

class OrderbyDemo {
  static void Main() {

    int[] nums = { 10, -19, 4, 7, 2, -5, 0 };

    // Сформировать запрос на получение значений в отсортированном порядке.
    var posNums = from n in nums
                  orderby n
                  select n;

    Console.Write("Значения по нарастающей: ");

    // Выполнить запрос и вывести его результаты.
    foreach(int i in posNums) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Значения по нарастающей: -19 -5 0 2 4 7 10
```

Для того чтобы изменить порядок сортировки по нарастающей на сортировку по убывающей, достаточно указать ключевое слово `descending`, как показано ниже.

```
var posNums = from n in nums
              orderby n descending
              select n;
```

Попробовав выполнить этот запрос, вы получите значения в обратном порядке.

Зачастую сортировка результатов запроса проводится по единственному критерию. Тем не менее для сортировки по нескольким критериям служит приведенная ниже форма оператора `orderby`.

```
orderby элемент_A направление, элемент_B направление, элемент_C направление, ...
```

В данной форме *элемент_A* обозначает конкретный элемент, по которому проводится основная сортировка; *элемент_B* — элемент, по которому производится сортировка каждой группы эквивалентных элементов; *элемент_C* — элемент, по которому производится сортировка всех этих групп, и т.д. Таким образом, каждый последующий *элемент* обозначает дополнительный критерий сортировки. Во всех этих критериях указывать *направление* сортировки необязательно, но по умолчанию сортировка проводится по нарастающей. Ниже приведен пример программы, в которой сортировка информации о банковских счетах осуществляется по трем критериям: фамилии, имени и остатку на счете.

```
// Сортировать результаты запроса по нескольким
// критериям, используя оператор orderby.

using System;
using System.Linq;

class Account {
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public double Balance { get; private set; }
  public string AccountNumber { get; private set; }

  public Account(string fn, string ln, string accnum, double b) {
    FirstName = fn;
    LastName = ln;
    AccountNumber = accnum;
    Balance = b;
  }
}

class OrderbyDemo {
  static void Main() {

   // Сформировать исходные данные.
   Account[] accounts =
     { new Account("Том", "Смит", "132CK", 100.23),
       new Account("Том", "Смит", "132CD", 10000.00),
```

```
        new Account("Ральф", "Джонс", "436CD", 1923.85),
        new Account("Ральф", "Джонс", "454MM", 987.132),
        new Account("Тед", "Краммер", "897CD", 3223.19),
        new Account("Ральф", "Джонс", "434CK", -123.32),
        new Account("Сара", "Смит", "543MM", 5017.40),
        new Account("Сара", "Смит", "547CD", 34955.79),
        new Account("Сара", "Смит", "843CK", 345.00),
        new Account("Альберт", "Смит", "445CK", -213.67),
        new Account("Бетти", "Краммер","968MM",5146.67),
        new Account("Карл", "Смит", "078CD", 15345.99),
        new Account("Дженни", "Джонс", "108CK", 10.98)
      };

    // Сформировать запрос на получение сведений о
    // банковских счетах в отсортированном порядке.
    // Отсортировать эти сведения сначала по имени, затем
    // по фамилии и, наконец, по остатку на счете.
    var accInfo = from acc in accounts
                  orderby acc.LastName, acc.FirstName, acc.Balance
                  select acc;
    Console.WriteLine("Счета в отсортированном порядке: ");

    string str = "";

    // Выполнить запрос и вывести его результаты.
    foreach(Account acc in accInfo) {
      if(str != acc.FirstName) {
        Console.WriteLine();
        str = acc.FirstName;
      }

    Console.WriteLine("{0}, {1}\tНомер счета: {2}, {3,10:C}",
                      acc.LastName, acc.FirstName,
                      acc. AccountNumber, acc.Balance);
    }
    Console.WriteLine();
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Счета в отсортированном порядке:

Джонс, Дженни     Номер счета: 108CK,    $10.98

Джонс, Ральф      Номер счета: 434CK,  ($123.32)
Джонс, Ральф      Номер счета: 454MM,    $987.13
Джонс, Ральф      Номер счета: 436CD,  $1,923.85

Краммер, Бетти    Номер счета: 968MM,  $5,146.67

Краммер, Тед      Номер счета: 897CD,  $3,223.19

Смит, Альберт     Номер счета: 445CK,  ($213.67)
```

```
Смит, Карл        Номер счета: 078CD, $15,345.99

Смит, Сара        Номер счета: 843CK,    $345.00
Смит, Сара        Номер счета: 543MM,  $5,017.40
Смит, Сара        Номер счета: 547CD, $34,955.79

Смит, Том         Номер счета: 132CK,    $100.23
Смит, Том         Номер счета: 132CD, $10,000.00
```

Внимательно проанализируем оператор `orderby` в следующем запросе из приведенной выше программы.

```
var accInfo = from acc in accounts
              orderby acc.LastName, acc.FirstName, acc.Balance
              select acc;
```

Сортировка результатов этого запроса осуществляется следующим образом. Сначала результаты сортируются по фамилии, затем элементы с одинаковыми фамилиями сортируются по имени. И наконец, группы элементов с одинаковыми фамилиями и именами сортируются по остатку на счете. Именно поэтому список счетов вкладчиков по фамилии Джонс выглядит так.

```
Джонс, Дженни     Номер счета: 108CK,    $10.98

Джонс, Ральф      Номер счета: 434CK, ($123.32)
Джонс, Ральф      Номер счета: 454MM,   $987.13
Джонс, Ральф      Номер счета: 436CD, $1,923.85
```

Как показывает результат выполнения данного запроса, список счетов отсортирован сначала по фамилии, затем по имени и, наконец, по остатку на счете.

Используя несколько критериев, можно изменить на обратный порядок любой сортировки с помощью ключевого слова `descending`. Например, результаты следующего запроса будут выведены по убывающей остатков на счетах.

```
var accInfo = from acc in accounts
              orderby x.LastName, x.FirstName, x.Balance descending
              select acc;
```

В этом случае список счетов вкладчиков по фамилии Джонс будет выглядеть так, как показано ниже.

```
Джонс, Дженни     Номер счета: 108CK,    $10.98

Джонс, Ральф      Номер счета: 436CD, $1,923.85
Джонс, Ральф      Номер счета: 454MM,   $987.13
Джонс, Ральф      Номер счета: 434CK, ($123.32)
```

Как видите, теперь счета вкладчика по фамилии Ральф Джонс выводятся по убывающей: от наибольшей до наименьшей суммы остатка на счете.

Подробное рассмотрение оператора `select`

Оператор `select` определяет конкретный тип элементов, получаемых по запросу. Ниже приведена его общая форма.

```
select выражение
```

В предыдущих примерах оператор `select` использовался для возврата переменной диапазона. Поэтому *выражение* в нем просто обозначало имя переменной диапазона. Но применение оператора `select` не ограничивается только этой простой функцией. Он может также возвращать отдельную часть значения переменной диапазона, результат выполнения некоторой операции или преобразования переменной диапазона и даже новый тип объекта, конструируемого из отдельных фрагментов информации, извлекаемой из переменной диапазона. Такое преобразование исходных данных называется *проецированием*.

Начнем рассмотрение других возможностей оператора `select` с приведенной ниже программы. В этой программе выводятся квадратные корни положительных значений, содержащихся в массиве типа `double`.

```
// Использовать оператор select для возврата квадратных корней всех
// положительных значений, содержащихся в массиве типа double.

using System;
using System.Linq;

class SelectDemo {
  static void Main() {

    double[] nums =
     { -10.0, 16.4, 12.125, 100.85, -2.2, 25.25, -3.5 };

    // Сформировать запрос на получение квадратных корней всех
    // положительных значений, содержащихся в массиве nums.
    var sqrRoots = from n in nums
                   where n > 0
                   select Math.Sqrt(n);

    Console.WriteLine("Квадратные корни положительных значений,\n" +
                      "округленные до двух десятичных цифр:");

    // Выполнить запрос и вывести его результаты.
    foreach(double r in sqrRoots)
      Console.WriteLine("{0:#.##}", r);
  }
}
```

Эта программа дает следующий результат.

```
Квадратные корни положительных значений,
округленные до двух десятичных цифр:
4.05
3.48
10.04
5.02
```

Обратите особое внимание в данном примере запроса на следующий оператор `select`.

```
select Math.Sqrt(n);
```

Он возвращает квадратный корень значения переменной диапазона. Для этого значение переменной диапазона передается методу `Math.Sqrt()`, который возвращает

квадратный корень своего аргумента. Это означает, что последовательность результатов, получаемых при выполнении запроса, будет содержать квадратные корни положительных значений, хранящихся в массиве `nums`. Если обобщить этот принцип, то его эффективность станет вполне очевидной. Так, с помощью оператора `select` можно сформировать любой требующийся тип последовательности результатов, исходя из значений, получаемых из источника данных.

Ниже приведена программа, демонстрирующая другое применение оператора `select`. В этой программе сначала создается класс `EmailAddress`, содержащий два свойства. В первом из них хранится имя адресата, а во втором — адрес его электронной почты. Затем в этой программе создается массив, содержащий несколько элементов данных типа `EmailAddress`. И наконец, в данной программе создается список, состоящий только из адресов электронной почты, извлекаемых по запросу.

```
// Возвратить часть значения переменной диапазона.

using System;
using System.Linq;

class EmailAddress {
  public string Name { get; set; }
  public string Address { get; set; }

  public EmailAddress(string n, string a) {
    Name = n;
    Address = a;
  }
}

class SelectDemo2 {
  static void Main() {

    EmailAddress[] addrs = {
  new EmailAddress("Герберт", "Herb@HerbSchildt.com"),
  new EmailAddress("Том", "Tom@HerbSchildt.com"),
  new EmailAddress("Сара", "Sara@HerbSchildt.com")
    };

    // Сформировать запрос на получение адресов
    // электронной почты.
    var eAddrs = from entry in addrs
                 select entry.Address;

    Console.WriteLine("Адреса электронной почты:");

    // Выполнить запрос и вывести его результаты.
    foreach(string s in eAddrs)
      Console.WriteLine("  " + s);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Адреса электронной почты:
  Herb@HerbSchildt.com
```

```
Tom@HerbSchildt.com
Sara@HerbSchildt.com
```

Обратите особое внимание на следующий оператор `select`.

```
select entry.Address;
```

Вместо полного значения переменной диапазона этот оператор возвращает лишь его адресную часть (`Address`). Это означает, что по данному запросу возвращается последовательность символьных строк, а не объектов типа `EmailAddress`. Именно поэтому переменная `s` указывается в цикле `foreach` как `string`. Ведь как пояснялось ранее, тип последовательности результатов, возвращаемых по запросу, определяется типом значения, возвращаемым оператором `select`.

Одной из самых эффективных для оператора `select` является возможность возвращать последовательность результатов, содержащую элементы данных, формируемые во время выполнения запроса. В качестве примера рассмотрим еще одну программу. В ней определяется класс `ContactInfo`, в котором хранится имя, адрес электронной почты и номер телефона адресата. Кроме того, в этой программе определяется класс `EmailAddress`, использовавшийся в предыдущем примере. В методе `Main()` создается массив объектов типа `ContactInfo`, а затем объявляется запрос, в котором источником данных служит этот массив, но возвращаемая последовательность результатов содержит объекты типа `EmailAddress`. Таким образом, типом последовательности результатов, возвращаемой оператором `select`, является класс `EmailAddress`, а не класс `ContactInfo`, причем его объекты создаются во время выполнения запроса.

```
// Использовать запрос для получения последовательности объектов
// типа EmailAddresses из списка объектов типа ContactInfo.

using System;
using System.Linq;

class ContactInfo {
  public string Name { get; set; }
  public string Email { get; set; }

  public string Phone { get; set; }
  public ContactInfo(string n, string a, string p) {
    Name = n;
    Email = a;
    Phone = p;
  }
}

class EmailAddress {
  public string Name { get; set; }
  public string Address { get; set; }
  public EmailAddress(string n, string a) {
    Name = n;
    Address = a;
  }
}

class SelectDemo3 {
  static void Main() {
```

```
    ContactInfo[] contacts = {
      new ContactInfo("Герберт", "Herb@HerbSchildt.com", "555-1010"),
      new ContactInfo("Том", "Tom@HerbSchildt.com", "555-1101"),
      new ContactInfo("Сара", "Sara@HerbSchildt.com", "555-0110")
    };

    // Сформировать запрос на получение списка объектов типа EmailAddress.
    var emailList = from entry in contacts
                    select new EmailAddress(entry.Name, entry.Email);

    Console.WriteLine("Список адресов электронной почты:");

    // Выполнить запрос и вывести его результаты.
    foreach(EmailAddress e in emailList)
     Console.WriteLine("  {0}: {1}", e.Name, e.Address );
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Список адресов электронной почты:
  Герберт: Herb@HerbSchildt.com
  Том: Tom@HerbSchildt.com
  Сара: Sara@HerbSchildt.com
```

Обратите особое внимание в данном запросе на следующий оператор `select`.

```
select new EmailAddress(entry.Name, entry.Email);
```

В этом операторе создается новый объект типа `EmailAddress`, содержащий имя и адрес электронной почты, получаемые из объекта типа `ContactInfo`, хранящегося в массиве `contacts`. Но самое главное, что новые объекты типа `EmailAddress` создаются в операторе `select` во время выполнения запроса.

Применение вложенных операторов `from`

Запрос может состоять из нескольких операторов `from`, которые оказываются в этом случае вложенными. Такие операторы `from` находят применение в тех случаях, когда по запросу требуется получить данные из двух разных источников. Рассмотрим простой пример, в котором два вложенных оператора `from` используются в запросе для циклического обращения к элементам двух разных массивов символов. В итоге по такому запросу формируется последовательность результатов, содержащая все возможные комбинации двух наборов символов.

```
// Использовать два вложенных оператора from для составления списка
// всех возможных сочетаний букв A, B и C с буквами X, Y и Z.

using System;
using System.Linq;

// Этот класс содержит результат запроса.
class ChrPair {
  public char First;
```

```
  public char Second;

  public ChrPair(char c, char c2) {
    First = c;
    Second = c2;
  }
}

class MultipleFroms {
  static void Main() {

    char[] chrs = { 'A', 'B', 'C' };
    char[] chrs2 = { 'X', 'Y', 'Z' };

    // В первом операторе from организуется циклическое обращение
    // к массиву символов chrs, а во втором операторе from -
    // циклическое обращение к массиву символов chrs2.
    var pairs = from ch1 in chrs
                from ch2 in chrs2
                select new ChrPair(ch1, ch2);

    Console.WriteLine("Все сочетания букв ABC и XYZ: ");
    foreach(var p in pairs)
     Console.WriteLine("{0} {1}", p.First, p.Second);
  }
}
```

Выполнение этого кода приводит к следующему результату.

```
Все сочетания букв ABC и XYZ:
A X
A Y
A Z
B X
B Y
B Z
C X
C Y
C Z
```

Этот пример кода начинается с создания класса `ChrPair`, в котором содержатся результаты запроса. Затем в нем создаются два массива, `chrs` и `chrs2`, и, наконец, формируется следующий запрос для получения всех возможных комбинаций двух последовательностей результатов.

```
var pairs = from ch1 in chrs
            from ch2 in chrs2
            select new ChrPair(ch1, ch2);
```

Во вложенных операторах `from` организуется циклическое обращение к обоим массивам символов, `chrs` и `chrs2`. Сначала из массива `chrs` получается символ, сохраняемый в переменной `ch1`. Затем перечисляется содержимое массива `chrs2`. На каждом шаге циклического обращения во внутреннем операторе `from` символ из массива `chrs2` сохраняется в переменной `ch2` и далее выполняется оператор `select`. В результате выполнения оператора `select` создается новый объект типа `ChrPair`,

содержащий пару символов, которые сохраняются в переменных ch1 и ch2 на каждом шаге циклического обращения к массиву во внутреннем операторе from. А в конечном итоге получается объект типа ChrPair, содержащий все возможные сочетания извлекаемых символов.

Вложенные операторы from применяются также для циклического обращения к источнику данных, который содержится в другом источнике данных. Соответствующий пример приведен в разделе "Применение оператора let для создания временной переменной в запросе" далее в этой главе.

Группирование результатов с помощью оператора group

Одним из самых эффективных средств формирования запроса является оператор group, поскольку он позволяет группировать полученные результаты по ключам. Используя последовательность сгруппированных результатов, можно без особого труда получить доступ ко всем данным, связанным с ключом. Благодаря этому свойству оператора group доступ к данным, организованным в последовательности связанных элементов, осуществляется просто и эффективно. Оператор group является одним из двух операторов, которыми может оканчиваться запрос. (Вторым оператором, завершающим запрос, является select.) Ниже приведена общая форма оператора group.

```
group переменная_диапазона by ключ
```

Этот оператор возвращает данные, сгруппированные в последовательности, причем каждую последовательность обозначает общий *ключ*.

Результатом выполнения оператора group является последовательность, состоящая из элементов типа IGrouping<TKey, TElement>, т.е. обобщенного интерфейса, объявляемого в пространстве имен System.Linq. В этом интерфейсе определена коллекция объектов с общим ключом. Типом переменной запроса, возвращающего группу, является IEnumerable<IGrouping<TKey, TElement>>. В интерфейсе IGrouping определено также доступное только для чтения свойство Key, возвращающее ключ, связанный с каждой коллекцией.

Ниже приведен пример, демонстрирующий применение оператора group. В коде этого примера сначала объявляется массив, содержащий список веб-сайтов, а затем формируется запрос, в котором этот список группируется по имени домена самого верхнего уровня, например .org или .com.

```
// Продемонстрировать применение оператора group.

using System;
using System.Linq;

class GroupDemo {
  static void Main() {

    string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                          "hsNameD.com", "hsNameE.org", "hsNameF.org",
                          "hsNameG.tv", "hsNameH.net", "hsNameI.tv"
    };

    // Сформировать запрос на получение списка веб-сайтов,
    // группируемых по имени домена самого верхнего уровня.
```

```
    var webAddrs = from addr in websites
                   where addr.LastIndexOf('.') != -1
                   group addr by addr.Substring(addr.LastIndexOf('.'));

    // Выполнить запрос и вывести его результаты.
    foreach(var sites in webAddrs) {
      Console.WriteLine("Веб-сайты, сгруппированные " +
                        "по имени домена" + sites.Key);
      foreach(var site in sites)
        Console.WriteLine("  " + site);
      Console.WriteLine();
    }
  }
}
```

Вот к какому результату приводит выполнение этого кода.

```
Веб-сайты, сгруппированные по имени домена .com

  hsNameA.com
  hsNameD.com

Веб-сайты, сгруппированные по имени домена .net
  hsNameB.net
  hsNameC.net
  hsNameH.net

Веб-сайты, сгруппированные по имени домена .org
  hsNameE.org
  hsNameF.org

Веб-сайты, сгруппированные по имени домена .tv
  hsNameG.tv
  hsNameI.tv
```

Как следует из приведенного выше результата, данные, получаемые по запросу, группируются по имени домена самого верхнего уровня в адресе веб-сайта. Обратите внимание на то, как это делается в операторе `group` из следующего запроса.

```
var webAddrs = from addr in websites
               where addr.LastIndexOf('.') != -1
               group addr by addr.Substring(addr.LastIndexOf('.'));
```

Ключ в этом операторе создается с помощью методов `LastIndexOf()` и `Substring()`, определенных для данных типа `string`. (Эти методы упоминаются в главе 7, посвященной массивам и строкам. Вариант метода `Substring()`, используемый в данном примере, возвращает подстроку, начинающуюся с места, обозначаемого индексом, и продолжающуюся до конца вызывающей строки.) Индекс последней точки в адресе веб-сайта определяется с помощью метода `LastIndexOf()`. По этому индексу в методе `Substring()` создается оставшаяся часть строки, в которой содержится имя домена самого верхнего уровня. Обратите внимание на то, что в операторе `where` отсеиваются все строки, которые не содержат точку. Метод `LastIndexOf()` возвращает –1, если указанная подстрока не содержится в вызывающей строке.

Последовательность результатов, получаемых при выполнении запроса, хранящегося в переменной webAddrs, представляет собой список групп, поэтому для доступа к каждому члену группы требуются два цикла foreach. Доступ к каждой группе осуществляется во внешнем цикле, а члены внутри группы перечисляются во внутреннем цикле. Переменная шага внешнего цикла foreach должна быть экземпляром интерфейса IGrouping, совместимым с ключом и типом элемента данных. В рассматриваемом здесь примере ключи и элементы данных относятся к типу string. Поэтому переменная sites шага внешнего цикла имеет тип IGrouping<string, string>, а переменная site шага внутреннего цикла — тип string. Ради краткости данного примера обе переменные объявляются неявно, хотя их можно объявить и явным образом, как показано ниже.

```
foreach(IGrouping<string, string> sites in webAddrs) {
  Console.WriteLine("Веб-сайты, сгруппированные " +
                    "по имени домена" + sites.Key);
  foreach(string site in sites)
    Console.WriteLine("   " + site);
  Console.WriteLine();
}
```

Продолжение запроса с помощью оператора into

При использовании в запросе оператора select или group иногда требуется сформировать временный результат, который будет служить *продолжением запроса* для получения окончательного результата. Такое продолжение осуществляется с помощью оператора into в комбинации с оператором select или group. Ниже приведена общая форма оператора into:

```
into имя тело_запроса
```

где *имя* обозначает конкретное имя переменной диапазона, используемой для циклического обращения к временному результату в продолжении запроса, на которое указывает *тело_запроса*. Когда оператор into используется вместе с оператором select или group, то его называют продолжением запроса, поскольку он продолжает запрос. По существу, продолжение запроса воплощает в себе принцип построения нового запроса по результатам предыдущего.

ПРИМЕЧАНИЕ

Существует также форма оператора into, предназначенная для использования вместе с оператором join, создающим групповое объединение, о котором речь пойдет далее в этой главе.

Ниже приведен пример программы, в которой оператор into используется вместе с оператором group. Эта программа является переработанным вариантом предыдущего примера, в котором список веб-сайтов формируется по имени домена самого верхнего уровня. А в данном примере первоначальные результаты запроса сохраняются в переменной диапазона ws и затем отбираются для исключения всех групп, состоящих менее чем из трех элементов.

```
// Использовать оператор into вместе с оператором group.

using System;
using System.Linq;

class IntoDemo {
  static void Main() {

    string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                          "hsNameD.com", "hsNameE.org", "hsNameF.org",
                          "hsNameG.tv", "hsNameH.net", "hsNameI.tv"
    };

    // Сформировать запрос на получение списка веб-сайтов, группируемых
    // по имени домена самого верхнего уровня, но выбрать только те
    // группы, которые состоят более чем из двух членов.
    // Здесь ws — это переменная диапазона для ряда групп,
    // возвращаемых при выполнении первой половины запроса.
    var webAddrs = from addr in websites
                   where addr.LastIndexOf('.') != -1
                   group addr by addr.Substring(addr.LastIndexOf('.'))
                             into ws
                   where ws.Count() > 2
                   select ws;

    // Выполнить запрос и вывести его результаты.
     Console.WriteLine("Домены самого верхнего уровня " +
                       "с более чем двумя членами.\n");

    foreach(var sites in webAddrs) {
      Console.WriteLine("Содержимое домена: " + sites.Key);
      foreach(var site in sites)
        Console.WriteLine("  " + site);
      Console.WriteLine();
    }
  }
}
```

Эта программа дает следующий результат:

```
Домены самого верхнего уровня с более чем двумя членами.

Содержимое домена: .net
  hsNameB.net
  hsNameC.net
  hsNameH.net
```

Как следует из результата выполнения приведенной выше программы, по запросу возвращается только группа `.net`, поскольку это единственная группа, содержащая больше двух элементов.

Обратите особое внимание в данном примере программы на следующую последовательность операторов в формируемом запросе.

```
group addr by addr.Substring(addr.LastIndexOf('.'))
           into ws
where ws.Count() > 2
select ws;
```

Сначала результаты выполнения оператора `group` сохраняются как временные для последующей обработки оператором `where`. В качестве переменной диапазона в данный момент служит переменная `ws`. Она охватывает все группы, возвращаемые оператором `group`. Затем результаты запроса отбираются в операторе `where` с таким расчетом, чтобы в конечном итоге остались только те группы, которые содержат больше двух членов. Для этой цели вызывается метод `Count()`, который является *методом расширения* и реализуется для всех объектов типа `IEnumerable`. Он возвращает количество элементов в последовательности. (Подробнее о методах расширения речь пойдет далее в этой главе.) А получающаяся в итоге последовательность групп возвращается оператором `select`.

Применение оператора `let` для создания временной переменной в запросе

Иногда возникает потребность временно сохранить некоторое значение в самом запросе. Допустим, что требуется создать переменную перечислимого типа, которую можно будет затем запросить, или же сохранить некоторое значение, чтобы в дальнейшем использовать его в операторе `where`. Независимо от преследуемой цели, эти виды функций могут быть осуществлены с помощью оператора `let`. Ниже приведена общая форма оператора `let`:

```
let имя = выражение
```

где *имя* обозначает идентификатор, получающий значение, которое дает *выражение*. Тип имени выводится из типа выражения.

В приведенном ниже примере программы демонстрируется применение оператора `let` для создания еще одного перечислимого источника данных. В качестве входных данных в запрос вводится массив символьных строк, которые затем преобразуются в массивы типа `char`. Для этой цели служит еще один метод обработки строк, называемый `ToCharArray()` и возвращающий массив, содержащий символы в строке. Полученный результат присваивается переменной `chrArray`, которая затем используется во вложенном операторе `from` для извлечения отдельных символов из массива. И наконец, полученные символы сортируются в запросе, и из них формируется результирующая последовательность.

```
// Использовать оператор let в месте с вложенным оператором from.

using System;
using System.Linq;

class LetDemo {
  static void Main() {

    string[] strs = { "alpha", "beta", "gamma" };

    // Сформировать запрос на получение символов, возвращаемых из
    // строк в отсортированной последовательности. Обратите внимание
    // на применение вложенного оператора from.
    var chrs = from str in strs
               let chrArray = str.ToCharArray()
```

```
                from ch in chrArray
                orderby ch
                select ch;

    Console.WriteLine("Отдельные символы, отсортированные по порядку:");


    // Выполнить запрос и вывести его результаты.
    foreach(char c in chrs) Console.Write(c + " ");

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Отдельные символы, отсортированные по порядку:
a a a a a b e g h l m m p t
```

Обратите внимание в данном примере программы на то, что в операторе `let` переменной `chrArray` присваивается ссылка на массив, возвращаемый методом `str.ToCharArray()`.

```
let chrArray = str.ToCharArray()
```

После оператора `let` переменная `chrArray` может использоваться в остальных операторах, составляющих запрос. А поскольку все массивы в C# преобразуются в тип `IEnumerable<T>`, то переменную `chrArray` можно использовать в качестве источника данных для запроса во втором, вложенном операторе `from`. Именно это и происходит в рассматриваемом здесь примере, где вложенный оператор `from` служит для перечисления в массиве отдельных символов, которые затем сортируются по нарастающей и возвращаются в виде конечного результата.

Оператор `let` может также использоваться для хранения неперечислимого значения. В качестве примера ниже приведен более эффективный вариант формирования запроса в программе `IntoDemo` из предыдущего раздела.

```
var webAddrs = from addr in websites
               let idx = addr.LastIndexOf('.')
               where idx != -1
               group addr by addr.Substring(idx)
                          into ws
               where ws.Count() > 2
               select ws;
```

В этом варианте индекс последнего вхождения символа точки в строку присваивается переменной `idx`. Данное значение затем используется в методе `Substring()`. Благодаря этому исключается необходимость дважды искать символ точки в строке.

Объединение двух последовательностей с помощью оператора `join`

Когда приходится иметь дело с базами данных, то зачастую требуется формировать последовательность, увязывающую данные из разных источников. Например,

в Интернет-магазине может быть организована одна база данных, связывающая наименование товара с его порядковым номером, и другая база данных, связывающая порядковый номер товара с состоянием его запасов на складе. В подобной ситуации может возникнуть потребность составить список, в котором состояние запасов товаров на складе отображается по их наименованию, а не порядковому номеру. Для этой цели придется каким-то образом "увязать" данные из двух разных источников (баз данных). И это нетрудно сделать с помощью такого средства LINQ, как оператор `join`.

Ниже приведена общая форма оператора `join` (совместно с оператором `from`).

```
from переменная_диапазона_A in источник_данных_A
  join переменная_диапазона_B in источник_данных_B
    on переменная_диапазона_A.свойство equals переменная_диапазона_B.свойство
```

Применяя оператор `join`, следует иметь в виду, что каждый источник должен содержать общие данные, которые можно сравнивать. Поэтому в приведенной выше форме этого оператора *источник_данных_A* и *источник_данных_B* должны иметь нечто общее, что подлежит сравнению. Сравниваемые элементы данных указываются в части `on` данного оператора. Поэтому если *переменная_диапазона_A.свойство* и *переменная_диапазона_A.свойство* равны, то эти элементы данных "увязываются" успешно. По существу, оператор `join` выполняет роль своеобразного фильтра, отбирая только те элементы данных, которые имеют общее значение.

Как правило, оператор `join` возвращает последовательность, состоящую из данных, полученных из двух источников. Следовательно, с помощью оператора `join` можно сформировать новый список, состоящий из элементов, полученных из двух разных источников данных. Это дает возможность организовать данные по-новому.

Ниже приведена программа, в которой создается класс `Item`, инкапсулирующий наименование товара и его порядковый номер. Затем в этой программе создается еще один класс `InStockStatus`, связывающий порядковый номер товара с булевым свойством, которое указывает на наличие или отсутствие товара на складе. И наконец, в данной программе создается класс `Temp` с двумя полями: строковым (`string`) и булевым (`bool`). В объектах этого класса будут храниться результаты запроса. В этом запросе оператор `join` используется для получения списка, в котором наименование товара связывается с состоянием его запасов на складе.

```
// Продемонстрировать применение оператора join.

using System;
using System.Linq;

// Класс, связывающий наименование товара с его порядковым номером.
class Item {
  public string Name { get; set; }
  public int ItemNumber { get; set; }

  public Item(string n, int inum) {
    Name = n;
    ItemNumber = inum;
  }
}

// Класс, связывающий наименование товара с состоянием его запасов на складе.
class InStockStatus {
```

```
  public int ItemNumber { get; set; }
  public bool InStock { get; set; }

  public InStockStatus(int n, bool b) {
    ItemNumber = n;
    InStock = b;
  }
}

// Класс, инкапсулирующий наименование товара и
// состояние его запасов на складе.
class Temp {
  public string Name { get; set; }
  public bool InStock { get; set; }

  public Temp(string n, bool b) {
    Name = n;
    InStock = b;
  }
}

class JoinDemo {
  static void Main() {

    Item[] items = {
           new Item("Кусачки", 1424),
           new Item("Тиски", 7892),
           new Item("Молоток", 8534),
           new Item("Пила", 6411)
    };

    InStockStatus[] statusList = {
           new InStockStatus(1424, true),
           new InStockStatus(7892, false),
           new InStockStatus(8534, true),
           new InStockStatus(6411, true)
    };

    // Сформировать запрос, объединяющий объекты классов Item
    // и InStockStatus для составления списка наименований товаров
    // и их наличия на складе. Обратите внимание на формирование
    // последовательности объектов класса Temp.
    var inStockList = from item in items
                      join entry in statusList
                        on item.ItemNumber equals entry.ItemNumber
                      select new Temp(item.Name, entry.InStock);

    Console.WriteLine("Товар\tНаличие\n");

    // Выполнить запрос и вывести его результаты.
    foreach(Temp t in inStockList)
      Console.WriteLine("{0}\t{1}", t.Name, t.InStock);
  }
}
```

Эта программа дает следующий результат.

```
Товар      Наличие

Кусачки    True
Тиски      False
Молоток    True
Пила       True
```

Для того чтобы стал понятнее принцип действия оператора `join`, рассмотрим каждую строку запроса из приведенной выше программы по порядку. Этот запрос начинается, как обычно, со следующего оператора `from`.

```
var inStockList = from item in items
```

В этом операторе указывается переменная диапазона `item` для источника данных `items`, который представляет собой массив объектов класса `Item`. В классе `Item` инкапсулируются наименование товара и порядковый номер товара, хранящегося на складе.

Далее следует приведенный ниже оператор `join`.

```
join entry in statusList
  on item.ItemNumber equals entry.ItemNumber
```

В этом операторе указывается переменная диапазона `entry` для источника данных `statusList`, который представляет собой массив объектов класса `InStockStatus`, связывающего порядковый номер товара с состоянием его запасов на складе. Следовательно, у массивов `items` и `statusList` имеется общее свойство: порядковый номер товара. Именно это свойство используется в части `on/equals` оператора `join` для описания связи, по которой из двух разных источников данных выбираются наименования товаров, когда их порядковые номера совпадают.

И наконец, оператор `select` возвращает объект класса `Temp`, содержащий наименование товара и состояние его запасов на складе.

```
select new Temp(item.Name, entry.InStock);
```

Таким образом, последовательность результатов, получаемая по данному запросу, состоит из объектов типа `Temp`.

Рассмотренный здесь пример применения оператора `join` довольно прост. Тем не менее этот оператор поддерживает и более сложные операции с источниками данных. Например, используя совместно операторы `into` и `join`, можно создать *групповое объединение*, чтобы получить результат, состоящий из первой последовательности и группы всех совпадающих элементов из второй последовательности. (Соответствующий пример будет приведен далее в этой главе.) Как правило, время и усилия, затраченные на полное освоение оператора `join`, окупаются сторицей, поскольку он дает возможность распознавать данные во время выполнения программы. Это очень ценная возможность. Но она становится еще ценнее, если используются анонимные типы, о которых речь пойдет в следующем разделе.

Анонимные типы

В C# предоставляется средство, называемое *анонимным типом* и связанное непосредственно с LINQ. Как подразумевает само название, анонимный тип представляет

собой класс, не имеющий имени. Его основное назначение состоит в создании объекта, возвращаемого оператором `select`. Результатом запроса нередко оказывается последовательность объектов, которые составляются из членов, полученных из двух или более источников данных (как, например, в операторе `join`), или же включают в себя подмножество членов из одного источника данных. Но в любом случае тип возвращаемого объекта зачастую требуется только в самом запросе и не используется в остальной части программы. Благодаря анонимному типу в подобных случаях отпадает необходимость объявлять класс, который предназначается только для хранения результата запроса.

Анонимный тип объявляется с помощью следующей общей формы:

```
new { имя_A = значение_A, имя_B = значение_B, ... }
```

где имена обозначают идентификаторы, которые преобразуются в свойства, доступные только для чтения и инициализируемые значениями, как в приведенном ниже примере.

```
new { Count = 10, Max = 100, Min = 0 }
```

В данном примере создается класс с тремя открытыми только для чтения свойствами: `Count`, `Max` и `Min`, которым присваиваются значения 10, 100 и 0 соответственно. К этим свойствам можно обращаться по имени из другого кода. Следует заметить, что в анонимном типе используются инициализаторы объектов для установки их полей и свойств в исходное состояние. Как пояснялось в главе 8, инициализаторы объектов обеспечивают инициализацию объекта без явного вызова конструктора. Именно это и требуется для анонимных типов, поскольку явный вызов конструктора для них невозможен. (Напомним, что у конструкторов такое же имя, как и у их класса. Но у анонимного класса нет имени, а значит, и нет возможности вызвать его конструктор.)

Итак, у анонимного типа нет имени, и поэтому для обращения к нему приходится использовать неявно типизированную переменную. Это дает компилятору возможность вывести надлежащий тип. В приведенном ниже примере объявляется переменная `myOb`, которой присваивается ссылка на объект, создаваемый в выражении анонимного типа.

```
var myOb = new { Count = 10, Max = 100, Min = 0 }
```

Это означает, что следующие операторы считаются вполне допустимыми.

```
Console.WriteLine("Счет равен " + myOb.Count);

if(i <= myOb.Max && i >= myOb.Min) // ...
```

Напомним, что при создании объекта анонимного типа указываемые идентификаторы становятся свойствами, открытыми только для чтения. Поэтому их можно использовать в других частях кода.

Термин *анонимный тип* не совсем оправдывает свое название. Ведь тип оказывается анонимным только для программирующего, но не для компилятора, который присваивает ему внутреннее имя. Следовательно, анонимные типы не нарушают принятые в C# правила строгого контроля типов.

Для того чтобы стало более понятным особое назначение анонимных типов, рассмотрим переделанную версию программы из предыдущего раздела, посвященного оператору `join`. Напомним, что в этой программе класс `Temp` требовался для инкапсуляции результата, возвращаемого оператором `join`. Благодаря применению

анонимного типа необходимость в этом классе-заполнителе отпадает, а исходный код программы становится менее громоздким. Результат выполнения программы при этом не меняется.

```
// Использовать анонимный тип для усовершенствования
// программы, демонстрирующей применение оператора join.

using System;
using System.Linq;

// Класс, связывающий наименование товара с его порядковым номером.
class Item {
  public string Name { get; set; }
  public int ItemNumber { get; set; }

  public Item(string n, int inum) {
    Name = n;
    ItemNumber = inum;
  }
}

// Класс, связывающий наименование товара с состоянием его запасов на складе.
class InStockStatus {
  public int ItemNumber { get; set; }
  public bool InStock { get; set; }

  public InStockStatus(int n, bool b) {
    ItemNumber = n;
    InStock = b;
  }
}

class AnonTypeDemo {
  static void Main() {
    Item[] items = {
           new Item("Кусачки", 1424),
           new Item("Тиски", 7892),
           new Item("Молоток", 8534),
           new Item("Пила", 6411)
    };

    InStockStatus[] statusList = {
           new InStockStatus(1424, true),
           new InStockStatus(7892, false),
           new InStockStatus(8534, true),
           new InStockStatus(6411, true)
    };

    // Сформировать запрос, объединяющий объекты классов Item и
    // InStockStatus для составления списка наименований товаров и их
    // наличия на складе. Теперь для этой цели используется анонимный тип.
    var inStockList = from item in items
                      join entry in statusList
                        on item.ItemNumber equals entry.ItemNumber
```

```
                    select new { Name = item.Name,
                                 InStock = entry.InStock };

    Console.WriteLine("Товар\tНаличие\n");

    // Выполнить запрос и вывести его результаты.
    foreach(var t in inStockList)
      Console.WriteLine("{0}\t{1}", t.Name, t.InStock);
  }
}
```

Обратите особое внимание на следующий оператор `select`.

```
select new { Name = item.Name,
             InStock = entry.InStock };
```

Он возвращает объект анонимного типа с двумя доступными только для чтения свойствами: `Name` и `InStock`. Этим свойствам присваиваются наименование товара и состояние его наличия на складе. Благодаря применению анонимного типа необходимость в упоминавшемся выше классе `Temp` отпадает.

Обратите также внимание на цикл `foreach`, в котором выполняется запрос. Теперь переменная шага этого цикла объявляется с помощью ключевого слова `var`. Это необходимо потому, что у типа объекта, хранящегося в переменной `inStockList`, нет имени. Данная ситуация послужила одной из причин, по которым в C# были внедрены неявно типизированные переменные, поскольку они нужны для поддержки анонимных типов.

Прежде чем продолжить изложение, следует отметить еще один заслуживающий внимания аспект анонимных типов. В некоторых случаях, включая и рассмотренный выше, синтаксис анонимного типа упрощается благодаря применению *инициализатора проекции*. В данном случае просто указывается имя самого инициализатора. Это имя автоматически становится именем свойства. В качестве примера ниже приведен другой вариант оператора `select` из предыдущей программы.

```
select new { item.Name, entry.InStock };
```

В данном примере имена свойств остаются такими же, как и прежде, а компилятор автоматически "проецирует" идентификаторы `Name` и `InStock`, превращая их в свойства анонимного типа. Этим свойствам присваиваются прежние значения, обозначаемые `item.Name` и `entry.InStock` соответственно.

Создание группового объединения

Как пояснялось ранее, оператор `into` можно использовать вместе с оператором `join` для создания *группового объединения*, образующего последовательность, в которой каждый результат состоит из элементов данных из первой последовательности и группы всех совпадающих элементов из второй последовательности. Примеры группового объединения не приводились выше потому, что в этом объединении нередко применяется анонимный тип. Но теперь, когда представлены анонимные типы, можно обратиться к простому примеру группового объединения.

В приведенном ниже примере программы групповое объединение используется для составления списка, в котором различные транспортные средства (автомашины, суда и самолеты) организованы по общим для них категориям транспорта: назем-

ного, морского, воздушного и речного. В этой программе сначала создается класс Transport, связывающий вид транспорта с его классификацией. Затем в методе Main() формируются две входные последовательности. Первая из них представляет собой массив символьных строк, содержащих названия общих категорий транспорта: наземного, морского, воздушного и речного, а вторая — массив объектов типа Transport, инкапсулирующих различные транспортные средства. Полученное в итоге групповое объединение используется для составления списка транспортных средств, организованных по соответствующим категориям.

```
// Продемонстрировать применение простого группового объединения.

using System;
using System.Linq;

// Этот класс связывает наименование вида транспорта,
// например поезда, с общей классификацией транспорта:
// наземного, морского, воздушного или речного.
class Transport {
  public string Name { get; set; }
  public string How { get; set; }

  public Transport(string n, string h) {
    Name = n;
    How = h;
  }
}

class GroupJoinDemo {
  static void Main() {

    // Массив классификации видов транспорта.
    string[] travelTypes = {
             "Воздушный",
             "Морской",
             "Наземный",
             "Речной",
    };

    // Массив видов транспорта.
    Transport[] transports = {
                new Transport("велосипед", "Наземный"),
                new Transport("аэростат", "Воздушный"),
                new Transport("лодка", "Речной"),
                new Transport("самолет", "Воздушный"),
                new Transport("каноэ", "Речной"),
                new Transport("биплан", "Воздушный"),
                new Transport("автомашина", "Наземный"),
                new Transport("судно", "Морской"),
                new Transport("поезд", "Наземный")
    };

    // Сформировать запрос, в котором групповое
    // объединение используется для составления списка
```

```
    // видов транспорта по соответствующим категориям.
    var byHow = from how in travelTypes
                      join trans in transports
                      on how equals trans.How
                      into lst
                    select new { How = how, Tlist = lst };

    // Выполнить запрос и вывести его результаты.
    foreach(var t in byHow) {
      Console.WriteLine("К категории <{0} транспорт> относится:", t.How);

      foreach(var m in t.Tlist)
        Console.WriteLine("  " + m.Name);

      Console.WriteLine();
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
К категории <Воздушный транспорт> относится:
  аэростат
  самолет
  биплан

К категории <Морской транспорт> относится:
  судно

К категории <Наземный транспорт> относится:
  велосипед
  автомашина
  поезд

К категории <Речной транспорт> относится:
  лодка
  каноэ
```

Главной частью данной программы, безусловно, является следующий запрос.

```
var byHow = from how in travelTypes
                  join trans in transports
                  on how equals trans.How
                  into lst
                select new { How = how, Tlist = lst };
```

Этот запрос формируется следующим образом. В операторе `from` используется переменная диапазона `how` для охвата всего массива `travelTypes`. Напомним, что массив `travelTypes` содержит названия общих категорий транспорта: воздушного, наземного, морского и речного. Каждый вид транспорта объединяется в операторе `join` со своей категорией. Например, велосипед, автомашина и поезд объединяются с наземным транспортом. Но благодаря оператору `into` для каждой категории транспорта в операторе `join` составляется список видов транспорта, относящихся к данной категории. Этот список сохраняется в переменной `lst`. И наконец, оператор `select` возвращает объект анонимного типа, инкапсулирующий каждое значение перемен-

ной how (категории транспорта) вместе со списком видов транспорта. Именно поэтому для вывода результатов запроса требуются два цикла foreach.

```
foreach(var t in byHow) {
  Console.WriteLine("К категории <{0} транспорт> относится:", t.How);

    foreach(var m in t.Tlist)
      Console.WriteLine("  " + m.Name);

  Console.WriteLine();
}
```

Во внешнем цикле получается объект, содержащий наименование общей категории транспорта, и список видов транспорта, относящихся к этой категории. А во внутреннем цикле выводятся отдельные виды транспорта.

Методы запроса

Синтаксис запроса, описанный в предыдущих разделах, применяется при формировании большинства запросов в C#. Он удобен, эффективен и компактен, хотя и не является единственным способом формирования запросов. Другой способ состоит в использовании *методов запроса*, которые могут вызываться для любого перечислимого объекта, например массива.

Основные методы запроса

Методы запроса определяются в классе System.Linq.Enumerable и реализуются в виде *методов расширения* функций обобщенной формы интерфейса IEnumerable<T>. (Методы запроса определяются также в классе System.Linq.Queryable, расширяющем функции обобщенной формы интерфейса IQueryable<T>, но этот интерфейс в настоящей главе не рассматривается.) Метод расширения дополняет функции другого класса, но без наследования. Поддержка методов расширения была внедрена в версию C# 3.0 и более подробно рассматривается далее в этой главе. А до тех пор достаточно сказать, что методы запроса могут вызываться только для тех объектов, которые реализуют интерфейс IEnumerable<T>.

В классе Enumerable предоставляется немало методов запроса, но основными считаются те методы, которые соответствуют описанным ранее операторам запроса. Эти методы перечислены ниже вместе с соответствующими операторами запроса. Следует, однако, иметь в виду, что эти методы имеют также перегружаемые формы, а здесь они представлены лишь в самой простой своей форме. Но именно эта их форма используется чаще всего.

Оператор запроса	Эквивалентный метод запроса
select	Select(*selector*)
where	Where(*predicate*)
orderby	OrderBy(*keySelector*) или OrderByDescending(*keySelector*)
join	Join(*inner*, *outerKeySelector*, *innerKeySelector*, *resultSelector*)
group	GroupBy(*keySelector*)

За исключением метода `Join()`, остальные методы запроса принимают единственный аргумент, который представляет собой объект некоторой разновидности обобщенного типа `Func<T, TResult>`. Это тип встроенного делегата, объявляемый следующим образом:

```
delegate TResult Func<in T, out TResult>(T arg)
```

где `TResult` обозначает тип результата, который дает делегат, а `T` — тип элемента. В методах запроса аргументы *selector*, *predicate* или *keySelector* определяют действие, которое предпринимает метод запроса. Например, в методе `Where()` аргумент *predicate* определяет порядок отбора данных в запросе. Каждый метод запроса возвращает перечислимый объект. Поэтому результат выполнения одного метода запроса можно использовать для вызова другого, соединяя эти методы в цепочку.

Метод `Join()` принимает четыре аргумента. Первый аргумент (*inner*) представляет собой ссылку на вторую объединяемую последовательность, а первой является последовательность, для которой вызывается метод `Join()`. Селектор ключа для первой последовательности передается в качестве аргумента *outerKeySelector*, а селектор ключа для второй последовательности — в качестве аргумента *innerKeySelector*. Результат объединения обозначается как аргумент *resultSelector*. Аргумент *outerKeySelector* имеет тип `Func<TOuter, TKey>`, аргумент *innerKeySelector* — тип `Func<TInner, TKey>`, тогда как аргумент *resultSelector* — тип `Func<TOuter, TInner, TResult>`, где `TOuter` — тип элемента из вызывающей последовательности; `TInner` — тип элемента из передаваемой последовательности; `TResult` — тип элемента из объединяемой в итоге последовательности, возвращаемой в виде перечислимого объекта.

Аргумент метода запроса представляет собой метод, совместимый с указываемой формой делегата `Func`, но он не обязательно должен быть явно объявляемым методом. На самом деле вместо него чаще всего используется лямбда-выражение. Как пояснялось в главе 15, лямбда-выражение обеспечивает более простой, но эффективный способ определения того, что, по существу, является анонимным методом, а компилятор C# автоматически преобразует лямбда-выражение в форму, которая может быть передана в качестве параметра делегату `Func`. Благодаря тому что лямбда-выражения обеспечивают более простой и рациональный способ программирования, они используются во всех примерах, представленных далее в этом разделе.

Формирование запросов с помощью методов запроса

Используя методы запроса одновременно с лямбда-выражениями, можно формировать запросы, вообще не пользуясь синтаксисом, предусмотренным в C# для запросов. Вместо этого достаточно вызвать соответствующие методы запроса. Обратимся сначала к простому примеру. Он представляет собой вариант первого примера программы из этой главы, переделанный с целью продемонстрировать применение методов запроса `Where()` и `Select()` вместо соответствующих операторов.

```
// Использовать методы запроса для формирования простого запроса.
// Это переделанный вариант первого примера программы из настоящей главы.

using System;
using System.Linq;

class SimpQuery {
  static void Main() {
```

```
    int[] nums = { 1, -2, 3, 0, -4, 5 };

    // Использовать методы Where() и Select() для
    // формирования простого запроса.
    var posNums = nums.Where(n => n > 0).Select(r => r);

    Console.Write("Положительные значения из массива nums: ");

    // Выполнить запрос и вывести его результаты.
    foreach(int i in posNums) Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

Эта версия программы дает такой же результат, как и исходная.

```
Положительные значения из массива nums: 1 3 5
```

Обратите особое внимание в данной программе на следующую строку кода.

```
var posNums = nums.Where(n => n > 0).Select(r => r);
```

В этой строке кода формируется запрос, сохраняемый в переменной `posNums`. По этому запросу, в свою очередь, формируется последовательность положительных значений, извлекаемых из массива `nums`. Для этой цели служит метод `Where()`, отбирающий запрашиваемые значения, а также метод `Select()`, избирательно формирующий из этих значений окончательный результат. Метод `Where()` может быть вызван для массива `nums`, поскольку во всех массивах реализуется интерфейс `IEnumerable<T>`, поддерживающий методы расширения запроса.

Формально метод `Select()` в рассматриваемом здесь примере не нужен, поскольку это простой запрос. Ведь последовательность, возвращаемая методом `Where()`, уже содержит конечный результат. Но окончательный выбор можно сделать и по более сложному критерию, как это было показано ранее на примерах использования синтаксиса запросов. Так, по приведенному ниже запросу из массива `nums` возвращаются положительные значения, увеличенные на порядок величины.

```
var posNums = nums.Where(n => n > 0).Select(r => r * 10);
```

Как и следовало ожидать, в цепочку можно объединять и другие операции над данными, получаемыми по запросу. Например, по следующему запросу выбираются положительные значения, которые затем сортируются по убывающей и возвращаются в виде результирующей последовательности:

```
var posNums = nums.Where(n => n > 0).OrderByDescending(j => j);
```

где выражение `j => j` обозначает, что упорядочение зависит от входного параметра, который является элементом данных из последовательности, получаемой из метода `Where()`.

В приведенном ниже примере демонстрируется применение метода запроса `GroupBy()`. Это измененный вариант представленного ранее примера.

```
// Продемонстрировать применение метода запроса GroupBy().
// Это переработанный вариант примера, представленного ранее
// для демонстрации синтаксиса запросов.

using System;
using System.Linq;
```

```
class GroupByDemo {
  static void Main() {

    string[] websites = {
          "hsNameA.com", "hsNameB.net", "hsNameC.net",
          "hsNameD.com", "hsNameE.org", "hsNameF.org",
          "hsNameG.tv", "hsNameH.net", "hsNameI.tv"
    };

    // Использовать методы запроса для группирования
    // веб-сайтов по имени домена самого верхнего уровня.
    var webAddrs = websites.Where(w => w.LastIndexOf('.') != 1).
        GroupBy(x => x.Substring(x.LastIndexOf(".", x.Length)));

    // Выполнить запрос и вывести его результаты.
    foreach(var sites in webAddrs) {
      Console.WriteLine("Веб-сайты, сгруппированные " +
                        "по имени домена " + sites.Key);
        foreach(var site in sites)
          Console.WriteLine("  " + site);
      Console.WriteLine();
    }
  }
}
```

Эта версия программы дает такой же результат, как и предыдущая. Единственное отличие между ними заключается в том, как формируется запрос. В данной версии для этой цели используются методы запроса.

Рассмотрим другой пример. Но сначала приведем еще раз запрос из представленного ранее примера применения оператора `join`.

```
var inStockList = from item in items
                  join entry in statusList
                    on item.ItemNumber equals entry.ItemNumber
                  select new Temp(item.Name, entry.InStock);
```

По этому запросу формируется последовательность, состоящая из объектов, инкапсулирующих наименование товара и состояние его запасов на складе. Вся эта информация получается путем объединения двух источников данных: `items` и `statusList`. Ниже приведен переделанный вариант данного запроса, в котором вместо синтаксиса, предусмотренного в C# для запросов, используется метод запроса `Join()`.

```
// Использовать метод запроса Join() для составления списка
// наименований товаров и состояния их запасов на складе.
var inStockList = items.Join(statusList,
                    k1 => k1.ItemNumber,
                    k2 => k2.ItemNumber,
                    (k1, k2) => new Temp(k1.Name, k2.InStock) );
```

В данном варианте именованный класс `Temp` используется для хранения результирующего объекта, но вместо него можно воспользоваться анонимным типом. Такой вариант запроса приведен ниже.

```
var inStockList = items.Join(statusList,
                    k1 => k1.ItemNumber,
```

```
k2 => k2.ItemNumber,
(k1, k2) => new { k1.Name, k2.InStock} );
```

Синтаксис запросов и методы запроса

Как пояснялось в предыдущем разделе, запросы в C# можно формировать двумя способами, используя синтаксис запросов или методы запроса. Любопытно, что оба способа связаны друг с другом более тесно, чем кажется, глядя на исходный код программы. Дело в том, что синтаксис запросов компилируется в вызовы методов запроса. Поэтому код

```
where x < 10
```

будет преобразован компилятором в следующий вызов.

```
Where(x => x < 10)
```

Таким образом, оба способа формирования запросов в конечном итоге сходятся на одном и том же.

Но если оба способа оказываются в конечном счете равноценными, то какой из них лучше для программирования на C#? В целом, рекомендуется чаще пользоваться синтаксисом запросов, поскольку он полностью интегрирован в язык C#, поддерживается соответствующими ключевыми словами и синтаксическими конструкциями.

Дополнительные методы расширения, связанные с запросами

Помимо методов, соответствующих операторам запроса, поддерживаемым в C#, имеется ряд других методов расширения, связанных с запросами и зачастую оказывающих помощь в формировании запросов. Эти методы предоставляются в среде .NET Framework и определены для интерфейса `IEnumerable<T>` в классе `Enumerable`. Ниже приведены наиболее часто используемые методы расширения, связанные с запросами. Многие из них могут перегружаться, поэтому они представлены лишь в самой общей форме.

Метод	Описание
`All(`*predicate*`)`	Возвращает логическое значение `true`, если все элементы в последовательности удовлетворяют условию, задаваемому параметром *predicate*
`Any(`*predicate*`)`	Возвращает логическое значение `true`, если любой элемент в последовательности удовлетворяет условию, задаваемому параметром *predicate*
`Average()`	Возвращает среднее всех значений в числовой последовательности
`Contains(`*value*`)`	Возвращает логическое значение `true`, если в последовательности содержится указанный объект
`Count()`	Возвращает длину последовательности, т.е. количество составляющих ее элементов
`First()`	Возвращает первый элемент в последовательности
`Last()`	Возвращает последний элемент в последовательности
`Max()`	Возвращает максимальное значение в последовательности
`Min()`	Возвращает минимальное значение в последовательности
`Sum()`	Возвращает сумму значений в числовой последовательности

Метод `Count()` уже демонстрировался ранее в этой главе. А в следующей программе демонстрируются остальные методы расширения, связанные с запросами.

```
// Использовать ряд методов расширения, определенных в классе Enumerable.

using System;
using System.Linq;

class ExtMethods {
  static void Main() {

    int[] nums = { 3, 1, 2, 5, 4 };

    Console.WriteLine("Минимальное значение равно " + nums.Min());
    Console.WriteLine("Максимальное значение равно " + nums.Max());

    Console.WriteLine("Первое значение равно " + nums.First());
    Console.WriteLine("Последнее значение равно " + nums.Last());

    Console.WriteLine("Суммарное значение равно " + nums.Sum());
    Console.WriteLine("Среднее значение равно " + nums.Average());

    if(nums.All(n => n > 0))
      Console.WriteLine("Все значения больше нуля.");

    if(nums.Any(n => (n % 2) == 0))
      Console.WriteLine("По крайней мере одно значение является четным.");

    if(nums.Contains(3))
      Console.WriteLine("Массив содержит значение 3.");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Минимальное значение равно 1
Максимальное значение равно 5
Первое значение равно 3
Последнее значение равно 4
Суммарное значение равно 15
Среднее значение равно 3
Все значения больше нуля.
По крайней мере одно значение является четным
Массив содержит значение 3.
```

Методы расширения, связанные с запросами, можно также использовать в самом запросе, основываясь на синтаксисе запросов, предусмотренном в C#. И в действительности это делается очень часто. Например, метод `Average()` используется в приведенной ниже программе для получения последовательности, состоящей только из тех значений, которые оказываются меньше среднего всех значений в массиве.

```
// Использовать метод Average() вместе с синтаксисом запросов.

using System;
using System.Linq;

class ExtMethods2 {
  static void Main() {
```

```
    int[] nums = { 1, 2, 4, 8, 6, 9, 10, 3, 6, 7 };

    var ltAvg = from n in nums
                let x = nums.Average()
                where n < x
                select n;

    Console.WriteLine("Среднее значение равно " + nums.Average());

    Console.Write("Значения меньше среднего: ");

    // Выполнить запрос и вывести его результаты.
    foreach(int i in ltAvg) Console.Write(i + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
Среднее значение равно 5.6
Значения меньше среднего: 1 2 4 3
```

Обратите особое внимание в этой программе на следующий код запроса.

```
var ltAvg = from n in nums
            let x = nums.Average()
            where n < x
            select n;
```

Как видите, переменной `x` в операторе `let` присваивается среднее всех значений в массиве `nums`. Это значение получается в результате вызова метода `Average()` для массива `nums`.

Режимы выполнения запросов: отложенный и немедленный

В LINQ запросы выполняются в двух разных режимах: немедленном и отложенном. Как пояснялось ранее в этой главе, при формировании запроса определяется ряд правил, которые не выполняются вплоть до оператора цикла `foreach`. Это так называемое *отложенное выполнение*.

Но если используются методы расширения, дающие результат, отличающийся от последовательности, то запрос должен быть выполнен для получения этого результата. Рассмотрим, например, метод расширения `Count()`. Для того чтобы этот метод возвратил количество элементов в последовательности, необходимо выполнить запрос, и это делается автоматически при вызове метода `Count()`. В этом случае имеет место *немедленное выполнение*, когда запрос выполняется автоматически для получения требуемого результата. Таким образом, запрос все равно выполняется, даже если он не используется явно в цикле `foreach`.

Ниже приведен простой пример программы для получения количества положительных элементов, содержащихся в последовательности.

```
// Использовать режим немедленного выполнения запроса.

using System;
```

```
using System.Linq;

class ImmediateExec {
  static void Main() {

    int[] nums = { 1, -2, 3, 0, -4, 5 };

    // Сформировать запрос на получение количества
    // положительных значений в массиве nums.
    int len = (from n in nums
               where n > 0
               select n).Count();

    Console.WriteLine("Количество положительных значений в массиве nums: " +
                      len);
  }
}
```

Эта программа дает следующий результат.

```
Количество положительных значений в массиве nums: 3
```

Обратите внимание на то, что цикл `foreach` не указан в данной программе явным образом. Вместо этого запрос выполняется автоматически благодаря вызову метода расширения `Count()`.

Любопытно, что запрос из приведенной выше программы можно было бы сформировать и следующим образом.

```
var posNums = from n in nums
              where n > 0
              select n;

int len = posNums.Count(); // запрос выполняется здесь
```

В данном случае метод `Count()` вызывается для переменной запроса. И в этот момент запрос выполняется для получения подсчитанного количества.

К числу других методов расширения, вызывающих немедленное выполнение запроса, относятся методы `ToArray()` и `ToList()`. Оба этих метода расширения определены в классе `Enumerable`. Метод `ToArray()` возвращает результаты запроса в массиве, а метод `ToList()` — результаты запроса в форме коллекции `List`. (Подробнее о коллекциях речь пойдет в главе 25.) В обоих случаях для получения результатов выполняется запрос. Например, в следующем фрагменте кода сначала получается массив результатов, сформированных по приведенному выше запросу в переменной `posNums`, а затем эти результаты выводятся на экран.

```
int[] pnums = posNum.ToArray(); // запрос выполняется здесь

foreach(int i in pnums)
  Console.Write(i + " ");
}
```

Деревья выражений

Еще одним средством, связанным с LINQ, является *дерево выражений*, которое представляет лямбда-выражение в виде данных. Это означает, что само лямбда-выражение

нельзя выполнить, но можно преобразовать в исполняемую форму. Деревья выражений инкапсулируются в классе `System.Linq.Expressions.Expression<TDelegate>`. Они оказываются пригодными в тех случаях, когда запрос выполняется вне программы, например средствами SQL в базе данных. Если запрос представлен в виде данных, то его можно преобразовать в формат, понятный для базы данных. Этот процесс выполняется, например, средствами LINQ to SQL в интегрированной среде разработки Visual Studio. Таким образом, деревья выражений способствуют поддержке в C# различных баз данных.

Для получения исполняемой формы дерева выражений достаточно вызвать метод `Compile()`, определенный в классе `Expression`. Этот метод возвращает ссылку, которая может быть присвоена делегату для последующего выполнения. А тип делегата может быть объявлен собственным или же одним из предопределенных типов делегата `Func` в пространстве имен `System`. Две формы делегата `Func` уже упоминались ранее при рассмотрении методов запроса, но существует и другие его формы.

Деревьям выражений присуще следующее существенное ограничение: они могут представлять только одиночные лямбда-выражения. С их помощью нельзя представить блочные лямбда-выражения.

Ниже приведен пример программы, демонстрирующий конкретное применение дерева выражений. В этой программе сначала создается дерево выражений, данные которого представляют метод, определяющий, является ли одно целое число множителем другого. Затем это дерево выражений компилируется в исполняемый код. И наконец, в этой программе демонстрируется выполнение скомпилированного кода.

```
// Пример простого дерева выражений.

using System;
using System.Linq;
using System.Linq.Expressions;

class SimpleExpTree {
  static void Main() {

    // Представить лямбда-выражение в виде данных.
    Expression<Func<int, int, bool>>
      IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;

    // Скомпилировать данные выражения в исполняемый код.
    Func<int, int, bool> IsFactor = IsFactorExp.Compile();

    // Выполнить выражение.
    if(IsFactor(10, 5))
     Console.WriteLine("Число 5 является множителем 10.");

    if(!IsFactor(10, 7))
      Console.WriteLine("Число 7 не является множителем 10.");

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Число 5 является множителем 10.
Число 7 не является множителем 10.
```

Данный пример программы наглядно показывает два основных этапа применения дерева выражений. Сначала в ней создается дерево выражений с помощью следующего оператора.

```
Expression<Func<int, int, bool>>
  IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;
```

В этом операторе конструируется представление лямбда-выражения в оперативной памяти. Как пояснялось выше, это представление доступно по ссылке, присваиваемой делегату `IsFactorExp`. А в следующем операторе данные выражения преобразуются в исполняемый код.

```
Func<int, int, bool> IsFactor = IsFactorExp.Compile();
```

После выполнения этого оператора делегат `IsFactorExp` может быть вызван, чтобы определить, является ли одно целое число множителем другого.

Обратите также внимание на то, что `<Func<int, int, bool>` обозначает тип делегата. В этой форме делегата `Func` указываются два параметра типа `int` и возвращаемый тип `bool`. В рассматриваемой здесь программе использована именно эта форма делегата `Func`, совместимая с лямбда-выражениями, поскольку для выражения требуются два параметра. Для других лямбда-выражений могут подойти иные формы делегата `Func` в зависимости от количества требуемых параметров. Вообще говоря, конкретная форма делегата `Func` должна удовлетворять требованиям лямбда-выражения.

Методы расширения

Как упоминалось выше, методы расширения предоставляют средства для расширения функций класса, не прибегая к обычному механизму наследования. Методы расширения создаются нечасто, поскольку механизм наследования, как правило, предлагает лучшее решение. Тем не менее знать, как они действуют, никогда не помешает. Ведь они имеют существенное значение для LINQ.

Метод расширения является статическим и поэтому должен быть включен в состав статического, необобщенного класса. Тип первого параметра метода расширения определяет тип объектов, для которых этот метод может быть вызван. Кроме того, первый параметр может быть указан с модификатором `this`. Объект, для которого вызывается метод расширения, автоматически передается его первому параметру. Он не передается явным образом в списке аргументов. Следует, однако, иметь в виду, что метод расширения может по-прежнему вызываться для объекта аналогично методу экземпляра, несмотря на то, что он объявляется как статический.

Ниже приведена общая форма метода расширения.

```
static возвращаемый_тип имя(this тип_вызывающего_объекта ob, список_параметров)
```

Очевидно, что *список_параметров* окажется пустым в отсутствие аргументов, за исключением аргумента, неявно передаваемого вызывающим объектом *ob*. Не следует, однако, забывать, что первым параметром метода расширения является автоматически передаваемый объект, для которого вызывается этот метод. Как правило, метод расширения становится открытым членом своего класса.

В приведенном ниже примере программы создаются три простых метода расширения.

```
// Создать и использовать ряд методов расширения.

using System;
using System.Globalization;
static class MyExtMeths {

  // Возвратить обратную величину числового значения типа double.
  public static double Reciprocal(this double v) {
    return 1.0 / v;
  }

  // Изменить на обратный регистр букв в символьной
  // строке и возвратить результат.
  public static string RevCase(this string str) {
    string temp = "";

    foreach(char ch in str) {
      if(Char.IsLower(ch)) temp += Char.ToUpper(ch, CultureInfo.
CurrentCulture);
      else temp += Char.ToLower(ch, CultureInfo.CurrentCulture);
    }
    return temp;
  }

  // Возвратить абсолютное значение выражения n / d.
  public static double AbsDivideBy(this double n, double d) {
    return Math.Abs(n / d);
  }
}

class ExtDemo {
  static void Main() {
    double val = 8.0;
    string str = "Alpha Beta Gamma";

    // Вызвать метод расширения Reciprocal()..
    Console.WriteLine("Обратная величина {0} равна {1}",
                      val, val.Reciprocal());

    // Вызвать метод расширения RevCase().
    Console.WriteLine(str + " после смены регистра: " +
                      str.RevCase());

    // Использовать метод расширения AbsDivideBy().
    Console.WriteLine("Результат вызова метода val.AbsDivideBy(-2): " +
                      val.AbsDivideBy(-2));
  }
}
```

Эта программа дает следующий результат.

```
Обратная величина 8 равна 0.125
Alpha Beta Gamma после смены регистра: aLPHA bETA gAMMA
Результат вызова метода val.AbsDivideBy(-2): 4
```

В данном примере программы каждый метод расширения содержится в статическом классе `MyExtMeths`. Как пояснялось выше, метод расширения должен быть объявлен в статическом классе. Более того, этот класс должен находиться в области действия своих методов расширения, чтобы ими можно было пользоваться. (Именно поэтому в исходный текст программы следует включить пространство имен `System.Linq`, так как это дает возможность пользоваться методами расширения, связанными с LINQ.)

Объявленные методы расширения вызываются для объекта таким же образом, как и методы экземпляра. Главное отличие заключается в том, что вызывающий объект передается первому параметру метода расширения. Поэтому при выполнении выражения

```
val.AbsDivideBy(-2)
```

объект `val` передается параметру `n` метода расширения `AbsDivideBy()`, а значение `-2` — параметру `d`.

Любопытно, что методы расширения `Reciprocal()` и `AbsDivideBy()` могут вполне законно вызываться и для литерала типа `double`, как показано ниже, поскольку они определены для этого типа данных.

```
8.0.Reciprocal()
8.0.AbsDivideBy(-1)
```

Кроме того, метод расширения `RevCase()` может быть вызван следующим образом.

```
"AbCDe".RevCase()
```

В данном случае возвращается строковый литерал с измененным на обратный регистром букв.

PLINQ

В версии .NET Framework 4.0 внедрено новое дополнение LINQ под названием PLINQ. Это средство предназначено для поддержки параллельного программирования. Оно позволяет автоматически задействовать в запросе несколько доступных процессоров. Подробнее о PLINQ и других средствах, связанных с параллельным программированием, речь пойдет в главе 24.

20
ГЛАВА

Небезопасный код, указатели, обнуляемые типы и разные ключевые слова

В этой главе рассматривается средство языка C#, которое обычно захватывает программистов врасплох. Это небезопасный код. В таком коде зачастую используются указатели. Совместно с небезопасным кодом указатели позволяют разрабатывать на C# приложения, которые обычно связываются с языком C++, высокой производительностью и системным кодом. Более того, благодаря включению небезопасного кода и указателей в состав C# в этом языке появились возможности, которые отсутствуют в Java.

В этой главе рассматриваются также обнуляемые типы, определения частичных классов и методов, буферы фиксированного размера. И в заключение этой главы представлен ряд ключевых слов, не упоминавшихся в предыдущих главах.

Небезопасный код

В C# разрешается писать так называемый "небезопасный" код. В этом странном на первый взгляд утверждении нет на самом деле ничего необычного. Небезопасным считается не плохо написанный код, а такой код, который не может быть выполнен под полным управлением в общеязыковой исполняющей среде (CLR). Как пояснялось в главе 1, результатом программирования на C# обычно является управляемый код. Тем не менее этот язык программирования допускает написание кода, который не выполняется под полным управлением в среде CLR. Такой неуправляемый код не подчиняется тем же самым средствам

управления и ограничениям, что и управляемый код, и называется он небезопасным потому, что нельзя никак проверить, не выполняет ли он какое-нибудь опасное действие. Следовательно, термин *небезопасный* совсем не означает, что коду присущи какие-то изъяны. Это просто означает, что код может выполнять действия, которые не подлежат контролю в управляемой среде.

Если небезопасный код может вызвать осложнения, то зачем вообще создавать такой код? Дело в том, что управляемый код не позволяет использовать указатели. Если у вас имеется некоторый опыт программирования на C или C++, то вам должно быть известно, что *указатели* представляют собой переменные, предназначенные для хранения адресов других объектов, т.е. они в какой-то степени похожи на ссылки в C#. Главное отличие указателя заключается в том, что он может указывать на любую область памяти, тогда как ссылка всегда указывает на объект своего типа. Но поскольку указатель способен указывать практически на любую область памяти, то существует большая вероятность его неправильного использования. Кроме того, используя указатели, легко допустить программные ошибки. Именно поэтому указатели не поддерживаются при создании управляемого кода в C#. А поскольку указатели все-таки полезны и необходимы для некоторых видов программирования (например, утилит системного уровня), в C# разрешается создавать и использовать их. Но при этом все операции с указателями должны быть помечены как небезопасные, потому что они выполняются вне управляемой среды.

В языке C# указатели объявляются и используются таким же образом, как и в C/C++. Если вы знаете, как пользоваться ими в C/C++, то вам нетрудно будет сделать это и в C#. Но не забывайте, что главное назначение C# — создание управляемого кода. А способность этого языка программирования поддерживать неуправляемый код следует использовать для решения лишь особого рода задач. Это, скорее, исключение, чем правило для программирования на C#. По существу, для компилирования неуправляемого кода следует использовать параметр компилятора `/unsafe`.

Указатели составляют основу небезопасного кода, поэтому мы начнем его рассмотрение именно с них.

Основы применения указателей

Указатель представляет собой переменную, хранящую адрес какого-нибудь другого объекта, например другой переменной. Так, если в переменной `x` хранится адрес переменной `y`, то говорят, что переменная `x` указывает на переменную `y`. Когда указатель указывает на переменную, то значение этой переменной может быть получено или изменено по указателю. Такие операции с указателями называют *непрямой адресацией*.

Объявление указателя

Переменные-указатели должны быть объявлены как таковые. Ниже приведена общая форма объявления переменной-указателя:

```
тип* имя_переменной;
```

где *тип* обозначает *соотносимый тип*, который не должен быть ссылочным. Это означает, что в C# нельзя объявить указатель на объект определенного класса. Соотносимый тип указателя иногда еще называют *базовым*. Обратите внимание на положение знака `*` в объявлении указателя. Он должен следовать после наименования типа. А *имя_переменной* обозначает конкретное имя указателя-переменной.

Обратимся к конкретному примеру. Для того чтобы сделать переменную `ip` указателем на значение типа `int`, необходимо объявить ее следующим образом.

```
int* ip;
```

А указатель-типа `float` объявляется так, как показано ниже.

```
float* fp;
```

Вообще говоря, если в операторе объявления после имени типа следует знак `*`, то это означает, что создается переменная типа указателя.

Тип данных, на которые будет указывать сам указатель, зависит от его соотносимого типа. Поэтому в приведенных выше примерах переменная `ip` может служить для указания на значение типа `int`, а переменная `fp` — для указания на значение типа `float`. Следует, однако, иметь в виду, что указателю ничто не мешает указывать на что угодно. Именно поэтому указатели потенциально небезопасны.

Если у вас есть опыт программирования на C/C++, то вы должны ясно понимать главное отличие в объявлении указателей в C# и C/C++. При объявлении указателя в C/C++ знак `*` *не* разделяет список переменных в объявлении. Поэтому в следующей строке кода:

```
int* p, q;
```

объявляется указатель `p` типа `int` и переменная `q` типа `int`. Это равнозначно двум следующим объявлениям.

```
int* p;
int q;
```

А в C# знак `*` *является* разделительным, и поэтому в объявлении

```
int* p, q;
```

создаются две переменные-указателя. Это равнозначно двум следующим объявлениям.

```
int* p;
int* q;
```

Это главное отличие следует иметь в виду при переносе кода C/C++ на C#.

Операторы * и & в указателях

В указателях применяются два оператора: `*` и `&`. Оператор `&` является унарным и возвращает адрес памяти своего операнда. (Напомним, что для унарного оператора требуется единственный операнд.) Например, в следующем фрагменте кода:

```
int* ip;
int num = 10;
ip = &num;
```

в переменной `ip` сохраняется адрес памяти переменной `num`. Это адрес расположения переменной `num` в оперативной памяти компьютера. Он не имеет *никакого* отношения к *значению* переменной `num`. Поэтому в переменной `ip` содержится не значение 10, являющееся исходным для переменной `num`, а конкретный адрес, по которому эта переменная хранится в оперативной памяти. Операцию `&` можно рассматривать как возврат адреса той переменной, перед которой она указывается. Таким образом, приведенное выше присваивание словами можно описать так: "Переменная `ip` получает адрес переменной `num`."

Второй оператор, *, является дополнением оператора &. Этот унарный оператор находит значение переменной, расположенной по адресу, на который указывает его операнд. Следовательно, этот оператор обращается к значению переменной, на которую указывает соответствующий указатель. Так, если переменная ip содержит адрес памяти переменной num, как показано в предыдущем примере, то в следующей строке кода:

```
int val = *ip;
```

в переменной val сохраняется значение 10 переменной num, на которую указывает переменная ip. Операцию * можно рассматривать как получение значения по адресу. Поэтому приведенный выше оператор присваивания описывается словами следующим образом: "Переменная val получает значение по адресу, хранящемуся в переменной ip."

Оператор * можно использовать также в левой части оператора присваивания. В этом случае он задает значение, на которое указывает соответствующий указатель, как в приведенном ниже примере.

```
*ip = 100;
```

В данном примере значение 100 присваивается переменной, на которую указывает переменная ip, т.е. переменной num. Поэтому приведенный выше оператор присваивания описывается словами следующим образом: "Разместить значение 100 по адресу, хранящемуся в переменной ip."

Применение ключевого слова unsafe

Любой код, в котором используются указатели, должен быть помечен как небезопасный с помощью специального ключевого слова unsafe. Подобным образом можно пометить конкретные типы данных (например, классы и структуры), члены класса (в том числе методы и операторы) или отдельные кодовые блоки как небезопасные. В качестве примера ниже приведена программа, где указатели используются в методе Main(), помеченном как небезопасный.

```
// Продемонстрировать применение указателей и ключевого слова unsafe.

using System;

class UnsafeCode {
  // Пометить метод Main() как небезопасный.
  unsafe static void Main() {
    int count = 99;
    int* p; // создать указатель типа int

    p = &count; // поместить адрес переменной count в переменной p

    Console.WriteLine("Исходное значение переменной count: " + *p);

    *p = 10; // присвоить значение 10 переменной count,
             // на которую указывает переменная p

    Console.WriteLine("Новое значение переменной count: " + *p);
  }
}
```

Эта программа дает следующий результат.

```
Исходное значение переменной count: 99
Новое значение переменной count: 10
```

Применение модификатора fixed

В работе с указателями нередко используется модификатор fixed, который препятствует удалению управляемой переменной средствами "сборки мусора". Потребность в этом возникает, например, в том случае, если указатель обращается к полю в объекте определенного класса. А поскольку указателю ничего не известно о действиях системы "сборки мусора", то он будет указывать не на тот объект, если удалить нужный объект. Ниже приведена общая форма модификатора fixed:

```
fixed (тип* p = &фиксированный_объект) {
  // использовать фиксированный объект
}
```

где *p* обозначает указатель, которому присваивается адрес объекта. Этот объект будет оставаться на своем текущем месте в памяти до конца выполнения кодового блока. В качестве адресата оператора fixed может быть также указано единственное выражение, а не целый кодовый блок. Модификатор fixed допускается использовать только в коде, помеченном как небезопасный. Кроме того, несколько указателей с модификатором fixed могут быть объявлены списком через запятую.

Ниже приведен пример применения модификатора fixed.

```
// Продемонстрировать применение оператора fixed.

using System;

class Test {
  public int num;
  public Test(int i) { num = i; }
}

class FixedCode {
  // Пометить метод Main() как небезопасный.
  unsafe static void Main() {
    Test o = new Test(19);

    fixed (int* p = &o.num) { // использовать модификатор fixed для размещения
                              // адреса переменной экземпляр o.num в переменной p

      Console.WriteLine("Исходное значение переменной o.num: " + *p);

      *p = 10; // присвоить значение 10 переменной count,
               // на которую указывает переменная p

      Console.WriteLine("Новое значение переменной o.num: " + *p);
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Исходное значение переменной o.num: 19
Новое значение переменной o.num: 10
```

В данном примере модификатор `fixed` препятствует удалению объекта `o`. А поскольку переменная `p` указывает на переменную экземпляра `o.num`, то она будет указывать на недостоверную область памяти, если удалить объект `o`.

Доступ к членам структуры с помощью указателя

Указатель может указывать на объект типа структуры при условии, что структура не содержит ссылочные типы данных. Для доступа к члену структуры с помощью указателя следует использовать оператор-стрелку (`->`), а не оператор-точку (`.`). Например, доступ к членам структуры

```
struct MyStruct {
  public int a;
  public int b;
  public int Sum() { return a + b; }
}
```

осуществляется следующим образом.

```
MyStruct o = new MyStruct();
MyStruct* p; // объявить указатель

p = &o;
p->a = 10; // использовать оператор ->
p->b = 20; // использовать оператор ->

Console.WriteLine("Сумма равна " + p->Sum());
```

Арифметические операции над указателями

Над указателями можно выполнять только четыре арифметические операции: `++`, `--`, `+` и `-`. Для того чтобы стало понятнее, что именно происходит в арифметических операциях над указателями, рассмотрим сначала простой пример. Допустим, что переменная `p1` является указателем с текущим значением 2000, т.е. она содержит адрес 2000. После выполнения выражения

```
p1++;
```

переменная `p1` будет содержать значение 2004, а не 2001! Дело в том, что после каждого инкрементирования переменная `p1` указывает на следующее значение типа `int`. А поскольку тип `int` представлен в C# 4 байтами, то в результате инкрементирования значение переменной `p1` увеличивается на 4. Справедливо и обратное: при каждом декрементировании переменной `p1` ее значение уменьшается на 4. Например выражение

```
p1--;
```

приводит к тому, что значение переменной `p1` становится равным 1996, если раньше оно было равно 2000!

Все сказанное выше можно обобщить: после каждого инкрементирования указатель будет указывать на область памяти, где хранится следующий элемент его соотносимого типа, а после каждого декрементирования указатель будет указывать на область памяти, где хранится предыдущий элемент его соотносимого типа.

Арифметические операции над указателями не ограничиваются только инкрементированием и декрементированием. К указателям можно добавлять и вычитать из них целые значения. Так, после вычисления следующего выражения:

```
p1 = p1 + 9;
```

переменная `p1` будет указывать на девятый элемент ее соотносимого типа по отношению к элементу, на который она указывает в настоящий момент.

Если складывать указатели нельзя, то разрешается вычитать один указатель из другого, при условии, что оба указателя имеют один и тот же соотносимый тип. Результатом такой операции окажется количество элементов соотносимого типа, которые разделяют оба указателя.

Кроме сложения и вычитания целого числа из указателя, а также вычитания двух указателей, другие арифметические операции над указателями не разрешаются. В частности, к указателям нельзя добавлять или вычитать из них значения типа `float` или `double`. Не допускаются также арифметические операции над указателями типа `void*`.

Для того чтобы проверить на практике результаты арифметических операций над указателями, выполните приведенную ниже короткую программу, где выводятся физические адреса, на которые указывает целочисленный указатель (`ip`) и указатель с плавающей точкой одинарной точности (`fp`). Понаблюдайте за изменениями каждого из этих указателей по отношению к их соотносимым типам на каждом шаге цикла.

```
// Продемонстрировать результаты арифметических операций над указателями.

using System;

class PtrArithDemo {
  unsafe static void Main() {
    int x;
    int i;
    double d;

    int* ip = &i;
    double* fp = &d;

    Console.WriteLine("int     double\n");
     for(x=0; x < 10; x++) {
       Console.WriteLine((uint) (ip) + " " + (uint) (fp));
       ip++;
       fp++;
    }
  }
}
```

Ниже приведен примерный результат выполнения данной программы. У вас он может оказаться иным, хотя промежутки между выводимыми значения должны быть такими же самыми.

```
int     double

1243464 1243468
1243468 1243476
1243472 1243484
1243476 1243492
1243480 1243500
```

```
1243484 1243508
1243488 1243516
1243492 1243524
1243496 1243532
1243500 1243540
```

Как следует из приведенного выше результата, арифметические операции выполняются над указателями относительно их соотносимого типа. Так, значения типа `int` занимают в памяти 4 байта, а значения типа `double` — 8 байтов, и поэтому их адреса изменяются с приращением именно на эти величины.

Сравнение указателей

Указатели можно сравнивать с помощью таких операторов отношения, как ==, < и >. Но для того чтобы результат сравнения указателей оказался содержательным, оба указателя должны быть каким-то образом связаны друг с другом. Так, если переменные `p1` и `p2` являются указателями на две разные и не связанные вместе переменные, то любое их сравнение, как правило, не имеет никакого смысла. Но если переменные `p1` и `p2` указывают на связанные вместе переменные, например на элементы одного массива, то их сравнение может иметь определенный смысл.

Указатели и массивы

В C# указатели и массивы связаны друг с другом. Например, при указании имени массива без индекса в операторе с модификатором `fixed` формируется указатель на начало массива. В качестве примера рассмотрим следующую программу.

```
/* Указание имени массива без индекса приводит к
   формированию указателя на начало массива. */

using System;

class PtrArray {
  unsafe static void Main() {
    int[] nums = new int[10];

    fixed(int* p = &nums[0], p2 = nums) {
      if(p == p2)
        Console.WriteLine("Указатели p и p2 содержат " +
                          "один и тот же адрес.");
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Указатели p и p2 содержат один и тот же адрес
```

Как следует из приведенного выше результата, выражения

```
&nums[0]
```

и

```
nums
```

оказываются одинаковыми. Но поскольку вторая форма более лаконична, то она чаще используется в программировании, когда требуется указатель на начало массива.

Индексирование указателей

Когда указатель обращается к массиву, его можно индексировать как сам массив. Такой синтаксис служит более удобной в некоторых случаях альтернативой арифметическим операциям над указателями. Рассмотрим следующий пример программы.

```
// Проиндексировать указатель как массив.

using System;

class PtrIndexDemo {
  unsafe static void Main() {
    int[] nums = new int[10];

    // Проиндексировать указатель.
    Console.WriteLine("Индексирование указателя как массива.");
    fixed (int* p = nums) {
      for(int i=0; i < 10; i++)
        p[i] = i; // индексировать указатель как массив

      for(int i=0; i < 10; i++)
        Console.WriteLine("p[{0}]: {1} ", i, p[i]);
    }

    // Использовать арифметические операции над указателями.
    Console.WriteLine("\nПрименение арифметических " +
                      "операций над указателями.");
    fixed (int* p = nums) {
      for(int i=0; i < 10; i++)
        *(p+i) = i; // использовать арифметическую операцию над указателем

      for(int i=0; i < 10; i++)
        Console.WriteLine("*(p+{0}): {1} ", i, *(p+i));
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Индексирование указателя как массива.
p[0]: 0
p[1]: 1
p[2]: 2
p[3]: 3
p[4]: 4
p[5]: 5
p[6]: 6
p[7]: 7
p[8]: 8
p[9]: 9

Применение арифметических операций над указателями.
*(p+0): 0
*(p+1): 1
*(p+2): 2
*(p+3): 3
*(p+4): 4
```

```
*(p+5): 5
*(p+6): 6
*(p+7): 7
*(p+8): 8
*(p+9): 9
```

Как следует из результата выполнения приведенной выше программы, общая форма выражения с указателем

```
*(ptr + i)
```

может быть заменена следующим синтаксисом индексирования массива.

```
ptr[i]
```

Что касается индексирования указателей, то необходимо иметь в виду следующее. Во-первых, при таком индексировании контроль границ массива не осуществляется. Поэтому указатель может обращаться к элементу вне границ массива. И во-вторых, для указателя не предусмотрено свойство `Length`, определяющее длину массива. Поэтому, если используется указатель, длина массива заранее неизвестна.

Указатели и строки

Символьные строки реализованы в C# в виде объектов. Тем не менее отдельные символы в строке могут быть доступны по указателю. Для этого указателю типа `char*` присваивается адрес начала символьной строки в следующем операторе с модификатором `fixed`.

```
fixed(char* p = str) { // ...
```

После выполнения оператора с модификатором `fixed` переменная *p* будет указывать на начало массива символов, составляющих строку. Этот массив оканчивается символом конца строки, т.е. нулевым символом. Поэтому данное обстоятельство можно использовать для проверки конца массива. В C/C++ строки реализуются в виде массивов, оканчивающихся символом конца строки, а следовательно, получив указатель типа `char*` на строку, ею можно манипулировать таким же образом, как и в C/C++.

Ниже приведена программа, демонстрирующая доступ к символьной строке по указателю типа `char*`.

```
// Использовать модификатор fixed для получения
// указателя на начало строки.

using System;

class FixedString {
  unsafe static void Main() {
    string str = "это тест";

    // Получить указатель p на начало строки str.
    fixed(char* p = str) {

      // Вывести содержимое строки str по указателю p.
      for(int i=0; p[i] != 0; i++)
        Console.Write(p[i]);
    }
```

```
    Console.WriteLine();

  }
}
```

Эта программа дает следующий результат.

```
это тест
```

Многоуровневая непрямая адресация

Один указатель может указывать на другой, а тот, свою очередь, — на целевое значение. Это так называемая *многоуровневая непрямая адресация*, или применение *указателей на указатели*. Такое применение указателей может показаться, на первый взгляд, запутанным. Для прояснения принципа многоуровневой непрямой адресации обратимся за помощью к рис. 20.1. Как видите, значением обычного указателя является адрес переменной, содержащей требуемое значение. Если же применяется указатель на указатель, то первый из них содержит адрес второго, указывающего на переменную, содержащую требуемое значение.



Рис. 20.1. Одно- и многоуровневая непрямая адресация

Многоуровневая непрямая адресация может быть продолжена до любого предела, но потребность более чем в двух уровнях адресации по указателям возникает крайне редко. На самом деле чрезмерная непрямая адресация очень трудно прослеживается и чревата ошибками.

Переменная, являющаяся указателем на указатель, должна быть объявлена как таковая. Для этого достаточно указать дополнительный знак * после имени типа переменной. Например, в следующем объявлении компилятор уведомляется о том, что переменная `q` является указателем на указатель и относится к типу `int`.

```
int** q;
```

Следует, однако, иметь в виду, что переменная `q` является указателем не на целое значение, а на указатель типа `int`.

Для доступа к целевому значению, косвенно адресуемому по указателю на указатель, следует дважды применить оператор *, как в приведенном ниже примере.

```
using System;

class MultipleIndirect {
```

```
  unsafe static void Main() {
    int x; // содержит значение типа int
    int* p; // содержит указатель типа int
    int** q; // содержит указатель на указатель типа int

    x = 10;
    p = &x; // поместить адрес переменной x в переменной p
    q = &p; // поместить адрес переменной p в переменной q

    Console.WriteLine(**q); // вывести значение переменной x
  }
}
```

Результатом выполнения этой программы будет выведенное на экран значение `10` переменной `x`. В данной программе переменная `p` объявляется как указатель на значение типа `int`, а переменная `q` — как указатель на указатель типа `int`.

И последнее замечание: не путайте многоуровневую непрямую адресацию со структурами данных высокого уровня, в том числе связными списками, так как это совершенно разные понятия.

Массивы указателей

Указатели могут быть организованы в массивы, как и любой другой тип данных. Ниже приведен пример объявления массива указателей типа `int` длиной в три элемента.

```
int * [] ptrs = new int * [3];
```

Для того чтобы присвоить адрес переменной `var` типа `int` третьему элементу массива указателей, достаточно написать следующую строку кода.

```
ptrs[2] = &var;
```

А для того чтобы обнаружить значение переменной `var`, достаточно написать приведенную ниже строку кода.

```
*ptrs[2]
```

Оператор `sizeof`

Во время работы с небезопасным кодом иногда полезно знать размер в байтах одного из встроенных в C# типов значений. Для получения этой информации служит оператор `sizeof`. Ниже приведена его общая форма:

```
sizeof(тип)
```

где *тип* обозначает тот тип, размер которого требуется получить. Вообще говоря, оператор `sizeof` предназначен главным образом для особых случаев и, в частности, для работы со смешанным кодом: управляемым и неуправляемым.

Оператор `stackalloc`

Для распределения памяти, выделенной под стек, служит оператор `stackalloc`. Им можно пользоваться лишь при инициализации локальных переменных. Ниже приведена общая форма этого оператора:

```
тип *p = stackalloc тип[размер]
```

где *p* обозначает указатель, получающий адрес области памяти, достаточной для хранения объектов, имеющих указанный *тип*, в количестве, которое обозначает *размер*. Если же в стеке недостаточно места для распределения памяти, то генерируется исключение `System.StackOverflowException`. И наконец, оператор `stackalloc` можно использовать только в небезопасном коде.

Как правило, память для объектов выделяется из кучи — динамически распределяемой свободной области памяти. А выделение памяти из стека является исключением. Ведь переменные, располагаемые в стеке, не удаляются средствами "сборки мусора", а существуют только в течение времени выполнения метода, в котором они объявляются. После возврата из метода выделенная память освобождается. Преимущество применения оператора `stackalloc` заключается, в частности, в том, что в этом случае не нужно беспокоиться об очистке памяти средствами "сборки мусора".

Ниже приведен пример применения оператора `stackalloc`.

```
// Продемонстрировать применение оператора stackalloc.

using System;

class UseStackAlloc {
  unsafe static void Main() {
    int* ptrs = stackalloc int[3];

    ptrs[0] = 1;
    ptrs[1] = 2;
    ptrs[2] = 3;

    for(int i=0; i < 3; i++)
      Console.WriteLine(ptrs[i]);
  }
}
```

Вот к какому результату приводит выполнение кода из данного примера.

```
1
2
3
```

Создание буферов фиксированного размера

Ключевое слово `fixed` находит еще одно применение при создании одномерных массивов фиксированного размера. В документации на C# такие массивы называются *буферами фиксированного размера*. Такие буферы всегда являются членами структуры. Они предназначены для создания структуры, в которой содержатся элементы массива, образующие буфер. Когда элемент массива включается в состав структуры, в ней, как правило, хранится лишь ссылка на этот массив. Используя буфер фиксированного размера, в структуре можно разместить весь массив. В итоге получается структура, пригодная в тех случаях, когда важен ее размер, как, например, в многоязыковом программировании, при согласовании данных, созданных вне программы на C#, или же когда требуется неуправляемая структура, содержащая массив. Но буферы фиксированного размера можно использовать только в небезопасном коде.

Для создания буфера фиксированного размера служит следующая общая форма:

```
fixed тип имя_буфера[размер];
```

где *тип* обозначает тип данных массива; *имя_буфера* — конкретное имя буфера фиксированного размера; *размер* — количество элементов, образующих буфер. Буферы фиксированного размера могут указываться только в структуре.

Для того чтобы стала очевиднее польза от буферов фиксированного размера, рассмотрим ситуацию, в которой программе ведения счетов, написанной на C++, требуется передать информацию о банковском счете. Допустим также, что учетная запись каждого счета организована так, как показано ниже.

`Name`	Строка длиной 80 байтов, состоящая из 8-разрядных символов в коде ASCII
`Balance`	Числовое значение типа `double` длиной 8 байтов
`ID`	Числовое значение типа `long` длиной 8 байтов

В программе на C++ каждая структура содержит массив `Name`, тогда как в программе на C# в такой структуре хранится лишь ссылка на массив. Поэтому для правильного представления данных из этой структуры в C# требуется буфер фиксированного размера, как показано ниже.

```
// Использовать буфер фиксированного размера.
unsafe struct FixedBankRecord {
  public fixed byte Name[80]; // создать буфер фиксированного размера
  public double Balance;
  public long ID;
}
```

Когда буфер фиксированного размера используется вместо массива `Name`, каждый экземпляр структуры `FixedBankRecord` будет содержать все 80 байтов массива `Name`. Именно таким образом структура и организована в программе на C++. Следовательно, общий размер структуры `FixedBankRecord` окажется равным 96, т.е. сумме ее членов. Ниже приведена программа, демонстрирующая этот факт.

```
// Продемонстрировать применение буфера фиксированного размера.

using System;

// Создать буфер фиксированного размера.
unsafe struct FixedBankRecord {
  public fixed byte Name[80];
  public double Balance;
  public long ID;
}

class FixedSizeBuffer {
   // Пометить метод Main() как небезопасный.
   unsafe static void Main() {
     Console.WriteLine("Размер структуры FixedBankRecord: " +
                       sizeof(FixedBankRecord));
  }
}
```

Эта программа дает следующий результат.

```
Размер структуры FixedBankRecord: 96
```

Размер структуры `FixedBankRecord` оказывается в точности равным сумме ее членов, но так бывает далеко не всегда со структурами, содержащими буферы фиксированного размера. Ради повышения эффективности кода общая длина структуры может быть увеличена для выравнивания по четной границе, например по границе слова. Поэтому общая длина структуры может оказаться на несколько байтов больше, чем сумма ее членов, даже если в ней содержатся буферы фиксированного размера. Как правило, аналогичное выравнивание длины структуры происходит и в C++. Следует, однако, иметь в виду возможные отличия в этом отношении.

И наконец, обратите внимание на то, как в данной программе создается буфер фиксированного размера вместо массива `Name`.

```
public fixed byte Name[80]; // создать буфер фиксированного размера
```

Как видите, размер массива указывается после его имени. Такое обозначение обычно принято в C++ и отличается в объявлениях массивов в C#. В данном операторе распределяется по 80 байтов памяти в пределах каждого объекта типа `FixedBankRecord`.

Обнуляемые типы

Начиная с версии 2.0, в C# внедрено средство, обеспечивающее изящное решение типичной и не очень приятной задачи распознавания и обработки полей, не содержащих значения, т.е. неинициализированных полей. Это средство называется *обнуляемым типом*. Для того чтобы стала более понятной суть данной задачи, рассмотрим пример простой базы данных заказчиков, в которой хранится запись с именем, адресом, идентификационным номером заказчика, номером счета-фактуры и текущим остатком на счету. В подобной ситуации может быть вполне создан элемент данных заказчика, вкотором одно или несколько полей не инициализированы. Например, заказчик может просто запросить каталог продукции, и в этом случае номер счета-фактуры не потребуется, а значит, его поле окажется неиспользованным.

Раньше для обработки неиспользуемых полей приходилось применять заполняющие значения или дополнительные поля, которые просто указывали, используется поле или нет. Безусловно, заполняющие значения пригодны лишь в том случае, если они подставляются вместо значения, которое в противном случае окажется недействительным, но так бывает далеко не всегда. А дополнительные поля, указывающие, используется поле или нет, пригодны во всех случаях, но их ввод и обработка вручную доставляют немало хлопот. Оба эти затруднения позволяет преодолеть обнуляемый тип.

Основы применения обнуляемых типов

Обнуляемый тип — это особый вариант типа значения, представленный структурой. Помимо значений, определяемых базовым типом, обнуляемый тип позволяет хранить пустые значения (`null`). Следовательно, обнуляемый тип имеет такой же диапазон представления чисел и характеристики, как и его базовый тип. Он предоставляет дополнительную возможность обозначить значение, указывающее на то, что переменная данного типа не инициализирована. Обнуляемые типы являются объектами типа `System.Nullable<T>`, где `T` — тип значения, которое не должно быть обнуляемым.

ПРИМЕЧАНИЕ

Обнуляемые эквиваленты могут быть только у типов значений.

Обнуляемый тип может быть указан двумя способами. Во-первых, объекты типа `Nullable<T>`, определенного в пространстве имен `System`, могут быть объявлены явным образом. Так, в приведенном ниже примере создаются обнуляемые переменные типа `int` и `bool`.

```
System.Nullable<int> count;
System.Nullable<bool> done;
```

И во-вторых, обнуляемый тип объявляется более кратким и поэтому чаще используемым способом с указанием знака ? после имени базового типа. В приведенном ниже примере демонстрируется более распространенный способ объявления обнуляемых переменных типа `int` и `bool`.

```
int? count;
bool? done;
```

Когда в коде применяются обнуляемые типы, создаваемый обнуляемый объект обычно выглядит следующим образом.

```
int? count = null;
```

В данной строке кода переменная `count` явно инициализируется пустым значением (`null`). Это вполне соответствует принятому правилу: прежде чем использовать переменную, ей нужно присвоить значение. В данном случае присваиваемое значение означает, что переменная не определена.

Значение может быть присвоено обнуляемой переменной обычным образом, поскольку преобразование базового типа в обнуляемый определено заранее. Например, в следующей строке кода переменной `count` присваивается значение 100.

```
count = 100;
```

Определить, имеет переменная обнуляемого типа пустое или конкретное значение, можно двумя способами. Во-первых, можно проверить переменную на пустое значение. Так, если переменная `count` объявлена так, как показано выше, то в следующей строке определяется, имеет ли эта переменная конкретное значение.

```
if(count != null) // переменная имеет значение
```

Если переменная `count` не является пустой, то она содержит конкретное значение.

И во-вторых, можно воспользоваться доступным только для чтения свойством `HasValue` типа `Nullable<T>`, чтобы определить, содержит ли переменная обнуляемого типа конкретное значение. Это свойство показано ниже.

```
bool HasValue
```

Свойство `HasValue` возвращает логическое значение `true`, если экземпляр объекта, для которого оно вызывается, содержит конкретное значение, а иначе оно возвращает логическое значение `false`. Ниже приведен пример, в котором конкретное значение обнуляемого объекта `count` определяется вторым способом с помощью свойства `HasValue`.

```
if(count.HasValue) // переменная имеет значение
```

Если обнуляемый объект содержит конкретное значение, то получить это значение можно с помощью доступного только для чтения свойства `Value` типа `Nullable<T>`.

```
T Value
```

Свойство `Value` возвращает экземпляр обнуляемого объекта, для которого оно вызывается. Если же попытаться получить с помощью этого свойства значение пустой переменной, то в итоге будет сгенерировано исключение `System.InvalidOperationException`. Кроме того, значение экземпляра обнуляемого объекта можно получить путем приведения к его базовому типу.

В следующей программе демонстрируется основной механизм обращения с обнуляемым типом.

```
// Продемонстрировать применение обнуляемого типа.

using System;

class NullableDemo {
  static void Main() {
    int? count = null;

    if(count.HasValue)
      Console.WriteLine("Переменная count имеет следующее значение: " +
                        count.Value);
    else
      Console.WriteLine("У переменной count отсутствует значение");

    count = 100;
    if(count.HasValue)
      Console.WriteLine("Переменная count имеет следующее значение: " +
                        count.Value);
     else
       Console.WriteLine("У переменной count отсутствует значение");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
У переменной count отсутствует значение
Переменная count имеет следующее значение: 100
```

Применение обнуляемых объектов в выражениях

Обнуляемый объект может использоваться в тех выражениях, которые являются действительными для его базового типа. Более того, обнуляемые объекты могут сочетаться с необнуляемыми объектами в одном выражении. И это вполне допустимо благодаря предопределенному преобразованию базового типа в обнуляемый. Когда обнуляемые и необнуляемые типы сочетаются в одной операции, ее результатом становится значение обнуляемого типа.

В приведенной ниже программе демонстрируется применение обнуляемых типов в выражениях.

```
// Использовать обнуляемые объекты в выражениях.

using System;

class NullableDemo {
  static void Main() {
```

```
    int? count = null;
    int? result = null;

    int incr = 10; // переменная incr не является обнуляемой

    // переменная result содержит пустое значение,
    // переменная оказывается count пустой.
    result = count + incr;
    if(result.HasValue)
      Console.WriteLine("Переменная result имеет следующее значение: " +
                        result.Value);
    else
      Console.WriteLine("У переменной result отсутствует значение");

    // Теперь переменная count получает свое значение, и поэтому
    // переменная result будет содержать конкретное значение.
    count = 100;
    result = count + incr;
    if(result.HasValue)
      Console.WriteLine("Переменная result имеет следующее значение: " +
                        result.Value);
    else
      Console.WriteLine("У переменной result отсутствует значение");
  }
}
```

При выполнении этой программы получается следующий результат.

```
У переменной result отсутствует значение
Переменная result имеет следующее значение: 110
```

Оператор ??

Попытка преобразовать обнуляемый объект в его базовый тип путем приведения типов обычно приводит к генерированию исключения `System.InvalidOperationException`, если обнуляемый объект содержит пустое значение. Это может произойти, например, в том случае, если значение обнуляемого объекта присваивается переменной его базового типа с помощью приведения типов. Появления данного исключения можно избежать, если воспользоваться оператором `??`, называемым *нулеобъединяющим оператором*. Этот оператор позволяет указать значение, которое будет использоваться по умолчанию, если обнуляемый объект содержит пустое значение. Он также исключает потребность в приведении типов.

Ниже приведена общая форма оператора `??`.

обнуляемый_объект `??` *значение_по_умолчанию*

Если *обнуляемый_объект* содержит конкретное значение, то результатом операции `??` будет именно это значение. В противном случае результатом операции `??` окажется *значение_по_умолчанию*.

Например, в приведенном ниже фрагменте кода переменная `balance` содержит пустое значение. Вследствие этого переменной `currentBalance` присваивается значение `0.0`, используемое по умолчанию, и тем самым устраняется причина для генерирования исключения.

```
double? balance = null;
double currentBalance;

currentBalance = balance ?? 0.0;
```

В следующем фрагменте кода переменной `balance` присваивается значение `123.75`.

```
double? balance = 123.75;
double currentBalance;

currentBalance = balance ?? 0.0;
```

Теперь переменная `currentBalance` содержит значение `123.75` переменной `balance`.

И еще одно замечание: выражение в правой части оператора `??` вычисляется только в том случае, если выражение в левой его части не содержит значение. Этот факт демонстрируется в приведенной ниже программе.

```
// Применение оператора ??

using System;

class NullableDemo2 {
  // Возвратить нулевой остаток.
  static double GetZeroBal() {
    Console.WriteLine("В методе GetZeroBal().");
    return 0.0;
  }

  static void Main() {
    double? balance = 123.75;
    double currentBalance;

    // Здесь метод GetZeroBal() не вызывается, поскольку
    // переменная balance содержит конкретное значение.
    currentBalance = balance ?? GetZeroBal();

    Console.WriteLine(currentBalance);
  }
}
```

В этой программе метод `GetZeroBal()` не вызывается, поскольку переменная `balance` содержит конкретное значение. Как пояснялось выше, если выражение в левой части оператора `??` содержит конкретное значение, то выражение в правой его части не вычисляется.

Обнуляемые объекты, операторы отношения и логические операторы

Обнуляемые объекты могут использоваться в выражениях отношения таким же образом, как и соответствующие объекты необнуляемого типа. Но они должны подчиняться следующему дополнительному правилу: когда два обнуляемых объекта сравниваются в операциях сравнения <, >, <= или >=, то их результат будет ложным, если любой из обнуляемых объектов оказывается пустым, т.е. содержит значение `null`. В качестве примера рассмотрим следующий фрагмент кода.

```
byte? lower = 16;
byte? upper = null;

// Здесь переменная lower определена, а переменная upper не определена.
if(lower < upper) // ложно
```

В данном случае проверка того, что значение одной переменой меньше значения другой, дает ложный результат. Хотя это и не совсем очевидно, как, впрочем, и следующая проверка противоположного характера.

```
if(lower > upper) // .. также ложно!
```

Следовательно, если один или оба сравниваемых обнуляемых объекта оказываются пустыми, то результат их сравнения всегда будет ложным. Это фактически означает, что пустое значение (`null`) не участвует в отношении порядка.

Тем не менее с помощью операторов == и != можно проверить, содержит ли обнуляемый объект пустое значение. Например, следующая проверка вполне допустима и дает истинный результат.

```
if(upper == null) // ...
```

Если в логическом выражении участвуют два объекта типа `bool?`, то его результат может иметь одно из трех следующих значений: `true` (истинное), `false` (ложное) или `null` (неопределенное). Ниже приведены результаты применения логических операторов `&` и `|` к объектам типа `bool?`.

P	Q	P | Q	P & Q
true	null	true	null
false	null	null	false
null	true	true	null
null	false	null	false
null	null	null	null

И наконец, если логический оператор `!` применяется к значению типа `bool?`, которое является пустым (`null`), то результат этой операции будет неопределенным (`null`).

Частичные типы

Начиная с версии 2.0, в C# появилась возможность разделять определение класса, структуры или интерфейса на две или более части с сохранением каждой из них в отдельном файле. Это делается с помощью контекстного ключевого слова `partial`. Все эти части объединяются вместе во время компиляции программы.

Если модификатор `partial` используется для создания частичного типа, то он принимает следующую общую форму:

```
partial тип имя_типа { // ...
```

где *имя_типа* обозначает имя класса, структуры или интерфейса, разделяемого на части. Каждая часть получающегося частичного типа должна указываться вместе с модификатором `partial`.

Рассмотрим пример разделения простого класса, содержащего координаты XY, на три отдельных файла. Ниже приведено содержимое первого файла.

```
partial class XY {
  public XY(int a, int b) {
    X = a;
    Y = b;
  }
}
```

Далее следует содержимое второго файла.

```
partial class XY {
  public int X { get; set; }
}
```

И наконец, содержимое третьего файла.

```
partial class XY {
  public int Y { get; set; }
}
```

В приведенном ниже файле исходного текста программы демонстрируется применение класса XY.

```
// Продемонстрировать определения частичного класса.

using System;

class Test {
  static void Main() {
    XY xy = new XY(1, 2);

    Console.WriteLine(xy.X + "," + xy.Y);
  }
}
```

Для того чтобы воспользоваться классом XY, необходимо включить в компиляцию все его файлы. Так, если файлы класса XY называются xy1.cs, xy2.cs и xy3.cs, а класс Test содержится в файле test.cs, то для его компиляции достаточно ввести в командной строке следующее.

```
csc test.cs xy1.cs xy2.cs xy3.cs
```

И последнее замечание: в C# допускаются частичные обобщенные классы. Но параметры типа в объявлении каждого такого класса должны совпадать с теми, что указываются в остальных его частях.

Частичные методы

Как пояснялось в предыдущем разделе, с помощью модификатора partial можно создать класс частичного типа. Начиная с версии 3.0, в C# появилась возможность использовать этот модификатор и для создания *частичного метода* в элементе данных частичного типа. Частичный метод объявляется в одной его части, а реализуется в другой. Следовательно, с помощью модификатора partial можно отделить объявление метода от его реализации в частичном классе или структуре.

Главная особенность частичного метода заключается в том, что его реализация не требуется! Если частичный метод не реализуется в другой части класса или структуры, то все его вызовы молча игнорируются. Это дает возможность определить, но не востребовать дополнительные, хотя и не обязательные функции класса. Если эти функции не реализованы, то они просто игнорируются.

Ниже приведена расширенная версия предыдущей программы, в которой создается частичный метод `Show()`. Этот метод вызывается другим методом, `ShowXY()`. Ради удобства все части класса `XY` представлены в одном файле, но они могут быть распределены по отдельным файлам, как было показано в предыдущем разделе.

```
// Продемонстрировать применение частичного метода.

using System;

partial class XY {
  public XY(int a, int b) {
    X = a;
    Y = b;
  }

  // Объявить частичный метод.
  partial void Show();
}

partial class XY {
  public int X { get; set; }

  // Реализовать частичный метод.
  partial void Show() {
    Console.WriteLine("{0}, {1}", X, Y);
  }
}

partial class XY {
  public int Y { get; set; }

  // Вызвать частичный метод.
  public void ShowXY() {
    Show();
  }
}

class Test {
  static void Main() {
    XY xy = new XY(1, 2);
    xy.ShowXY();
  }
}
```

Обратите внимание на то, что метод `Show()` объявляется в одной части класса `XY`, а реализуется в другой его части. В реализации этого метода выводятся значения координат X и Y. Это означает, что когда метод `Show()` вызывается из метода `ShowXY()`, то данный вызов действительно имеет конкретные последствия: вывод значений

координат X и Y. Но если закомментировать реализацию метода `Show()`, то его вызов из метода `ShowXY()` ни к чему не приведет.

Частичным методам присущ ряд следующих ограничений. Они должны возвращать значение типа `void`. У них не может быть модификаторов доступа и они не могут быть виртуальными. В них нельзя также использовать параметры `out`.

Создание объектов динамического типа

Как уже упоминалось не раз, начиная с главы 3, C# является строго типизированным языком программирования. Вообще говоря, это означает, что все операции проверяются во время компиляции на соответствие типов, и поэтому действия, не поддерживаемые конкретным типом, не подлежат компиляции. И хотя строгий контроль типов дает немало преимуществ программирующему, помогая создавать устойчивые и надежные программы, он может вызвать определенные осложнения в тех случаях, когда тип объекта остается неизвестным вплоть до времени выполнения. Нечто подобное может произойти при использовании рефлексии, доступе к COM-объекту или же в том случае, если требуется возможность взаимодействия с таким динамическим языком, как, например, IronPython. До появления версии C# 4.0 подобные ситуации были трудноразрешимы. Поэтому для выхода из столь затруднительного положения в версии C# 4.0 был внедрен новый тип данных под названием `dynamic`.

За одним важным исключением, тип `dynamic` очень похож на тип `object`, поскольку его можно использовать для ссылки на объект любого типа. А отличается он от типа `object` тем, что вся проверка объектов типа `dynamic` на соответствие типов откладывается до времени выполнения, тогда как объекты типа `object` подлежат этой проверке во время компиляции. Преимущество откладывания подобной проверки до времени выполнения состоит в том, что во время компиляции предполагается, что объект типа `dynamic` поддерживает любые операции, включая применение операторов, вызовы методов, доступ к полям и т.д. Это дает возможность скомпилировать код без ошибок. Конечно, если во время выполнения фактический тип, присваиваемый объекту, не поддерживает ту или иную операцию, то возникнет исключительная ситуация во время выполнения.

В приведенном ниже примере программы применение типа `dynamic` демонстрируется на практике.

```
// Продемонстрировать применение типа dynamic.

using System;
using System.Globalization;

class DynDemo {
  static void Main() {
    // Объявить две динамические переменные.
    dynamic str;
    dynamic val;

    // Поддерживается неявное преобразование в динамические типы.
    // Поэтому следующие присваивания вполне допустимы.
    str = "Это строка";
    val = 10;
```

```
    Console.WriteLine("Переменная str содержит: " + str);
    Console.WriteLine("Переменная val содержит: " + val + '\n');

    str = str.ToUpper(CultureInfo.CurrentCulture);
    Console.WriteLine("Переменная str теперь содержит: " + str);

    val = val + 2;
    Console.WriteLine("Переменная val теперь содержит: " + val + '\n');

    string str2 = str.ToLower(CultureInfo.CurrentCulture);
    Console.WriteLine("Переменная str2 содержит: " + str2);

    // Поддерживаются неявные преобразования из динамических типов.
    int x = val * 2;
    Console.WriteLine("Переменная x содержит: " + x);
  }
}
```

Выполнение этой программы дает следующий результат.

```
Переменная str содержит: Это строка
Переменная val содержит: 10

Переменная str теперь содержит: ЭТО СТРОКА
Переменная val теперь содержит: 12

Переменная str2 содержит: это строка
Переменная x содержит: 24
```

Обратите внимание в этой программе на две переменные `str` и `val`, объявляемые с помощью типа `dynamic`. Это означает, что проверка на соответствие типов операций с участием обеих переменных не будет произведена во время компиляции. В итоге для них оказывается пригодной любая операция. В данном случае для переменной `str` вызываются методы `ToUpper()` и `ToLower()` класса `String`, а переменная участвует в операциях сложения и умножения. И хотя все перечисленные выше действия совместимы с типами объектов, присваиваемых обеим переменным в рассматриваемом здесь примере, компилятору об этом ничего не известно — он просто принимает. И это, конечно, упрощает программирование динамических процедур, хотя и допускает возможность появления ошибок в подобных действиях во время выполнения.

В разбираемом здесь примере программа ведет себя "правильно" во время выполнения, поскольку объекты, присваиваемые упомянутым выше переменным, поддерживают действия, выполняемые в программе. В частности, переменной `val` присваивается целое значение, и поэтому она поддерживает такие целочисленные операции, как сложение. А переменной `str` присваивается символьная строка, и поэтому она поддерживает строковые операции. Следует, однако, иметь в виду, что ответственность за фактическую поддержку типом объекта, на который делается ссылка, всех операций над данными типа `dynamic` возлагается на самого программирующего. В противном случае выполнение программы завершится аварийным сбоем.

В приведенном выше примере обращает на себя внимание еще одно обстоятельство: переменной типа `dynamic` может быть присвоен любой тип ссылки на объект благодаря неявному преобразованию любого типа в тип `dynamic`. Кроме того, тип `dynamic` автоматически преобразуется в любой другой тип. Разумеется, если во время выполнения такое преобразование окажется неправильным, то произойдет ошибка

при выполнении. Так, если добавить в конце рассматриваемой здесь программы следующую строку кода:

```
bool b = val;
```

то возникнет ошибка при выполнении из-за отсутствия неявного преобразования типа `int` (который оказывается типом переменной `val` во время выполнения) в тип `bool`. Поэтому данная строка кода приведет к ошибке при выполнении, хотя она и будет скомпилирована безошибочно.

Прежде чем оставить данный пример программы, попробуйте поэкспериментировать с ней. В частности, измените тип переменных `str` и `val` на `object`, а затем попытайтесь скомпилировать программу еще раз. В итоге появятся ошибки при компиляции, поскольку тип `object` не поддерживает действия, выполняемые над обеими переменными, что и будет обнаружено во время компиляции. В этом, собственно, и заключается основное отличие типов `object` и `dynamic`. Несмотря на то что оба типа могут использоваться для ссылки на объект любого другого типа, над переменной типа `object` можно производить только те действия, которые поддерживаются типом `object`. Если же вы используете тип `dynamic`, то можете указать какое угодно действие, при условии что это действие поддерживается конкретным объектом, на который делается ссылка во время выполнения.

Для того чтобы стало понятно, насколько тип `dynamic` способен упростить решение некоторых задач, рассмотрим простой пример его применения вместе с рефлексией. Как пояснялось в главе 17, чтобы вызвать метод для объекта класса, получаемого во время выполнения с помощью рефлексии, можно, в частности, обратиться к методу `Invoke()`. И хотя такой способ оказывается вполне работоспособным, нужный метод намного удобнее вызвать по имени в тех случаях, когда его имя известно. Например, вполне возможна такая ситуация, когда в некоторой сборке содержится конкретный класс, поддерживающий методы, имена и действия которых заранее известны. Но поскольку эта сборка подвержена изменениям, то приходится постоянно убеждаться в том, что используется последняя ее версия. Для проверки текущей версии сборки можно, например, воспользоваться рефлексией, сконструировать объект искомого класса, а затем вызвать методы, определенные в этом классе. Теперь эти методы можно вызвать по имени с помощью типа `dynamic`, а не метода `Invoke()`, поскольку их имена известны.

Разместите сначала приведенный ниже код в файле с именем `MyClass.cs`. Этот код будет динамически загружаться посредством рефлексии.

```
public class DivBy {
  public bool IsDivBy(int a, int b) {
    if((a % b) == 0) return true;
      return false;
  }

  public bool IsEven(int a) {
    if((a % 2) == 0) return true;
      return false;
  }
}
```

Затем скомпилируйте этот файл в библиотеку DLL под именем `MyClass.dll`. Если вы пользуетесь компилятором командной строки, введите в командной строке следующее.

```
csc /t:library MyClass.cs
```

Далее составьте программу, в которой применяется библиотека `MyClass.dll`, как показано ниже.

```
// Использовать тип dynamic вместе с рефлексией.

using System;
using System.Reflection;

class DynRefDemo {
  static void Main() {

    Assembly asm = Assembly.LoadFrom("MyClass.dll");

    Type[] all = asm.GetTypes();

    // Найти класс DivBy.
    int i;
    for(i = 0; i < all.Length; i++)
      if(all[i].Name == "DivBy") break;

    if(i == all.Length) {
      Console.WriteLine("Класс DivBy не найден в сборке.");
      return;
    }

    Type t = all[i];

    // А теперь найти используемый по умолчанию конструктор.
    ConstructorInfo[] ci = t.GetConstructors();

    int j;
    for(j = 0; j < ci.Length; j++)
      if(ci[j].GetParameters().Length == 0) break;

    if(j == ci.Length) {
      Console.WriteLine("Используемый по умолчанию конструктор не найден.");
      return;
    }

    // Создать объект класса DivBy динамически.
    dynamic obj = ci[j].Invoke(null);

    // Далее вызвать по имени методы для переменной obj. Это вполне допустимо,
    // поскольку переменная obj относится к типу dynamic, а вызовы методов
    // проверяются на соответствие типов во время выполнения, а не компиляции.
    if(obj.IsDivBy(15, 3))
      Console.WriteLine("15 делится нацело на 3.");
    else
      Console.WriteLine("15 НЕ делится нацело на 3.");

      if(obj.IsEven(9))
        Console.WriteLine("9 четное число.");
      else
        Console.WriteLine("9 НЕ четное число.");
  }
}
```

Как видите, в данной программе сначала динамически загружается библиотека `MyClass.dll`, а затем используется рефлексия для построения объекта класса `DivBy`. Построенный объект присваивается далее переменной `obj` типа `dynamic`. А раз так, то методы `IsDivBy()` и `IsEven()` могут быть вызваны для переменной `obj` по имени, а не с помощью метода `Invoke()`. В данном примере это вполне допустимо, поскольку переменная `obj` на самом деле ссылается на объект класса `DivBy`. В противном случае выполнение программы завершилось бы неудачно.

Приведенный выше пример сильно упрощен и несколько надуман. Тем не менее он наглядно показывает главное преимущество, которое дает тип `dynamic` в тех случаях, когда типы получаются во время выполнения. Когда характеристики искомого типа, в том числе методы, операторы, поля и свойства, заранее известны, эти характеристики могут быть получены по имени с помощью типа `dynamic`, как следует из приведенного выше примера. Благодаря этому код становится проще, короче и понятнее.

Применяя тип `dynamic`, следует также иметь в виду, что при компиляции программы тип `dynamic` фактически заменяется объектом, а для описания его применения во время выполнения предоставляется соответствующая информация. И поскольку тип `dynamic` компилируется в тип `object` для целей перегрузки, то оба типа `dynamic` и `object` расцениваются как одно и то же. Поэтому при компиляции двух следующих перегружаемых методов возникнет ошибка.

```
static void f(object v) { // ... }
static void f(dynamic v) { // ... } // Ошибка!
```

И последнее замечание: тип `dynamic` поддерживается компонентом DLR (Dynamic Language Runtime — Средство создания динамических языков во время выполнения), внедренным в .NET 4.0.

Возможность взаимодействия с моделью COM

В версии C# 4.0 внедрены средства, упрощающие возможность взаимодействия с неуправляемым кодом, определяемым моделью компонентных объектов (COM) и применяемым, в частности, в COM-объекте Office Automation. Некоторые из этих средств, в том числе тип `dynamic`, именованные и необязательные свойства, пригодны для применения помимо возможности взаимодействия с моделью COM. Тема модели COM вообще и COM-объекта Office Automation в частности весьма обширна, а порой и довольно сложна, чтобы обсуждать ее в этой книге. Поэтому возможность взаимодействия с моделью COM выходит за рамки данной книги.

Тем не менее две особенности, имеющие отношение к возможности взаимодействия с моделью COM, заслуживают краткого рассмотрения в этом разделе. Первая из них состоит в применении индексированных свойств, а вторая — в возможности передавать аргументы значения тем COM-методам, которым требуется ссылка.

Как вам должно быть уже известно, в C# свойство обычно связывается только с одним значением с помощью одного из аксессоров `get` или `set`. Но совсем иначе дело обстоит со свойствами модели COM. Поэтому, начиная с версии C# 4.0, в качестве выхода из этого затруднительного положения во время работы с COM-объектом появилась возможность пользоваться *индексированным свойством* для доступа к COM-свойству, имеющему несколько параметров. С этой целью имя свойства индексируется, почти так же, как это делается с помощью индексатора. Допустим, что имеется объект `myXLApp`, который относится к типу `Microsoft.Office.Interop.Execl.Application`.

В прошлом для установки строкового значения "OK" в ячейках C1–C3 электронной таблицы Excel можно было бы воспользоваться оператором, аналогичным следующему.

```
myXLapp.get_Range("C1", "C3").set_Value(Type.Missing, "OK");
```

В этой строке кода интервал ячеек электронной таблицы получается при вызове метода `get_Range()`, для чего достаточно указать начало и конец интервала. А значения задаются при вызове метода `set_Value()`, для чего достаточно указать тип (что не обязательно) и конкретное значение. В этих методах используются свойства `Range` и `Value`, поскольку у обоих свойств имеются два параметра. Поэтому в прошлом к ним нельзя было обращаться как к свойствам, но приходилось пользоваться упомянутыми выше методами. Кроме того, аргумент `Type.Missing` служил в качестве обычного заполнителя, который передавался для указания на тип, используемый по умолчанию. Но, начиная с версии C# 4.0, появилась возможно переписать приведенный выше оператор, приведя его к следующей более удобной форме.

```
myXLapp.Range["C1", "C3"].Value = "OK";
```

В этом случае значения интервала ячеек электронной таблицы передаются с использованием синтаксиса индексаторов, а заполнитель `Type.Missing` уже не нужен, поскольку данный параметр теперь задается по умолчанию.

Как правило, при определении в методе параметра `ref` приходится передавать ссылку на этот параметр. Но, работая с моделью COM, можно передавать параметру `ref` значение, не заключая его предварительно в оболочку объекта. Дело в том, что компилятор будет автоматически создавать временный аргумент, который уже заключен в оболочку объекта, и поэтому указывать параметр `ref` в списке аргументов уже не нужно.

Дружественные сборки

Одну сборку можно сделать *дружественной* по отношению к другой. Такой сборке доступны закрытые члены дружественной ей сборки. Благодаря этому средству становится возможным коллективное использование членов выбранных сборок, причем эти члены не нужно делать открытыми. Для того чтобы объявить дружественную сборку, необходимо воспользоваться атрибутом `InternalsVisibleTo`.

Разные ключевые слова

В заключение этой главы в частности и всей части I вообще будут вкратце представлены ключевые слова, определенные в C# и не упоминавшиеся в предыдущих главах данной книги.

Ключевое слов `lock`

Ключевое слово `lock` используется при создании многопоточных программ. Подробнее оно рассматривается в главе 23, где речь пойдет о многопоточном программировании. Но ради полноты изложения ниже приведено краткое описание этого ключевого слова.

Программа на C# может состоять из нескольких *потоков исполнения*. В этом случае программа считается *многопоточной*, и отдельные ее части выполняются параллельно, т.е. одновременно и независимо друг от друга. В связи с такой организацией программы возникает особого рода затруднение, когда два потока пытаются воспользоваться ресурсом, которым можно пользоваться только по очереди. Для разрешения этого затруднения можно создать *критический раздел кода*, который будет одновременно выполняться одним и только одним потоком. И это делается с помощью ключевого слова `lock`. Ниже приведена общая форма этого ключевого слова:

```
lock(obj) {
  // критический раздел кода
}
```

где `obj` обозначает объект, для которого согласуется блокировка кода. Если один поток уже вошел в критический раздел кода, то второму потоку придется ждать до тех пор, пока первый поток не выйдет из данного критического раздела кода. Когда же первый поток покидает критический раздел кода, блокировка снимается и предоставляется второму потоку. С этого момента второй поток может выполнять критический раздел кода.

ПРИМЕЧАНИЕ

Более подробно ключевое слово `lock` рассматривается в главе 23.

Ключевое слово `readonly`

Отдельное поле можно сделать доступным в классе только для чтения, объявив его как `readonly`. Значение такого поля можно установить только с помощью инициализатора, когда оно объявляется или же когда ему присваивается значение в конструкторе. После того как значение доступного только для чтения поля будет установлено, оно не подлежит изменению за пределами конструктора. Следовательно, поле типа `readonly` удобно для установки фиксированного значения с помощью конструктора. Такое поле можно, например, использовать для обозначения размера массива, который часто используется в программе. Допускаются как статические, так и нестатические поля типа `readonly`.

ПРИМЕЧАНИЕ

Несмотря на кажущееся сходство, поля типа `readonly` не следует путать с полями типа `const`, которые рассматриваются далее в этой главе.

Ниже приведен пример применения поля с ключевым словом `readonly`.

```
// Продемонстрировать применение поля с ключевым словом readonly.

using System;

class MyClass {
  public static readonly int SIZE = 10;
}
```

```
class DemoReadOnly {
  static void Main() {
    int[] source = new int[MyClass.SIZE];
    int[] target = new int[MyClass.SIZE];

    // Присвоить ряд значений элементам массива source.
    for(int i=0; i < MyClass.SIZE; i++)
      source[i] = i;

    foreach(int i in source)
      Console.Write(i + " ");

    Console.WriteLine();

    // Перенести обращенную копию массива source в массив target.
    for(int i = MyClass.SIZE-1, j = 0; i > 0; i--, j++)
      target[j] = source[i];

    foreach(int i in target)
      Console.Write(i + " ");

    Console.WriteLine();

// MyClass.SIZE = 100; // Ошибка!!! Не подлежит изменению!
  }
}
```

В данном примере поле `MyClass.SIZE` инициализируется значением 10. После этого его можно использовать, но не изменять. Для того чтобы убедиться в этом, удалите символы комментария в начале последней строки приведенного выше кода и попробуйте скомпилировать его. В итоге вы получите сообщение об ошибке.

Ключевые слова `const` и `volatile`

Ключевое слово, или модификатор, `const` служит для объявления полей и локальных переменных, которые нельзя изменять. Исходные значения таких полей и переменных должны устанавливаться при их объявлении. Следовательно, переменная с модификатором `const`, по существу, является константой. Например, в следующей строке кода:

```
const int i = 10;
```

создается переменная `i` типа `const` и устанавливается ее значение 10. Поле типа `const` очень похоже на поле типа `readonly`, но все же между ними есть отличие. Если поле типа `readonly` можно устанавливать в конструкторе, то поле типа `const` — нельзя.

Ключевое слово, или модификатор, `volatile` уведомляет компилятор о том, что значение поля может быть изменено двумя или более параллельно выполняющимися потоками. В этой ситуации одному потоку может быть неизвестно, когда поле было изменено другим потоком. И это очень важно, поскольку компилятор C# будет автоматически выполнять определенную оптимизацию, которая будет иметь результат лишь в том случае, если поле доступно только одному потоку. Для того чтобы подобной оптимизации не подвергалось общедоступное поле, оно объявляется как `volatile`.

Этим компилятор уведомляется о том, что значение поля типа `volatile` следует получать всякий раз, когда к нему осуществляется доступ.

Оператор `using`

Помимо рассматривавшейся ранее *директивы* `using`, имеется вторая форма ключевого слова `using` в виде *оператора*. Ниже приведены две общие формы этого оператора:

```
using (obj) {
  // использовать объект obj
}

using (тип obj = инициализатор) {
  // использовать объект obj
}
```

где *obj* является выражением, в результате вычисления которого должен быть получен объект, реализующий интерфейс `System.IDisposable`. Этот объект определяет переменную, которая будет использоваться в блоке оператора `using`. В первой форме объект объявляется вне оператора `using`, а во второй форме — в этом операторе. По завершении блока оператора `using` для объекта *obj* вызывается метод `Dispose()`, определенный в интерфейсе `System.IDisposable`. Таким образом, оператор `using` предоставляет средства, необходимые для автоматической утилизации объектов, когда они больше не нужны. Не следует, однако, забывать, что оператор `using` применяется только к объектам, реализующим интерфейс `System.IDisposable`.

В приведенном ниже примере демонстрируются обе формы оператора `using`.

```
// Продемонстрировать применение оператора using.

using System;
using System.IO;

class UsingDemo {
  static void Main() {
    try {
      StreamReader sr = new StreamReader("test.txt");

      // Использовать объект в операторе using.
      using(sr) {
        // ...
      }
    } catch(IOException exc) {
       // ...
    }

    try {
      // Создать объект класса StreamReader в операторе using.
      using(StreamReader sr2 = new StreamReader("test.txt")) {
        // ...
      }
    } catch(IOException exc) {
       // ...
    }
  }
}
```

В данном примере интерфейс `IDisposable` реализуется в классе `StreamReader` (посредством его базового класса `TextReader`). Поэтому он может использоваться в операторе `using`. По завершении этого оператора автоматически вызывается метод `Dispose()` для переменной потока, закрывая тем самым поток.

Как следует из приведенного выше примера, оператор `using` особенно полезен для работы с файлами, поскольку файл автоматически закрывается по завершении блока этого оператора, даже если он и завершается исключением. Таким образом, закрытие файла с помощью оператора `using` зачастую упрощает код обработки файлов. Разумеется, применение оператора `using` не ограничивается только работой с файлами. В среде .NET Framework имеется немало других ресурсов, реализующих интерфейс `IDisposable`. И всеми этими ресурсами можно управлять с помощью оператора `using`.

Ключевое слово `extern`

Ключевое слово `extern` находит два основных применения. Каждое из них рассматривается далее по порядку.

Объявление внешних методов

В первом своем применении ключевое слово `extern` было доступно с момента создания C#. Оно обозначает, что метод предоставляется в неуправляемом коде, который не является составной частью программы. Иными словами, метод предоставляется внешним кодом.

Для того чтобы объявить метод как внешний, достаточно указать в самом начале его объявления модификатор `extern`. Таким образом, общая форма объявления внешнего метода выглядит следующим образом.

```
extern возвращаемый_тип имя_метода(список_аргументов);
```

Обратите внимание на отсутствие фигурных скобок.

В данном варианте ключевое слово `extern` нередко применяется вместе с атрибутом `DllImport`, обозначающим библиотеку DLL, в которой содержится внешний метод. Атрибут `DllImport` принадлежит пространству имен `System.Runtime.InteropServices`. Он допускает несколько вариантов, но, как правило, достаточно указать лишь имя библиотеки DLL, в которой содержится внешний метод. Вообще говоря, внешние методы следует программировать на C. (Если же это делается на C++, то имя внешнего метода может быть изменено в библиотеке DLL путем дополнительного оформления типов.)

Для того чтобы стало понятнее, как пользоваться внешними методами, обратимся к примеру конкретной программы, состоящей из двух файлов. Ниже приведен исходный код C из первого файла `ExtMeth.c`, где определяется метод `AbsMax()`.

```
#include <stdlib.h>

int __declspec(dllexport) AbsMax(int a, int b) {
  return abs(a) < abs(b) ? abs(b) : abs(a);
}
```

В методе `AbsMax()` сравниваются абсолютные значения двух его параметров и возвращается самое большое из них. Обратите внимание на обозначение `__declspec(dllexport)`. Это специальное расширение языка C для программных

средств корпорации Microsoft. Оно уведомляет компилятор о необходимости экспортировать метод `AbsMax()` из библиотеки DLL, в которой он содержится. Для компилирования файла `ExtMeth.c` в командной строке указывается следующее.

```
CL /LD /MD ExtMeth.c
```

В итоге создается библиотечный файл DLL — `ExtMeth.dll`.

Далее следует программа на C#, в которой применяется внешний метод `AbsMax()`.

```
using System;
using System.Runtime.InteropServices;

class ExternMeth {

  // Здесь объявляется внешний метод.
  [DllImport("ExtMeth.dll")]
  public extern static int AbsMax(int a, int b);

  static void Main() {

    // Использовать внешний метод.
    int max = AbsMax(-10, -20);
    Console.WriteLine(max);
  }
}
```

Обратите внимание на использование атрибута `DllImport` в приведенной выше программе. Он уведомляет компилятор о наличии библиотеки DLL, содержащей внешний метод `AbsMax()`. В данном случае это файл `ExtMeth.dll`, созданный во время компиляции файла с исходным текстом метода `AbsMax()` на C. В результате выполнения данной программы на экран, как и ожидалось, выводится значение 20.

Объявление псевдонима внешней сборки

Во втором применении ключевое слово `extern` предоставляет псевдоним для внешней сборки, что полезно в тех случаях, когда в состав программы включаются две отдельные сборки с одним и тем же именем элемента. Так, если в сборке `test1` содержится класс `MyClass`, а в сборке `test2` класс с таким же именем, то при обращении к классу по этому имени в одной и той же программе может возникнуть конфликт.

Для разрешения подобного конфликта необходимо создать псевдоним каждой сборки. Это делается в два этапа. На первом этапе нужно указать псевдонимы, используя параметр компилятора `/r`, как в приведенном ниже примере.

```
/r:Asm1=test1
/r:Asm2=test2
```

А на втором этапе необходимо ввести операторы с ключевым словом `extern`, в которых делается ссылка на указанные выше псевдонимы. Ниже приведена форма такого оператора для создания псевдонима сборки.

```
extern alias имя_сборки;
```

Если продолжить приведенный выше пример, то в программе должны появиться следующие строки кода.

```
extern alias Asm1;
extern alias Asm2;
```

Теперь оба варианта класса `MyClass` будут доступны в программе по соответствующему псевдониму.

Рассмотрим полноценный пример программы, в которой демонстрируется применение внешних псевдонимов. Эта программа состоит из трех файлов. Ниже приведен исходный текст, который следует поместить в первый файл — `test1.cs`.

```
using System;

namespace MyNS {
  public class MyClass {
    public MyClass() {
      Console.WriteLine("Конструирование из файла MyClass1.dll.");
    }
  }
}
```

Далее следует исходный текст из файла `test2.cs`.

```
using System;

namespace MyNS {
  public class MyClass {
    public MyClass() {
      Console.WriteLine("Конструирование из файла MyClass2.dll.");
    }
  }
}
```

Обратите внимание на то, что в обоих файлах, `test1.cs` и `test2.cs`, объявляется пространство имен `MyNS` и что именно в этом пространстве в обоих файлах определяется класс `MyClass`. Следовательно, без псевдонима оба варианта класса `MyClass` будут недоступными ни одной из программ.

И наконец, ниже приведен исходный текст из третьего файла `test3.cs`, где используются оба варианта класса `MyClass` из файлов `test1.cs` и `test2.cs`. Это становится возможным благодаря операторам с внешними псевдонимами.

```
// Операторы с внешними псевдонимами должны быть указаны в самом начале файла.
extern alias Asm1;
extern alias Asm2;

using System;

class Demo {
  static void Main() {
    Asm1::MyNS.MyClass t = new Asm1::MyNS.MyClass();
    Asm2::MyNS.MyClass t2 = new Asm2::MyNS.MyClass();
  }
}
```

Сначала следует скомпилировать файлы `test1.cs` и `test2.cs` в их библиотечные эквиваленты DLL. Для этого достаточно ввести в командной строке следующее.

```
csc /t:library test1.cs
csc /t:library test2.cs
```

Затем необходимо скомпилировать файл `test3.cs`, указав в командной строке

```
csc /r:Asm1=test1.dll /r:Asm2=test2.dll test3.cs
```

Обратите внимание на применение параметра `/r`, уведомляющего компилятор о том, что ссылка на метаданные находится в соответствующем файле. В данном случае псевдоним `Asm1` связывается с файлом `test1.dll`, а псевдоним `Asm2` — с файлом `test2.dll`.

В самой программе псевдонимы указываются в приведенных ниже операторах с модификатором `extern`, которые располагаются в самом начале файла.

```
extern alias Asm1;
extern alias Asm2;
```

А в методе `Main()` псевдонимы используются для разрешения неоднозначности ссылок на класс `MyClass`. Обратите внимание на следующее применение псевдонима для обращения к классу `MyClass`.

```
Asm1::MyNS.MyClass
```

В этой строке кода первым указывается псевдоним, затем оператор разрешения пространства имен, далее имя пространства имен, в котором находится класс с неоднозначным именем, и, наконец, имя самого класса, следующее после оператора-точки. Та же самая общая форма пригодна и для других внешних псевдонимов.

Ниже приведен результат выполнения данной программы.

```
Конструирование из файла MyClass1.dll.
Конструирование из файла MyClass2.dll.
```

Часть II

Библиотека C#

ГЛАВА 21 Пространство имен `System`

ГЛАВА 22 Строки и форматирование

ГЛАВА 23 Многопоточное программирование. Часть первая: основы

ГЛАВА 24 Многопоточное программирование. Часть вторая: библиотека TPL

ГЛАВА 25 Коллекции, перечислители и итераторы

ГЛАВА 26 Сетевые средства подключения к Интернету

В части II рассматривается библиотека C#. Как пояснялось в части I, используемая в C# библиотека на самом деле является библиотекой классов для среды .NET Framework. Поэтому материал этой части книги имеет отношение не только к языку C#, но и ко всей среде .NET Framework в целом.

Библиотека классов для среды .NET Framework организована по пространствам имен. Для использования отдельной части этой библиотеки, как правило, достаточно импортировать ее пространство имен, указав его с помощью директивы `using` в исходном тексте программы. Конечно, ничто не мешает определить имя отдельного элемента библиотеки полностью вместе с его пространством имен, но ведь намного проще импортировать сразу все пространство имен.

Библиотека среды .NET Framework довольно обширна, и поэтому ее полное описание выходит за рамки этой книги. (На самом деле для этого потребовалась бы отдельная и довольно объемистая книга!) Поэтому в части II рассматриваются лишь самые основные элементы данной библиотеки, многие из которых находятся в пространстве имен `System`. Кроме того, в этой части описываются классы коллекций, а также вопросы организации многопоточной обработки и сетей.

ПРИМЕЧАНИЕ

Классы ввода-вывода подробно рассматривались в главе 14.

21 ГЛАВА

Пространство имен System

В этой главе речь пойдет о пространстве имен `System`. Это пространство имен самого верхнего уровня в библиотеке классов для среды .NET Framework. В нем непосредственно находятся те классы, структуры, интерфейсы, делегаты и перечисления, которые чаще всего применяются в программах на C# или же считаются неотъемлемой частью среды .NET Framework. Таким образом, пространство имен `System` составляет ядро рассматриваемой здесь библиотеки классов.

Кроме того, в пространство имен `System` входит много вложенных пространств имен, поддерживающих отдельные подсистемы, например `System.Net`. Некоторые из этих пространств имен рассматриваются далее в этой книге. А в этой главе речь пойдет только о членах самого пространства имен `System`.

Члены пространства имен `System`

Помимо большого количества классов исключений, в пространстве имен содержатся приведенные ниже классы.

ActivationContext	Activator	AppDomain
AppDomainManager	AppDomainSetup	ApplicationId
ApplicationIdentity	Array	AssemblyLoadEventArgs
Attribute	AttributeUsageAttribute	BitConverter
Buffer	CharEnumerator	CLSCompliantAttribute
Console	ConsoleCancelEventArgs	ContextBoundObject
ContextStaticAttribute	Convert	DBNull
Delegate	Enum	Environment
EventArgs	Exception	FileStyleUriParser
FlagsAttribute	FtpStyleUriParser	GC
GenericUriParser	GopherStyleUriParser	HttpStyleUriParser
Lazy<T>	Lazy<T, TMetadata>	LdapStyleUriParser
LoaderOptimizationAttribute	LocalDataStoreSlot	MarshalByRefObject
Math	MTAThreadAttribute	MulticastDelegate
NetPipeStyleUriParser	NetTcpStyleUriParser	NewsStyleUriParser
NonSerializedAttribute	Nullable	Object
ObsoleteAttribute	OperatingSystem	ParamArrayAttribute
Random	ResolveEventArgs	SerializableAttribute
STAThreadAttribute	String	StringComparer
ThreadStaticAttribute	TimeZone	TimeZoneInfo
TimeZoneInfo.AdjustmentRule	Tuple	Tuple<...> (различные формы)
Type	UnhandledExceptionEventArgs	Uri
UriBuilder	UriParser	UriTemplate
UriTemplateEquivalenceComparer	UriTemplateMatch	UriTemplateTable
UriTypeConverter	ValueType	Version
WeakReference		

Ниже приведены структуры, определенные в пространстве имен `System`.

ArgIterator	ArraySegment<T>	Boolean
Byte	Char	ConsoleKeyInfo
DateTime	DateTimeOffset	Decimal
Double	Guid	Int16
Int32	Int64	IntPtr
ModuleHandle	Nullable<T>	RuntimeArgumentHandle
RuntimeFieldHandle	RuntimeMethodHandle	RuntimeTypeHandle
Sbyte	Single	TimeSpan
TimeZoneInfo.TransitionTime	TypedReference	UInt16
UInt32	UInt64	UIntPtr
Void		

В пространстве имен `System` определены приведенные ниже интерфейсы.

_AppDomain	IappDomainSetup	IAsyncResult
ICloneable	IComparable	IComparable<T>
IConvertible	ICustomFormatter	IDisposable
IEquatable<T>	IFormatProvider	IFormattable
IObservable<T>	IObserver<T>	IServiceProvider

Ниже приведены делегаты, определенные в пространстве имен `System`.

Action	Action<...> (различные формы)	AppDomainInitializer
AssemblyLoadEventHandler	AsyncCallback	Comparison<T>
ConsoleCancelEventHandler	Converter<TInput, VOutput>	CrossAppDomainDelegate
EventHandler	EventHandler<TEventArgs>	Func<...> (различные формы)
Predicate<T>	ResolveEventHandler	UnhandledExceptionEventHandler

В пространстве имен `System` определены приведенные ниже перечисления.

ActivationContext.contextForm	AppDomainManagerInitializationOptions	AttributeTargets
Base64FormattingOptions	ConsoleColor	ConsoleKey
ConsoleModifiers	ConsoleSpecialKey	DateTimeKind
DayOfWeek	Environment.SpecialFolder	Environment.SpecialFolderOption
EnvironmentVariableTarget	GCCollectionMode	GCNotificationStatus
GenericUriParserOptions	LoaderOptimization	MidpointRounding
PlatformID	StringComparison	StringSplitOptions
TypeCode	UriComponents	UriFormat
UriHostNameType	UriIdnScope	UriKind
UriPartial		

Как следует из приведенных выше таблиц, пространство имен `System` довольно обширно, поэтому в одной главе невозможно рассмотреть подробно все его составляющие. К тому же, некоторые члены пространства имен `System`, в том числе `Nullable<T>`, `Type`, `Exception` и `Attribute`, уже рассматривались в части I или будут представлены в последующих главах части II. И наконец, класс `System.String`, в котором определяется тип `string` для символьных строк в C#, обсуждается вместе с вопросами форматирования в главе 22. В силу этих причин в настоящей главе рассматриваются только те члены данного пространства имен, которые чаще всего применяются в программировании на C# и не поясняются полностью в остальных главах книги.

Класс `Math`

В классе `Math` определен ряд стандартных математических операций, в том числе извлечение квадратного корня, вычисление синуса, косинуса и логарифмов. Класс `Math` является статическим, а это означает, что все методы, определенные в нем, относятся к типу `static`, объекты типа `Math` не конструируются, а сам класс `Math` неявно герметичен и не может наследоваться. Методы, определенные в классе `Math`, перечислены в табл. 21.1, где все углы указаны в радианах.

В классе `Math` определены также два следующих поля:

```
public const double E
public const double PI
```

где `E` — значение основания натурального логарифма числа, которое обычно обозначается как *e*; а `PI` — значение числа пи.

Таблица 21.1. Методы, определенные в классе `Math`

Метод	Описание
`public static double Abs(double value)`	Возвращает абсолютную величину *value*
`public static float Abs(float value)`	Возвращает абсолютную величину *value*
`public static decimal Abs(decimal value)`	Возвращает абсолютную величину *value*
`public static int Abs(int value)`	Возвращает абсолютную величину *value*
`public static short Abs(short value)`	Возвращает абсолютную величину *value*
`public static long Abs(long value)`	Возвращает абсолютную величину *value*
`public static sbyte Abs(sbyte value)`	Возвращает абсолютную величину *value*
`public static double Acos(double d)`	Возвращает арккосинус *d*. Значение *d* должно находиться в пределах от -1 до 1
`public static double Asin(double d)`	Возвращает арксинус *d*. Значение *d* должно находиться в пределах от -1 до 1
`public static double Atan(double d)`	Возвращает арктангенс *d*
`public static double Atan2(double y, double x)`	Возвращает арктангенс частного от деления *y / x*
`public static long BigMul(int a, int b)`	Возвращает произведение `a*b` в виде значения типа `long`, исключая переполнение
`public static double Ceiling(double a)`	Возвращает наименьшее целое, которое представлено в виде значения с плавающей точкой и не меньше *a*. Так, если *a* равно 1,02, метод `Ceiling()` возвращает значение 2,0. А если *a* равно -1,02, то метод `Ceiling()` возвращает значение -1
`public static double Ceiling(decimal d)`	Возвращает наименьшее целое, которое представлено в виде значения десятичного типа и не меньше *d*. Так, если *d* равно 1,02, метод `Ceiling()` возвращает значение 2,0. А если *d* равно -1,02, то метод `Ceiling()` возвращает значение -1
`public static double Cos(double d)`	Возвращает косинус *d*
`public static double Cosh(double d)`	Возвращает гиперболический косинус *d*
`public static int DivRem(int a, int b, out int result)`	Возвращает частное от деления *a / b*, а остаток — в виде параметра *result* типа `out`
`public static long DivRem(long a, long b,out long result)`	Возвращает частное от деления *a / b*, а остаток — в виде параметра *result* типа `out`

Продолжение табл. 21.1

Метод	Описание
`public static double Exp(double d)`	Возвращает основание натурального логарифма e, возведенное в степень *d*
`public static decimal Floor(decimal d)`	Возвращает наибольшее целое, которое представлено в виде значения десятичного типа и не больше *d*. Так, если *d* равно 1,02, метод `Floor()` возвращает значение 1,0. А если *d* равно -1,02, метод `Floor()` возвращает значение -2
`public static double Floor(double d)`	Возвращает наибольшее целое, которое представлено в виде значения с плавающей точкой и не больше *d*. Так, если *d* равно 1,02, метод `Floor()` возвращает значение 1,0. А если *d* равно -1,02, метод `Floor()` возвращает значение -2
`public static double IEEERemainder(double x, double y)`	Возвращает остаток от деления *x* / *y*
`public static double Log(double d)`	Возвращает натуральный логарифм значения *d*
`public static double Log(double d, double newBase)`	Возвращает натуральный логарифм по основанию *newBase* значения *d*
`public static double Log10(double d)`	Возвращает логарифм по основанию 10 значения *d*
`public static double Max(double val1, double val2)`	Возвращает большее из значений *val1* и *val2*
`public static float Max(float val1, float val2)`	Возвращает большее из значений *val1* и *val2*
`public static decimal Max(decimal val1, decimal val2)`	Возвращает большее из значений *val1* и *val2*
`public static int Max(int val1, int val2)`	Возвращает большее из значений *val1* и *val2*
`public static short Max(short val1, short val2)`	Возвращает большее из значений *val1* и *val2*
`public static long Max(long val1, long val2)`	Возвращает большее из значений *val1* и *val2*
`public static uint Max(uint val1, uint val2)`	Возвращает большее из значений *val1* и *val2*
`public static ushort Max(ushort val1, ushort val2)`	Возвращает большее из значений *val1* и *val2*
`public static ulong Max(ulong val1, ulong val2)`	Возвращает большее из значений *val1* и *val2*
`public static byte Max(byte val1, byte val2)`	Возвращает большее из значений *val1* и *val2*
`public static sbyte Max(sbyte val1, sbyte val2)`	Возвращает большее из значений *val1* и *val2*

Продолжение табл. 21.1

Метод	Описание
`public static double Min(double val1, double val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static float Min(float val1, float val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static decimal Min(decimal val1, decimal val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static int Min(int val1, int val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static short Min(short val1, short val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static long Min(long val1, long val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static uint Min(uint val1, uint val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static ushort Min(ushort val1, ushort val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static ulong Min(ulong val1, ulong val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static byte Min(byte val1, byte val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static sbyte Min(sbyte val1, sbyte val2)`	Возвращает меньшее из значений *val1* и *val2*
`public static double Pow(double x, double y)`	Возвращает значение `x`, возведенное в степень *y* (x^y)
`public static double Round(double a)`	Возвращает значение *a*, округленное до ближайшего целого числа
`public static decimal Round(decimal d)`	Возвращает значение *d*, округленное до ближайшего целого числа
`public static double Round(double value, int digits)`	Возвращает значение *value*, округленное до числа, количество цифр в дробной части которого равно значению параметра *digits*
`public static decimal Round(decimal d, int digits)`	Возвращает значение *d*, округленное до числа, количество цифр в дробной части которого равно значению *digits*
`public static double Round(double value, MidpointRounding mode)`	Возвращает значение *value*, округленное до ближайшего целого числа в режиме, определяемом параметром *mode*
`public static decimal Round(decimal d, MidpointRounding mode)`	Возвращает значение *d*, округленное до ближайшего целого числа в режиме, определяемом параметром *mode*
`public static double Round(double value, int digits, MidpointRounding mode)`	Возвращает значение *value*, округленное до числа, количество цифр в дробной части которого равно значению *digits*, а параметр `mode` определяет режим округления

Окончание табл. 21.1

Метод	Описание
`public static decimal Round(decimal d, int digits, MidpointRounding mode)`	Возвращает значение `d`, округленное до числа, количество цифр в дробной части которого равно значению `digits`, а параметр `mode` определяет режим округления
`public static int Sign(double value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static int Sign(float value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static int Sign(decimal value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static int Sign(int value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static int Sign(short value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static int Sign(long value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static int Sign(sbyte value)`	Возвращает -1, если значение `value` меньше нуля; 0, если значение `value` равно нулю; и 1, если значение `value` больше нуля
`public static double Sin(double a)`	Возвращает синус числа `a`
`public static double Sinh(double value)`	Возвращает гиперболический синус числа `value`
`public static double Sqrt(double d)`	Возвращает квадратный корень числа `d`
`public static double Tan(double a)`	Возвращает тангенс числа `a`
`public static double Tanh(double value)`	Возвращает гиперболический тангенс числа `value`
`public static double Truncate(double d)`	Возвращает целую часть числа `d`
`public static decimal Truncate(decimal d)`	Возвращает целую часть числа `d`

В приведенном ниже примере программы метод `Sqrt()` служит для расчета гипотенузы по длине противоположных сторон прямоугольного треугольника согласно теореме Пифагора.

```
// Расчет гипотенузы по теореме Пифагора.

using System;

class Pythagorean {
  static void Main() {
    double s1;
    double s2;
    double hypot;
    string str;

    Console.WriteLine("Введите длину первой стороны треугольника: ");
    str = Console.ReadLine();
    s1 = Double.Parse(str);

    Console.WriteLine("Введите длину второй стороны треугольника: ");
    str = Console.ReadLine();
    s2 = Double.Parse(str);

    hypot = Math.Sqrt(s1*s1 + s2*s2);
    Console.WriteLine("Длина гипотенузы равна " + hypot);
  }
}
```

Ниже приведен один из возможных результатов выполнения этой программы.

```
Введите длину первой стороны треугольника: 3
Введите длину второй стороны треугольника: 4
Длина гипотенузы равна: 5
```

Далее следует пример программы, в которой метод `Pow()` служит для расчета первоначальных капиталовложений, требующихся для получения предполагаемой будущей стоимости, исходя из годовой нормы прибыли и количества лет. Ниже приведена формула для расчета первоначальных капиталовложений.

$$\textit{первоначальные капиталовложения} = \textit{будущая стоимость} \;/\; (1 + \textit{норма прибыли})^{\textit{количество лет}}$$

В вызове метода `Pow()` необходимо указывать аргументы типа `double`, поэтому норма прибыли и количество лет задаются в виде значений типа `double`. А первоначальные капиталовложения и будущая стоимость задаются в виде значений типа `decimal`.

```
/* Рассчитать первоначальные капиталовложения, необходимые
  для получения заданной будущей стоимости, исходя из
  годовой нормы прибыли и количества лет. */

using System;

class InitialInvestment {
  static void Main() {
    decimal initInvest; // первоначальные капиталовложения
    decimal futVal;     // будущая стоимость

    double numYears;    // количество лет
    double intRate;     // годовая норма прибыли
```

```
    string str;

    Console.Write("Введите будущую стоимость: ");
    str = Console.ReadLine();
    try {
      futVal = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Введите норму прибыли (например, 0.085): ");
    str = Console.ReadLine();
    try {
      intRate = Double.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.Write("Введите количество лет: ");
    str = Console.ReadLine();
    try {
      numYears = Double.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    initInvest =
     futVal / (decimal) Math.Pow(intRate+1.0, numYears);

    Console.WriteLine("Необходимые первоначальные капиталовложения: {0:C}",
                      initInvest);
  }
}
```

Ниже приведен один из возможных результатов выполнения этой программы.

```
Введите будущую стоимость: 10000
Введите норму прибыли (например, 0.085): 0.07
Введите количество лет: 10
Необходимые первоначальные капиталовложения: $5,083.49
```

Структуры .NET, соответствующие встроенным типам значений

Структуры, соответствующие встроенным в C# типам значений, были представлены в главе 14, где они упоминались в связи с преобразованием строк, содержащих числовые значения в удобочитаемой форме, в эквивалентные двоичные значения. В этом разделе структуры .NET рассматриваются более подобно.

Имена структур .NET и соответствующие им ключевые слова, обозначающие типы значений в C#, перечислены в приведенной ниже таблице.

Имя структуры в .NET	Имя типа значения в C#
System.Boolean	bool
System.Char	char
System.Decimal	decimal
System.Double	double
System.Single	float
System.Int16	short
System.Int32	int
System.Int64	long
System.UInt16	ushort
System.UInt32	uint
System.UInt64	ulong
System.Byte	byte
System.Sbyte	sbyte

Используя члены, определенные в этих структурах, можно выполнять операции над значениями простых типов данных. Все перечисленные выше структуры рассматриваются далее по порядку.

ПРИМЕЧАНИЕ

Некоторые методы, определенные в структурах, соответствующих встроенным в C# типам значений, принимают параметры типа `IFormatProvider` или `NumberStyles`. Тип `IFormatProvider` вкратце описывается далее в этой главе, а тип `NumberStyles` представляет собой перечисление из пространства имен `System.Globalization`. Вопросы форматирования подробнее рассматриваются в главе 22.

Структуры целочисленных типов данных

Ниже перечислены структуры целочисленных типов данных.

Byte	SByte	Int16	Uint16
Int32	UInt32	Int64	UInt64

Каждая из этих структур содержит одинаковое количество членов. В табл. 21.2 для примера перечислены члены структуры `Int32`. Аналогичные члены в виде методов имеются и у других структур, за исключением целочисленного типа, который они представляют.

Помимо перечисленных выше методов, в структурах целочисленных типов данных определены следующие поля типа `const`.

```
MaxValue
MinValue
```

В каждой структуре эти поля содержат наибольшее и наименьшее значения, допустимые для данных соответствующего целочисленного типа.

Во всех структурах целочисленных типов данных реализуются следующие интерфейсы: `IComparable`, `IComparable<T>`, `IConvertible`, `IFormattable`

и IEquatable<T>, где параметр обобщенного типа T заменяется соответствующим типом данных. Например, в структуре Int32 вместо T подставляется тип int.

Таблица 21.2. Методы, поддерживаемые структурой Int32

Метод	Назначение
public int CompareTo(object *value*)	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; и, наконец, положительное значение, если вызывающий объект имеет большее значение
public int CompareTo(int *value*)	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; и, наконец, положительное значение, если вызывающий объект имеет большее значение
public override bool Equals(object *obj*)	Возвращает логическое значение true, если значение вызывающего объекта равно значению параметра *obj*
public bool Equals(int *obj*)	Возвращает логическое значение true, если значение вызывающего объекта равно значению параметра *obj*
public override int GetHashCode()	Возвращает хеш-код для вызывающего объекта
public TypeCode GetTypeCode()	Возвращает значение перечисления TypeCode для эквивалентного типа. Например, для структуры Int32 возвращается значение TypeCode.Int32
public static int Parse(string *s*)	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*. Если числовое значение не представлено в строке так, как определено в структуре данного типа, то генерируется исключение
public static int Parse(string *s*, IformatProvider *provider*)	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром *provider*. Если числовое значение не представлено в строке так, как определено в структуре данного типа, то генерируется исключение
public static int Parse(string *s*, NumberStyles *styles*)	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*, с использованием данных о стилях, определяемых параметром *styles*. Если числовое значение не представлено в строке так, как определено в структуре данного типа, то генерируется исключение

Окончание табл. 21.2

Метод	Назначение
`public static int Parse(string s, NumberStyles styles, IformatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде строки символьной `s`, с использованием данных о стилях, определяемых параметром `styles`, а также форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`. Если числовое значение не представлено в строке так, как определено в структуре данного типа, то генерируется исключение
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString(string format)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`
`public string ToString(IformatProvider provider)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public string ToString(string format, IformatProvider provider)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public static bool TryParse(string s, out int result)`	Предпринимает попытку преобразовать числовое значение, заданное в виде символьной строки `s`, в двоичное значение. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`, а иначе возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования
`public static bool TryParse(string s, NumberStyles styles, IformatProvider provider, out int result)`	Предпринимает попытку преобразовать числовое значение, заданное в виде символьной строки `s`, в двоичное значение с использованием информации о стилях, обозначаемых параметром `styles`, а также форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`, а иначе возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования

Структуры типов данных с плавающей точкой

Типам данных с плавающей точкой соответствуют только две структуры: `Double` и `Single`. Структура `Single` представляет тип `float`. Ее методы перечислены в табл. 21.3, а поля — в табл. 21.4. Структура `Double` представляет тип `double`.

Ее методы перечислены в табл. 21.5, а поля — в табл. 21.6. Как и в структурах целочисленных типов данных, при вызове метода `Parse()` или `ToString()` из структур типов данных с плавающей точкой можно указывать информацию, характерную для конкретной культурной среды, а также данные форматирования.

Таблица 21.3. Методы, поддерживаемые структурой `Single`

Метод	Назначение
`public int CompareTo(object value)`	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное число, если вызывающий объект имеет меньшее значение, и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public int CompareTo(float value)`	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное число, если вызывающий объект имеет меньшее значение, и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public override bool Equals(object obj)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению *obj*
`public bool Equals(float obj)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению *obj*
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение из перечисления `TypeCode` для структуры `Single`, т.е. `TypeCode.Single`
`public static bool IsInfinity(float f)`	Возвращает логическое значение `true`, если значение *f* представляет плюс или минус бесконечность. В противном случае возвращает логическое значение `false`
`public static bool IsNaN(float f)`	Возвращает логическое значение `true`, если значение *f* не является числовым. В противном случае возвращает логическое значение `false`
`public static bool IsPositiveInfinity(float f)`	Возвращает логическое значение `true`, если значение *f* представляет плюс бесконечность. В противном случае возвращает логическое значение `false`
`public static bool IsNegativeInfinity(float f)`	Возвращает логическое значение `true`, если значение *f* представляет минус бесконечность. В противном случае возвращает логическое значение `false`
`public static float Parse(string s)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*. Если в строке не представлено числовое значение типа `float`, то генерируется исключение
`public static float Parse(string s, IformatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром *provider*. Если в строке не представлено числовое значение типа `float`, то генерируется исключение

Окончание табл. 21.3

Метод	Назначение
`public static float Parse(string s, NumberStyles styles)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки `s`, с использованием данных о стилях, определяемых параметром `styles`. Если в строке не представлено числовое значение типа `float`, то генерируется исключение
`public static float Parse(string s, NumberStyles styles, IformatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки `s`, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`, а также сведений о стилях, обозначаемых параметром `styles`. Если в строке не представлено числовое значение типа `float`, то генерируется исключение
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString(string format)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`
`public string ToString(IformatProvider provider)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public string ToString(string format, IformatProvider provider)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public static bool TryParse(string s, out float result)`	Предпринимает попытку преобразовать число, заданное в виде символьной строки `s`, в значение типа `float`. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`, а иначе возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования
`public static bool TryParse(string s, NumberStyles styles, IformatProvider provider, out float result)`	Предпринимает попытку преобразовать числовое значение, заданное в виде символьной строки `s`, в значение типа `float`, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`, а также сведений о стилях, обозначаемых параметром `styles`. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`, а иначе возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования

Таблица 21.4. Поля, поддерживаемые структурой `Single`

Поле	Назначение
`public const float Epsilon`	Наименьшее ненулевое положительное значение
`public const float MaxValue`	Наибольшее значение, допустимое для данных типа `float`
`public const float MinValue`	Наименьшее значение, допустимое для данных типа `float`
`public const float NaN`	Значение, не являющееся числом
`public const float NegativeInfinity`	Значение, представляющее минус бесконечность
`public const float PositiveInfinity`	Значение, представляющее плюс бесконечность

Таблица 21.5. Методы, поддерживаемые структурой `Double`

Метод	Назначение
`public int CompareTo(object` *value*)	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное число, если вызывающий объект имеет меньшее значение, и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public int CompareTo(double` *value*)	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное число, если вызывающий объект имеет меньшее значение, и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public override bool Equals(object` *obj*)	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению *obj*
`public bool Equals(double` *obj*)	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению *obj*
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение из перечисления `TypeCode` для структуры `Double`, т.е. `TypeCode.Double`
`public static bool IsInfinity(double` *d*)	Возвращает логическое значение `true`, если значение *d* представляет плюс или минус бесконечность. В противном случае возвращает логическое значение `false`
`public static bool IsNaN(double` *d*)	Возвращает логическое значение `true`, если значение *d* не является числовым. В противном случае возвращает логическое значение `false`
`public static bool IsPositiveInfinity(double` *d*)	Возвращает логическое значение `true`, если значение *d* представляет плюс бесконечность. В противном случае возвращает логическое значение `false`

Продолжение табл. 21.5

Метод	Назначение
`public static bool IsNegativeInfinity(double d)`	Возвращает логическое значение `true`, если значение `d` представляет минус бесконечность. В противном случае возвращает логическое значение `false`
`public static double Parse(string s)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки `s`. Если в строке не представлено числовое значение типа `double`, то генерируется исключение
`public static double Parse(string s, IFormatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки `s`, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`. Если в строке не представлено числовое значение типа `double`, то генерируется исключение
`public static double Parse(string s, NumberStyles styles)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки `s`, с использованием данных о стилях, определяемых параметром `styles`. Если в строке не представлено числовое значение типа `double`, то генерируется исключение
`public static double Parse(string s, NumberStyles styles, IFormatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки `s`, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`, а также данных о стилях, обозначаемых параметром `styles`. Если в строке не представлено числовое значение типа `double`, то генерируется исключение
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта
`public string ToString(string format)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`
`public string ToString(IformatProvider provider)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public string ToString(string format, IformatProvider provider)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public static bool TryParse(string s, out double result)`	Предпринимает попытку преобразовать число, заданное в виде символьной строки `s`, в значение типа `double`. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`, а иначе возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования

Окончание табл. 21.5

Метод	Назначение
`public static bool TryParse(string s, NumberStyles styles, IFormatProvider provider, out double result)`	Предпринимает попытку преобразовать числовое значение, заданное в виде символьной строки `s`, в значение типа `double`, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`, а также сведений о стилях, обозначаемых параметром `styles`. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`, а иначе возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования

Таблица 21.6. Поля, поддерживаемые структурой `Double`

Поле	Назначение
`public const double Epsilon`	Наименьшее ненулевое положительное значение
`public const double MaxValue`	Наибольшее значение, допустимое для данных типа `double`
`public const double MinValue`	Наименьшее значение, допустимое для данных типа `double`
`public const double NaN`	Значение, не являющееся числом
`public const double NegativeInfinity`	Значение, представляющее минус бесконечность
`public const double PositiveInfinity`	Значение, представляющее плюс бесконечность

Структура `Decimal`

Структура `Decimal` немного сложнее, чем ее аналоги для целочисленных типов данных, а также типов данных с плавающей точкой. Она содержит немало конструкторов, полей, методов и операторов, способствующих использованию типа `decimal` вместе с другими числовыми типами, поддерживаемыми в C#. Так, целый ряд методов из этой структуры обеспечивает преобразование типа `decimal` в другие числовые типы.

В структуре `Decimal` определено восемь открытых конструкторов. Ниже приведены шесть наиболее часто используемых из них.

```
public Decimal(int значение)
public Decimal(uint значение)
public Decimal(long значение)
public Decimal(ulong значение)
public Decimal(float значение)
public Decimal(double значение)
```

Каждый из этих конструкторов создает объект типа `Decimal` из значения указанного типа.

Кроме того, объект типа `Decimal` может быть создан из отдельно указываемых составляющих с помощью следующего конструктора.

```
public Decimal(int lo, int mid, int hi, bool IsNegative, byte scale)
```

Десятичное значение состоит из трех частей. Первую часть составляет 96-разрядное целое значение, вторую — флаг знака, третью — масштабный коэффициент. В частности, 96-разрядное целое значение передается конструктору тремя 32-разрядными фрагментами с помощью параметров *lo*, *mid* и *hi*; знак флага — с помощью параметра *IsNegative*, причем логическое значение `false` этого параметра обозначает положительное число, тогда как логическое значение `true` обозначает отрицательное число; а масштабный коэффициент — с помощью параметра *scale*, принимающего значения от 0 до 28. Этот коэффициент обозначает степень числа 10 (т.е. 10^{scale}), на которую делится число для получения его дробной части.

Вместо того чтобы передавать каждую составляющую объекта типа `Decimal` отдельно, все его составляющие можно указать в массиве, используя следующий конструктор.

```
public Decimal(int[] bits)
```

Три первых элемента типа `int` в массиве *bits* содержат 96-разрядное целое значение; 31-й разряд содержимого элемента `bits[3]` обозначает флаг знака (0 — положительное число, 1 — отрицательное число); а в разрядах 16–23 содержится масштабный коэффициент.

В структуре `Decimal` реализуются следующие интерфейсы: `IComparable`, `IComparable<decimal>`, `IConvertible`, `IFormattable`, `IEquatable<decimal>`, а также `IDeserializationCallback`.

В приведенном ниже примере программы значение типа `decimal` формируется вручную.

```
// Сформировать десятичное число вручную.

using System;

class CreateDec {
  static void Main() {
    decimal d = new decimal(12345, 0, 0, false, 2);
    Console.WriteLine(d);
  }
}
```

Эта программа дает следующий результат.

```
123.45
```

В данном примере значение 96-разрядного целого числа равно 12345. У него положительный знак и два десятичных разряда в дробной части.

Методы, определенные в структуре `Decimal`, приведены в табл. 21.7, а поля — в табл. 21.8. Кроме того, в структуре `Decimal` определяется обширный ряд операторов и преобразований, позволяющих использовать десятичные значения вместе со значениями других типов в выражениях. Правила, устанавливающие порядок присваивания десятичных значений и их применения в выражениях, представлены в главе 3.

Таблица 21.7. Методы, определенные в структуре `Decimal`

Метод	Назначение
`public static decimal Add(decimal d1, decimal d2)`	Возвращает значение *d1* + *d2*
`public static decimal Ceiling(d)`	Возвращает наименьшее целое, которое представлено в виде значения типа `decimal` и не меньше *d*. Так, если `d` равно 1,02, метод `Ceiling()` возвращает значение 2,0. А если `d` равно -1,02, то метод `Ceiling()` возвращает значение -1
`public static int Compare(decimal d1, decimal d2)`	Сравнивает числовое значение *d1* со значением *d2*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если *d1* меньше *d2*; и, наконец, положительное значение, если *d1* больше *d2*
`public int CompareTo(object value)`	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public int CompareTo(decimal value)`	Сравнивает числовое значение вызывающего объекта со значением *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public static decimal Divide(decimal d1, decimal d2)`	Возвращает частное от деления *d1* / *d2*
`public bool Equals(decimal value)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению *value*
`public override bool Equals(object value)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению *value*
`public static bool Equals(decimal d1, decimal d2)`	Возвращает логическое значение `true`, если если *d1* равно *d2*
`public static decimal Floor(decimal d)`	Возвращает наибольшее целое, которое представлено в виде значения типа `decimal` и не больше *d*. Так, если `d` равно 1,02, метод `Floor()` возвращает значение 1,0. А если `d` равно -1,02, метод `Floor()` возвращает значение -2
`public static decimal FromOACurrency(long cy)`	Преобразует значение *cy* из формата денежной единицы, применяемого в компоненте OLE Automation, в его десятичный эквивалент и возвращает полученный результат

Продолжение табл. 21.7

Метод	Назначение
`public static int[] GetBits(decimal d)`	Возвращает двоичное представление значения *d* в виде массива типа `int`. Организация этого массива описана в тексте настоящего раздела
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение из перечисления `TypeCode` для структуры `Decimal`, т.е. `TypeCode.Decimal`
`public static decimal Multiply(decimal d1, decimal d2)`	Возвращает произведение *d1* * *d2*
`public static decimal Negate(decimal d)`	Возвращает значение *-d*
`public static decimal Parse(string s)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*. Если в строке не представлено числовое значение типа `decimal`, то генерируется исключение
`public static decimal Parse(string s, IFormatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром *provider*. Если в строке не представлено числовое значение типа `decimal`, то генерируется исключение
`public static decimal Parse(string s, NumberStyles styles)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*, с использованием данных о стилях, определяемых параметром *styles*. Если в строке не представлено числовое значение типа `decimal`, то генерируется исключение
`public static decimal Parse(string s, NumberStyles styles, IformatProvider provider)`	Возвращает двоичный эквивалент числа, заданного в виде символьной строки *s*, с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром *provider*, а также данных о стилях, обозначаемых параметром *styles*. Если в строке не представлено числовое значение типа `decimal`, то генерируется исключение
`public static decimal Remainder(decimal d1, decimal d2)`	Возвращает остаток от целочисленного деления *d1* / *d2*
`public static decimal Round(decimal d)`	Возвращает значение d, округленное до ближайшего целого числа
`public static decimal Round(decimal d, int decimals)`	Возвращает значение *d*, округленное до числа с количеством цифр в дробной части, равным значению параметра *decimals*, которое должно находиться в пределах от 0 до 28

Продолжение табл. 21.7

Метод	Назначение
`public static decimal Round(decimal d, MidPointRounding mode)`	Возвращает значение *d*, округленное до ближайшего целого числа в режиме, определяемом параметром *mode*. Режим округления применяется лишь в том случае, если значение *d* оказывается посредине между двумя целыми числами
`public static decimal Round(decimal d, int decimals, MidPointRounding mode)`	Возвращает значение *d*, округленное до числа с количеством цифр в дробной части, равным значению параметра *decimals*, которое должно находиться в пределах от 0 до 28, а параметр *mode* определяет режим округления. Режим округления применяется лишь в том случае, если значение *d* оказывается посредине между двумя округляемыми числами
`public static decimal Subtract(decimal d1, decimal d2)`	Возвращает разность *d1* - *d2*
`public static byte ToByte(decimal value)`	Возвращает эквивалент значения *value* типа `byte`. Дробная часть отбрасывается. Если значение *value* оказывается вне диапазона представления чисел для типа `byte`, то генерируется исключение `OverflowException`
`public static double ToDouble(decimal d)`	Возвращает эквивалент значения *d* типа `double`. При этом возможна потеря точности, поскольку у значения типа `double` меньше значащих цифр, чем у значения типа `decimal`
`public static short ToInt16(decimal d)`	Возвращает эквивалент значения *d* типа `short`. Дробная часть отбрасывается. Если значение *d* оказывается вне диапазона представления чисел для типа `short`, то генерируется исключение `OverflowException`
`public static int ToInt32(decimal d)`	Возвращает эквивалент значения *d* типа `int`. Дробная часть отбрасывается. Если значение *d* оказывается вне диапазона представления чисел для типа `int`, то генерируется исключение `OverflowException`
`public static long ToInt64(decimal d)`	Возвращает эквивалент значения d типа `long`. Дробная часть отбрасывается. Если значение *d* оказывается вне диапазона представления чисел для типа `long`, то генерируется исключение `OverflowException`
`public static long ToOACurrency(decimal value)`	Преобразует значение *value* в его эквивалент формата денежной единицы, применяемого в компоненте OLE Automation, и возвращает полученный результат

Продолжение табл. 21.7

Метод	Назначение
`public static sbyte ToSByte(decimal value)`	Возвращает эквивалент значения `value` типа `sbyte`. Дробная часть отбрасывается. Если значение `value` оказывается вне диапазона представления чисел для типа `sbyte`, то генерируется исключение `OverflowException`
`public static float ToSingle(decimal d)`	Возвращает эквивалент значения `d` типа `float`. Дробная часть отбрасывается. Если значение `d` оказывается вне диапазона представления чисел для типа `float`, то генерируется исключение `OverflowException`
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта в используемом по умолчанию формате
`public string ToString(string format)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`
`public string ToString(IFormatProvider provider)`	Возвращает строковое представление значения вызывающего объекта с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public string ToString(string format, IFormatProvider provider)`	Возвращает строковое представление значения вызывающего объекта, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром `provider`
`public static ushort ToUInt16(decimal value)`	Возвращает эквивалент значения `value` типа `ushort`. Дробная часть отбрасывается. Если значение `value` оказывается вне диапазона представления чисел для типа `ushort`, то генерируется исключение `OverflowException`
`public static uint ToUInt32(decimal d)`	Возвращает эквивалент значения `d` типа `uint`. Дробная часть отбрасывается. Если значение `d` оказывается вне диапазона представления чисел для типа `uint`, то генерируется исключение `OverflowException`
`public static ulong ToUInt64(decimal d)`	Возвращает эквивалент значения `d` типа `ulong`. Дробная часть отбрасывается. Если значение `d` оказывается вне диапазона представления чисел для типа `ulong`, то генерируется исключение `OverflowException`
`public static decimal Truncate(decimal d)`	Возвращает целую часть числа `d`. Дробная часть отбрасывается

Окончание табл. 21.7

Метод	Назначение
`public static bool TryParse(string s, out decimal result)`	Предпринимает попытку преобразовать числовое значение, заданное в виде символьной строки *s*, в значение типа `decimal`. При успешной попытке это значение сохраняется в параметре *result* и возвращается логическое значение `true`. В противном случае возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования
`public static bool TryParse(string s, NumberStyles styles, IFormatProvider provider, out decimal result)`	Предпринимает попытку преобразовать числовое значение, заданное в виде символьной строки *s*, в значение типа `decimal`, как указано в форматирующей строке, определяемой параметром `format`, но с использованием форматов данных, характерных для конкретной культурной среды и определяемых параметром *provider*, а также сведений о стилях, обозначаемых параметром *styles*. При успешной попытке это значение сохраняется в параметре *result* и возвращается логическое значение `true`. В противном случае возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования

Таблица 21.8. Поля, поддерживаемые структурой `Decimal`

Поле	Назначение
`public static readonly decimal MaxValue`	Наибольшее значение, допустимое для данных типа `decimal`
`public static readonly decimal MinusOne`	Представление числа -1 в виде значения типа `decimal`
`public static readonly decimal MinValue`	Наименьшее значение, допустимое для данных типа `decimal`
`public static readonly decimal One`	Представление числа 1 в виде значения типа `decimal`
`public static readonly decimal Zero`	Представление числа 0 в виде значения типа `decimal`

Структура `Char`

Структура `Char` соответствует типу `char` и применяется довольно часто, поскольку предоставляет немало методов, позволяющих обрабатывать символы и распределять их по отдельным категориям. Например, символ строчной буквы можно преобразовать в символ прописной буквы, вызвав метод `ToUpper()`, а с помощью метода `IsDigit()` можно определить, обозначает ли символ цифру.

Методы, определенные в структуре `Char`, приведены в табл. 21.9. Следует, однако, иметь в виду, что некоторые методы, например `ConvertFromUtf32()` и `ConvertToUtf32()`, позволяют обрабатывать символы уникода в форматах UTF-16 и UTF-32. Раньше все символы уникода могли быть представлены 16 разрядами, что соответствует величине значения типа `char`. Но несколько лет назад набор символов уникода был расширен, для чего потребовалось более 16 разрядов. Каждый символ уникода представлен *кодовой точкой*, а способ кодирования кодовой точки зависит от используемого формата преобразования уникода (UTF). Так, в формате UTF-16 для кодирования большинства кодовых точек требуется одно 16-разрядное значение, а для кодирования остальных кодовых точек — два 16-разрядных значения. Если для этой цели требуются два 16-разрядных значения, то для их представления служат два значения типа `char`. Первое символьное значение называется *старшим суррогатом*, а второе — *младшим суррогатом*. В формате UTF-32 каждая кодовая точка кодируется с помощью одного 32-разрядного значения. В структуре `Char` предоставляются все необходимые средства для преобразования из формата UTF-16 в формат UTF-32 и обратно.

В отношении методов структуры `Char` необходимо также отметить следующее: в используемых по умолчанию формах методов `ToUpper()` и `ToLower()` применяются текущие настройки культурной среды (языки и региональные стандарты), чтобы указать способ представления символов верхнего и нижнего регистра. На момент написания этой книги рекомендовалось явно указывать текущие настройки культурной среды, используя для этой цели параметр типа `CultureInfo` во второй форме обоих упоминаемых методов. Класс `CultureInfo` относится к пространству имен `System.Globalization`, а для указания текущей культурной среды следует передать свойство `CultureInfo.CurrentCulture` соответствующему методу.

В структуре `Char` определены также следующие поля.

```
public const char MaxValue
public const char MinValue
```

Кроме того, в структуре `Char` реализуются следующие интерфейсы: `IComparable`, `IComparable<char>`, `IConvertible` и `IEquatable<char>`.

Таблица 21.9. Методы, определенные в структуре `Char`

Метод	Назначение
`public int CompareTo(char value)`	Сравнивает символ в вызывающем объекте с символом *value*. Возвращает нуль, если сравниваемые символы равны; отрицательное значение, если вызывающий объект имеет меньшее значение; и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public int CompareTo(object value)`	Сравнивает символ в вызывающем объекте с символом *value*. Возвращает нуль, если сравниваемые символы равны; отрицательное значение, если вызывающий объект имеет меньшее значение; и, наконец, положительное значение, если вызывающий объект имеет большее значение
`public static string ConvertFromUtf32(int utf32)`	Преобразует кодовую точку уникода, представленную параметром *utf32* в формате UTF-32, в символьную строку формата UTF-16 и возвращает полученный результат

Продолжение табл. 21.9

Метод	Назначение
`pubic static int ConvertToUtf32(char highSurrogate, char lowSurrogate)`	Преобразует старший и младший суррогаты, представленные параметрами `highSurrogate` и `lowSurrogate` в формате UTF-16, в кодовую точку формата UTF-32 и возвращает полученный результат
`pubic static int ConvertToUtf32(string s, int index)`	Преобразует пару суррогатов формата UTF-16, доступных из символьной строки по индексу `s[index]`, в кодовую точку формата UTF-32 и возвращает полученный результат
`public bool Equals(char obj)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению `obj`
`public override bool Equals(object obj)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению `obj`
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public static double GetNumericValue(char c)`	Возвращает числовое значение символа `c`, если он обозначает цифру. В противном случае возвращает `-1`
`public static double GetNumericValue(string s, int index)`	Возвращает числовое значение символа, доступного из строки по индексу `s[index]`, если он обозначает цифру. В противном случае возвращает `-1`
`public TypeCode GetTypeCode()`	Возвращает значение из перечисления `TypeCode` для структуры `Char`, т.е. `TypeCode.Char`
`public static UnicodeCategory GetUnicodeCategory(char c)`	Возвращает значение из перечисления `UnicodeCategory` для символа `c`. Перечисление `UnicodeCategory` определено в пространстве имен `System.Globalization` и распределяет символы уникода по категориям
`public static UnicodeCategory GetUnicodeCategory(string s, int index)`	Возвращает значение из перечисления `UnicodeCategory` для символа, доступного из строки по индексу `s[index]`. Перечисление `UnicodeCategory` определено в пространстве имен `System.Globalization` и распределяет символы уникода по категориям
`public static bool IsControl(char c)`	Возвращает логическое значение `true`, если символ `c` является управляющим, иначе возвращает логическое значение `false`
`public static bool IsControl(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, является управляющим, иначе возвращает логическое значение `false`
`public static bool IsDigit(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает цифру, а иначе возвращает логическое значение `false`

Продолжение табл. 21.9

Метод	Назначение
`public static bool IsDigit(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает цифру, а иначе возвращает логическое значение `false`
`public static bool IsHighSurrogate(char c)`	Возвращает логическое значение `true`, если символьное значение `c` является действительным старшим суррогатом формата UTF-32, а иначе возвращает логическое значение `false`
`public static bool IsHighSurrogate(string s, int index)`	Возвращает логическое значение `true`, если символьное значение, доступное из строки по индексу `s[index]`, является действительным старшим суррогатом формата UTF-32, а иначе возвращает логическое значение `false`
`public static bool IsLetter(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает букву алфавита, а иначе возвращает логическое значение `false`
`public static bool IsLetter(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает букву алфавита, а иначе возвращает логическое значение `false`
`public static bool IsLetterOrDigit(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает букву алфавита или цифру, а иначе возвращает логическое значение `false`
`public static bool IsLetterOrDigit(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает букву алфавита или цифру, а иначе возвращает логическое значение `false`
`public static bool IsLower(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает строчную букву алфавита, а иначе возвращает логическое значение `false`
`public static bool IsLower(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает строчную букву алфавита, а иначе возвращает логическое значение `false`
`public static bool IsLowSurrogate(char c)`	Возвращает логическое значение `true`, если символьное значение `c` является действительным младшим суррогатом формата UTF-32, а иначе возвращает логическое значение `false`
`public static bool IsLowSurrogate(string s, int index)`	Возвращает логическое значение `true`, если символьное значение, доступное из строки по индексу `s[index]`, является действительным младшим суррогатом формата UTF-32, а иначе возвращает логическое значение `false`
`public static bool IsNumber(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает число (десятичное или шестнадцатеричное), а иначе возвращает логическое значение `false`

Продолжение табл. 21.9

Метод	Назначение
`public static bool IsNumber(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает число (десятичное или шестнадцатеричное), а иначе возвращает логическое значение `false`
`public static bool IsPunctuation(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает знак препинания, а иначе возвращает логическое значение `false`
`public static bool IsPunctuation(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает знак препинания, а иначе возвращает логическое значение `false`
`public static bool IsSeparator(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает разделительный знак, а иначе возвращает логическое значение `false`
`public static bool IsSeparator(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает разделительный знак, а иначе возвращает логическое значение `false`
`public static bool IsSurrogate(char c)`	Возвращает логическое значение `true`, если символьное значение `c` является суррогатным символом уникода, а иначе возвращает логическое значение `false`
`public static bool IsSurrogate(string s, int index)`	Возвращает логическое значение `true`, если символьное значение, доступное из строки по индексу `s[index]`, является суррогатным символом уникода, а иначе возвращает логическое значение `false`
`public static bool IsSurrogatePair(char highSurrogate, char lowSurrogate)`	Возвращает логическое значение `true`, если символьные значения `highSurrogate` и `lowSurrogate` образуют суррогатную пару
`public static bool IsSymbol(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает символический знак, например денежной единицы, а иначе возвращает логическое значение `false`
`public static bool IsSymbol(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает символический знак, например денежной единицы, а иначе возвращает логическое значение `false`
`public static bool IsUpper(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает прописную букву алфавита, а иначе возвращает логическое значение `false`
`public static bool IsUpper(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает прописную букву алфавита, а иначе возвращает логическое значение `false`

Продолжение табл. 21.9

Метод	Назначение
`public static bool IsWhiteSpace(char c)`	Возвращает логическое значение `true`, если символ `c` обозначает пробел, табуляцию или пустую строку, а иначе возвращает логическое значение `false`
`public static bool IsWhiteSpace(string s, int index)`	Возвращает логическое значение `true`, если символ, доступный из строки по индексу `s[index]`, обозначает пробел, табуляцию или пустую строку, а иначе возвращает логическое значение `false`
`public static char Parse(string s)`	Возвращает эквивалент типа `char` символа из строки `s`. Если строка `s` состоит из нескольких символов, то генерируется исключение `FormatException`
`public static char ToLower(char c)`	Возвращает строчный эквивалент символа `c`, если он обозначает прописную букву. В противном случае значение символа `c` не изменяется
`public static char ToLower(char c, CultureInfo culture)`	Возвращает строчный эквивалент символа `c`, если он обозначает прописную букву. В противном случае значение символа `c` не изменяется. Преобразование выполняется в соответствии с информацией о культурной среде, указываемой в параметре `culture`, где `CultureInfo` — это класс, определенный в пространстве имен `System.Globalization`
`public static char ToLowerInvariant(char c)`	Возвращает строчный эквивалент символа `c` независимо от настроек культурной среды
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта типа `Char`
`public static string ToString(char c)`	Возвращает строковое представление символьного значения `c`
`public string ToString(IFormatProvider provider)`	Возвращает строковое представление значения вызывающего объекта типа `Char` с учетом информации о культурной среде, указываемой в параметре `provider`
`public static char ToUpper(char c)`	Возвращает прописной эквивалент символа `c`, если он обозначает строчную букву. В противном случае значение символа `c` не изменяется
`public static char ToUpper(char c, CultureInfo culture)`	Возвращает прописной эквивалент символа `c`, если он обозначает строчную букву. В противном случае значение символа `c` не изменяется. Преобразование выполняется в соответствии с информацией о культурной среде, указываемой в параметре `culture`, где `CultureInfo` — это класс, определенный в пространстве имен `System.Globalization`

Окончание табл. 21.9

Метод	Назначение
`public static char ToUpperInvariant(char c)`	Возвращает прописной эквивалент символа `c` независимо от настроек культурной среды
`public static bool TryParse(string s, out char result)`	Предпринимает попытку преобразовать символ из строки `s` в его эквивалентное значение типа `char`. При успешной попытке это значение сохраняется в параметре `result` и возвращается логическое значение `true`. Если же строка `s` состоит из нескольких символов, то возвращается логическое значение `false`, в отличие от метода `Parse()`, который генерирует исключение при неудачном исходе преобразования

Ниже приведен пример программы, в которой демонстрируется применение нескольких методов, определенных в структуре `Char`.

```
// Продемонстрировать применение нескольких методов,
// определенных в структуре Char.

using System;
using System.Globalization;

class CharDemo {
  static void Main() {
    string str = "Это простой тест. $23";
    int i;
    for(i=0; i < str.Length; i++) {
      Console.Write(str[i] + " является");
      if(Char.IsDigit(str[i]))
        Console.Write(" цифрой");
      if(Char.IsLetter(str[i]))
        Console.Write(" буквой");
      if(Char.IsLower(str[i]))
        Console.Write(" строчной");
       if(Char.IsUpper(str[i]))
         Console.Write(" прописной");
       if(Char.IsSymbol(str[i]))
         Console.Write(" символическим знаком");
       if(Char.IsSeparator(str[i]))
         Console.Write(" разделительным");
       if(Char.IsWhiteSpace(str[i]))
         Console.Write(" пробелом");
       if(Char.IsPunctuation(str[i]))
         Console.Write(" знаком препинания");

       Console.WriteLine();
    }
```

```
    Console.WriteLine("Исходная строка: " + str);

    // Преобразовать в прописные буквы.
    string newstr = "";
    for(i=0; i < str.Length; i++)
      newstr += Char.ToUpper(str[i], CultureInfo.CurrentCulture);

    Console.WriteLine("После преобразования: " + newstr);
  }
}
```

Эта программа дает следующий результат.

```
Э является буквой прописной
т является буквой строчной
о является буквой строчной
  является разделительным пробелом
п является буквой строчной
р является буквой строчной
о является буквой строчной
с является буквой строчной
т является буквой строчной
о является буквой строчной
й является буквой строчной
  является разделительным пробелом
т является буквой строчной
е является буквой строчной
с является буквой строчной
т является буквой строчной
. является знаком препинания
  является разделительным пробелом
$ является символическим знаком
2 является цифрой
3 является цифрой
Исходная строка: Это простой тест. $23
После преобразования: ЭТО ПРОСТОЙ ТЕСТ. $23
```

Структура `Boolean`

В структуре `Boolean` поддерживаются данные типа `bool`. Методы, определенные в этой структуре, перечислены в табл. 21.10. Кроме того, в ней определены следующие поля.

```
public static readonly string FalseString
public static readonly string TrueString
```

В этих полях логические значения `true` и `false` содержатся в удобочитаемой форме. Так, если вывести содержимое поля `FalseString` с помощью метода `WriteLine()`, то на экране появится строка `"False"`.

В структуре `Boolean` реализованы следующие интерфейсы: `IComparable`, `IComparable<bool>`, `IConvertible` и `IEquatable<bool>`.

Таблица 21.10. Методы, определенные в структуре `Boolean`

Метод	Назначение
`public int CompareTo(bool value)`	Сравнивает логическое значение вызывающего объекта со значением параметра *value*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если вызывающий объект имеет логическое значение `false`, а параметр *value* — логическое значение `true`; и, наконец, положительное значение, если вызывающий объект имеет логическое значение `true`, а параметр *value* — логическое значение `false`
`public int CompareTo(object obj)`	Сравнивает логическое значение вызывающего объекта со значением параметра *obj*. Возвращает нуль, если сравниваемые значения равны; отрицательное значение, если вызывающий объект имеет логическое значение `false`, а параметр *obj* — логическое значение `true`; и, наконец, положительное значение, если вызывающий объект имеет логическое значение `true`, а параметр *obj* — логическое значение `false`
`public bool Equals(bool obj)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению параметра *obj*
`public override bool Equals(object obj)`	Возвращает логическое значение `true`, если значение вызывающего объекта равно значению параметра *obj*
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public TypeCode GetTypeCode()`	Возвращает значение перечисления `TypeCode` для структуры `Boolean`, т.е. `TypeCode.Boolean`
`public static bool Parse(string s)`	Возвращает эквивалент типа `bool` символьной строки *s*. Если строка *s* не содержит ни поле `Boolean.TrueString`, ни поле `Boolean.FalseString`, то генерируется исключение `FormatException`, независимо от того, какими буквами набрано содержимое строки: прописными или строчными
`public override string ToString()`	Возвращает строковое представление значения вызывающего объекта, которое должно быть либо значением поля `TrueString`, либо значением поля `FalseString`
`public string ToString(IFormatProvider provider)`	Возвращает строковое представление значения вызывающего объекта, которое должно быть либо значением поля `TrueString`, либо значением поля `FalseString`. При этом параметр *provider* игнорируется
`public static bool TryParse(string s, out bool result)`	Предпринимает попытку преобразовать символ из строки *s* в его эквивалентное значение типа `bool`. При успешной попытке это значение сохраняется в параметре *result* и возвращается логическое значение `true`. Если же строка *s* не содержит ни поле `Boolean.TrueString`, ни поле `Boolean.FalseString`, то возвращается логическое значение `false`, независимо от того, какими буквами набрано содержимое строки: прописными или строчными, в отличие от метода `Parse()`, который генерирует исключение в аналогичной ситуации

Класс Array

Класс Array относится к числу наиболее часто используемых в пространстве имен System. Он является базовым классом для всех массивов в C#. Следовательно, его методы можно применять к массивам любого встроенного в C# типа или же к массивам определяемого пользователем типа. Свойства, определенные в классе Array, перечислены в табл. 21.11, а методы — в табл. 21.12.

В классе Array реализуются следующие интерфейсы: ICloneable, ICollection, IEnumerable, IStructuralComparable, IStructuralEquatable, а также IList. Все интерфейсы, кроме ICloneable, определены в пространстве имен System.Collections, подробнее рассматриваемом в главе 25.

В ряде методов данного класса используется параметр типа IComparer или IComparer<T>. Интерфейс IComparer находится в пространстве имен System.Collections. В нем определяется метод Compare() для сравнения значений двух объектов, как показано ниже.

```
int Compare(object x, object y)
```

Этот метод возвращает значение больше нуля, если *x* больше *y*; значение меньше нуля, если *x* меньше *y*; и, наконец, нулевое значение, если оба значения равны.

Интерфейс IComparer<T> находится в пространстве имен System.Collections.Generic. В нем определяется метод Compare(), общая форма которого приведена ниже.

```
int Compare(T x, T y)
```

Он действует таким же образом, как и его необобщенный аналог, возвращая значение больше нуля, если *x* больше *y*; значение меньше нуля, если *x* меньше *y*; и, наконец, нулевое значение, если оба значения равны. Преимущество интерфейса IComparer<T> заключается в том, что он обеспечивает типовую безопасность. Ведь в этом случае тип обрабатываемых данных указывается явным образом, а следовательно, никакого приведения типов не требуется.

В последующих разделах демонстрируется ряд наиболее распространенных операций с массивами.

Таблица 21.11. Свойства, определенные в классе Array

Свойство	Назначение
`public bool IsFixedSize { get; }`	Доступно только для чтения. Принимает логическое значение `true`, если массив имеет фиксированный размер, и логическое значение `false`, если массив может изменять его динамически
`public bool IsReadOnly { get; }`	Доступно только для чтения. Принимает логическое значение `true`, если объект класса Array предназначен только для чтения, а иначе — логическое значение `false`. Для массивов это свойство всегда имеет логическое значение `true`
`public bool IsSynchronized { get; }`	Доступно только для чтения. Принимает логическое значение `true`, если массив можно безопасно использовать в многопоточной среде, а иначе — логическое значение `false`. Для массивов это свойство всегда имеет логическое значение `true`

Окончание табл. 21.11

Свойство	Назначение
`public int Length { get; }`	Доступно только для чтения. Имеет тип `int` и содержит количество элементов в массиве
`public long LongLength { get; }`	Доступно только для чтения. Имеет тип `long` и содержит количество элементов в массиве
`public int Rank { get; }`	Доступно только для чтения. Содержит размерность массива
`public object SyncRoot { get; }`	Доступно только для чтения. Содержит объект, предназначенный для синхронизации доступа к массиву

Таблица 21.12. Методы, определенные в классе `Array`

Метод	Назначение
`public static ReadOnlyCollection<T> AsReadOnly<T>(T[] array)`	Возвращает доступную только для чтения коллекцию, которая включает в себя массив, определяемый параметром *array*
`public static int BinarySearch(Array array, object value)`	Осуществляет поиск значения *value* в массиве *array*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив *array* должен быть отсортированным и одномерным
`public static int BinarySearch<T>(T[] array, T value)`	Осуществляет поиск значения *value* в массиве *array*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив *array* должен быть отсортированным и одномерным
`public static int BinarySearch(Array array, object value, IComparer comparer)`	Осуществляет поиск значения *value* в массиве, определяемом параметром *array*, используя способ сравнения, задаваемый параметром *comparer*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив *array* должен быть отсортированным и одномерным
`public static int BinarySearch<T>(T[] array, T value, IComparer<T> comparer)`	Осуществляет поиск значения *value* в массиве *array*, используя способ сравнения, задаваемый параметром *comparer*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив *array* должен быть отсортированным и одномерным
`public static int BinarySearch(Array array, int index, int length, object value)`	Осуществляет поиск значения *value* в части массива *array*. Поиск начинается с индекса, задаваемого параметром *index*, и охватывает число элементов, определяемых параметром *length*. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив *array* должен быть отсортированным и одномерным

Продолжение табл. 21.12

Метод	Назначение
`public static int BinarySearch<T>(T[] array, int index, int length, T value)`	Осуществляет поиск значения `value` в части массива `array`. Поиск начинается с индекса, задаваемого параметром `index`, и охватывает число элементов, определяемых параметром `length`. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив `array` должен быть отсортированным и одномерным
`public static int BinarySearch(Array array, int index, int length, object value, IComparer comparer)`	Осуществляет поиск значения `value` в части массива `array`, используя способ сравнения, определяемый параметром `comparer`. Поиск начинается с индекса, задаваемого параметром `index`, и охватывает число элементов, определяемых параметром `length`. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив `array` должен быть отсортированным и одномерным
`public static int BinarySearch<T>(T [] array, int index, int length, T value, Icomparer<T> comparer)`	Осуществляет поиск значения `value` в части массива `array`, используя способ сравнения, определяемый параметром `comparer`. Поиск начинается с индекса, задаваемого параметром `index`, и охватывает число элементов, определяемых параметром `length`. Возвращает индекс первого вхождения искомого значения. Если оно не найдено, возвращает отрицательное значение. Массив `array` должен быть отсортированным и одномерным
`public static void Clear(Array array, int index, int length)`	Устанавливает заданные элементы массива `array` равными нулю, пустому значению `null` или логическому значению `false` в зависимости от типа элемента: значения, ссылочного или логического. Подмножество элементов, подлежащих обнулению, начинается с индекса, задаваемого параметром `index`, и включает в себя число элементов, определяемых параметром `length`
`public object Clone()`	Возвращает копию вызывающего массива. Эта копия ссылается на те же элементы, что и оригинал, поэтому она называется "неполной". Таким образом, изменения, вносимые в элементы, влияют на оба массива, поскольку и в том и в другом используются одни и те же элементы
`public static void ConstrainedCopy(Array sourceArray, int sourceIndex, Array destinationArray, int destinationIndex, int length)`	Копирует число элементов, задаваемых параметром `length`, из исходного массива `sourceArray`, начиная с элемента, указываемого по индексу `sourceIndex`, в целевой массив `destinationArray`, начиная с элемента,

Продолжение табл. 21.12

Метод	Назначение
	указываемого по индексу *destinationIndex*. Если оба массива имеют одинаковый ссылочный тип, то метод ConstrainedCopy() создает "неполную копию", в результате чего оба массива будут ссылаться на одни и те же элементы. Если же во время копирования возникает ошибка, то содержимое целевого массива *destinationArray* остается прежним
public static TTo[] ConvertAll<TInput, TTo>(TFrom[] *array*, Converter<TOutput, TTo> *converter*)	Преобразует массив *array* из типа TInput в тип TOutput и возвращает получающийся в итоге массив. Исходный массив остается прежним. Преобразование выполняется преобразователем, задаваемым параметром *converter*
public static void Copy(Array *sourceArray*, Array *destinationArray*, int *length*)	Копирует число элементов, задаваемых параметром length, из исходного массива *sourceArray* в целевой массив *destinationArray*, начиная с первого элемента массива. Если оба массива имеют одинаковый ссылочный тип, то метод Copy() создает "неполную копию", в результате чего оба массива будут ссылаться на одни и те же элементы. Если же во время копирования возникает ошибка, то содержимое целевого массива *destinationArray* оказывается неопределенным
public static void Copy(Array *sourceArray*, Array *destinationArray*, long *length*)	Копирует число элементов, задаваемых параметром *length*, из исходного массива *sourceArray* в целевой массив *destinationArray*, начиная с первого элемента массива. Если оба массива имеют одинаковый ссылочный тип, то метод Copy() создает "неполную копию", в результате чего оба массива будут ссылаться на одни и те же элементы. Если же во время копирования возникает ошибка, то содержимое целевого массива *destinationArray* оказывается неопределенным
public static void Copy(Array *sourceArray*, int *sourceIndex*, Array *destinationArray*, int *destinationIndex*, int *length*)	Копирует число элементов, задаваемых параметром *length*, из исходного массива *sourceArray*, начиная с элемента, указываемого по индексу *sourceArray*[*sourceIndex*], в целевой массив *destinationArray*, начиная с элемента, указываемого по индексу *destinationArray*[*destinationIndex*]. Если оба массива имеют одинаковый ссылочный тип, то метод Copy() создает "неполную копию", в результате чего оба массива будут ссылаться на одни и те же элементы. Если же во время копирования возникает ошибка, то содержимое целевого массива *destinationArray* оказывается неопределенным

Продолжение табл. 21.12

Метод	Назначение
`public static void Copy(Array sourceArray, long sourceIndex, Array destinationArray, long destinationIndex, long length)`	Копирует число элементов, задаваемых параметром *length*, из исходного массива *sourceArray*, начиная с элемента, указываемого по индексу *sourceArray*[*sourceIndex*], в целевой массив *destinationArray*, начиная с элемента, указываемого по индексу *destinationArray*[*destinationIndex*]. Если оба массива имеют одинаковый ссылочный тип, то метод `Copy()` создает "неполную копию", в результате чего оба массива будут ссылаться на одни и те же элементы. Если же во время копирования возникает ошибка, то содержимое целевого массива `destinationArray` оказывается неопределенным
`public void CopyTo(Array array, int index)`	Копирует элементы вызывающего массива в целевой массив *array*, начиная с элемента, указываемого по индексу *array*[*index*]. Если же во время копирования возникает ошибка, то содержимое целевого массива `array` оказывается неопределенным
`public void CopyTo(Array array, long index)`	Копирует элементы вызывающего массива в целевой массив *array*, начиная с элемента, указываемого по индексу *array*[*index*]. Если же во время копирования возникает ошибка, то содержимое целевого массива `array` оказывается неопределенным
`public static Array CreateInstance(Type elementType, int length)`	Возвращает ссылку на одномерный массив, который содержит число элементов типа *elementType*, определяемое параметром *length*
`public static Array CreateInstance(Type elementType, int length1, int length2)`	Возвращает ссылку на двумерный массив размером *length1*×*length2*. Каждый элемент этого массива имеет тип *elementType*
`public static Array CreateInstance(Type elementType, int length1, int length2, int length3)`	Возвращает ссылку на трехмерный массив размером *length1*×*length2*×*length3*. Каждый элемент этого массива имеет тип *elementType*
`public static Array CreateInstance(Type elementType, params int[] lengths)`	Возвращает ссылку на многомерный массив, размерность которого задается в массиве *lengths*. Каждый элемент этого массива имеет тип *elementType*
`public static Array CreateInstance(Type elementType, params long[] lengths)`	Возвращает ссылку на многомерный массив, размерность которого задается в массиве *lengths*. Каждый элемент этого массива имеет тип *elementType*

Продолжение табл. 21.12

Метод	Назначение
`public static Array CreateInstance(Type elementType, int[] lengths, int[] lowerBounds)`	Возвращает ссылку на многомерный массив, размерность которого задается в массиве `lengths`. Каждый элемент этого массива имеет тип `elementType`. Начальный индекс каждого измерения задается в массиве `lowerBounds`. Таким образом, этот метод позволяет создавать массивы, которые начинаются с некоторого индекса, отличного от нуля
`public static bool Exists<T>(T[] array, Predicate<T> match)`	Возвращает логическое значение `true`, если массив `array` содержит хотя бы один элемент, удовлетворяющий условию предиката, задаваемого параметром `match`, а иначе возвращает логическое значение `false`
`public static T Find<T>(T[] array, Predicate<T> match)`	Возвращает первый элемент массива `array`, удовлетворяющий условию предиката, задаваемого параметром `match`, а иначе возвращает значение типа `default(T)`
`public static T[] FindAll<T>(T[] array, Predicate<T> match)`	Возвращает все элементы массива `array`, удовлетворяющие условию предиката, задаваемого параметром `match`, а иначе возвращает массив нулевой длины
`public static int FindIndex<T>(T[] array, Predicate<T> match)`	Возвращает индекс первого элемента массива `array`, удовлетворяющего условию предиката, задаваемого параметром `match`, иначе возвращает значение `-1`
`public static int FindIndex<T>(T[]array, int startIndex, Predicate<T> match)`	Возвращает индекс первого элемента массива `array`, удовлетворяющего условию предиката, задаваемого параметром `match`. Поиск начинается с элемента, указываемого по индексу `array[startIndex]`. Если ни один из элементов, удовлетворяющих данному условию, не найден, то возвращается значение `-1`
`public static int FindIndex<T>(T[] array, int startIndex, int count, Predicate<T> match)`	Возвращает индекс первого элемента массива `array`, удовлетворяющего условию предиката, задаваемого параметром `match`. Поиск начинается с элемента, указываемого по индексу `array[startIndex]`, и продолжается среди числа элементов, определяемых параметром `count`. Если ни один из элементов, удовлетворяющих данному условию, не найден, то возвращается значение `-1`
`public static T FindLast<T>(T[] array, Predicate<T> match)`	Возвращает последний элемент массива `array`, удовлетворяющий условию предиката, задаваемого параметром `match`, иначе возвращает значение типа `default(T)`

Продолжение табл. 21.12

Метод	Назначение
`public static int FindLastIndex<T>(T[] array, Predicate<T> match)`	Возвращает индекс последнего элемента массива `array`, удовлетворяющего условию предиката, задаваемого параметром `match`, иначе возвращает значение `-1`
`public static int FindLastIndex<T>(T[] array, int startIndex, Predicate<T> match)`	Возвращает индекс последнего элемента массива `array`, удовлетворяющего условию предиката, задаваемого параметром `match`. Поиск начинается в обратном порядке с элемента, указываемого по индексу `array[startIndex]`, и оканчивается на элементе `array[0]`. Если ни один из элементов, удовлетворяющих данному условию, не найден, то возвращается значение `-1`
`public static int FindLastIndex<T>(T[] array, int startIndex, int count, Predicate<T> match)`	Возвращает индекс последнего элемента массива `array`, удовлетворяющего условию предиката, задаваемого параметром `v`. Поиск начинается в обратном порядке с элемента, указываемого по индексу `array[start]`, и продолжается среди числа элементов, определяемых параметром `count`. Если ни один из элементов, удовлетворяющих данному условию, не найден, то возвращается значение `-1`
`public static void ForEach<T>(T[]array, Action<T> action)`	Применяет метод, задаваемый параметром `action`, к каждому элементу массива `array`
`public IEnumerator GetEnumerator()`	Возвращает перечислительный объект для массива. Перечислители позволяют опрашивать массив в цикле. Боле подробно перечислители описываются в главе 25
`public override int GetHashCode()`	Возвращает хеш-код для вызывающего объекта
`public int GetLength(int dimension)`	Возвращает длину заданного измерения массива. Отсчет измерений начинается с нуля, поэтому для получения длины первого измерения необходимо передать данному методу значение 0 параметра `dimension`, для получения длины второго измерения — значение 1 и т.д.
`public long GetLongLength(int dimension)`	Возвращает длину заданного измерения массива в виде значения типа `long`. Отсчет измерений начинается с нуля, поэтому для получения длины первого измерения необходимо передать данному методу значение 0 параметра `dimension`, для получения длины второго измерения — значение 1 и т.д.
`public int GetLowerBound(int dimension)`	Возвращает начальный индекс заданного измерения массива, который обычно равен нулю. Параметр `dimension` определяет отсчет измерений

Продолжение табл. 21.12

Метод	Назначение
	с нуля, поэтому для получения начального индекса первого измерения необходимо передать данному методу значение 0 параметра *dimension*, для получения начального индекса второго измерения — значение 1 и т.д.
`public int GetUpperBound(int dimension)`	Возвращает конечный индекс заданного измерения массива. Параметр *dimension* определяет отсчет измерений с нуля, поэтому для получения конечного индекса первого измерения необходимо передать данному методу значение 0 параметра *dimension*, для получения конечного индекса второго измерения — значение 1 и т.д.
`public object GetValue(int index)`	Возвращает значение элемента из вызывающего массива по индексу *index*. Массив должен быть одномерным
`public object GetValue(long index)`	Возвращает значение элемента из вызывающего массива по индексу *index*. Массив должен быть одномерным
`public object GetValue(int index1, int index2)`	Возвращает значение элемента из вызывающего массива по индексам [*index1, index2*]. Массив должен быть двумерным
`public object GetValue(long index1, long index2)`	Возвращает значение элемента из вызывающего массива по индексам [*index1, index2*]. Массив должен быть двумерным
`public object GetValue(int index1, int index2, int index3)`	Возвращает значение элемента из вызывающего массива по индексам [*index1, index2, index3*]. Массив должен быть трехмерным
`public object GetValue(long index1, long index2, long idx3)`	Возвращает значение элемента из вызывающего массива по индексам [*index1, index2, index3*]. Массив должен быть трехмерным
`public object GetValue(int[] indices)`	Возвращает значение элемента из вызывающего массива по указанным индексам. Число измерений массива должно соответствовать числу элементов массива *indices*
`public object GetValue(long[] indices)`	Возвращает значение элемента из вызывающего массива по указанным индексам. Число измерений массива должно соответствовать числу элементов массива *indices*
`public static int IndexOf(Array array, object value)`	Возвращает индекс первого элемента, имеющего значение *value* в одномерном массиве *array*. Если искомое значение не найдено, то возвращает -1. (Если же массив имеет ненулевую нижнюю границу, то неудачный исход поиска будет обозначаться значением нижней границы, уменьшенным на 1.)

Продолжение табл. 21.12

Метод	Назначение
`public static int IndexOf<T>(T[] array, T value)`	Возвращает индекс первого элемента, имеющего значение `value` в одномерном массиве `array`. Если искомое значение не найдено, то возвращает -1
`public static int IndexOf(Array array, object value, int startIndex)`	Возвращает индекс первого элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается с элемента, указываемого по индексу `array[startIndex]`. Метод возвращает -1, если искомое значение не найдено. (Если массив имеет ненулевую нижнюю границу, то неудачный исход поиска будет обозначаться значением нижней границы, уменьшенным на 1.)
`public static int IndexOf<T>(T[] array, T value, int startIndex)`	Возвращает индекс первого элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается с элемента, указываемого по индексу `array[startIndex]`. Метод возвращает -1, если искомое значение не найдено
`public static int IndexOf(Array array, object value, int startIndex, int count)`	Возвращает индекс первого элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается с элемента, указываемого по индексу `array[startIndex]`, и продолжается среди числа элементов, определяемых параметром `count`. Метод возвращает -1, если искомое значение не найдено в заданных пределах. (Если же массив имеет ненулевую нижнюю границу, то неудачный исход поиска будет обозначаться значением нижней границы, уменьшенным на 1.)
`public static int IndexOf<T>(T[] array, T value, int startIndex, int count)`	Возвращает индекс первого элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается с элемента, указываемого по индексу `array[startIndex]`, и продолжается среди числа элементов, определяемых параметром `count`. Метод возвращает -1, если искомое значение не найдено в заданных пределах
`public void Initialize()`	Инициализирует каждый элемент вызывающего массива с помощью конструктора, используемого по умолчанию для соответствующего элемента. Этот метод можно использовать только для массивов простых типов значений
`public static int LastIndexOf(Array array, object value)`	Возвращает индекс последнего элемента, имеющего значение `value` в одномерном массиве `array`. Если искомое значение не найдено, то возвращает -1. (Если массив имеет ненулевую нижнюю границу, то неудачный исход поиска будет обозначаться значением нижней границы, уменьшенным на 1.)

Продолжение табл. 21.12

Метод	Назначение
`public static int LastIndexOf<T>(T[]array, T value)`	Возвращает индекс последнего элемента, имеющего значение `value` в одномерном массиве `array`. Если искомое значение не найдено, то возвращает -1
`public static int LastIndexOf(Array array, object value, int startIndex)`	Возвращает индекс последнего элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается в обратном порядке с элемента, указываемого по индексу `array[startIndex]`, и оканчивается на элементе `a[0]`. Метод возвращает -1, если искомое значение не найдено. (Если массив имеет ненулевую нижнюю границу, то неудачный исход поиска будет обозначаться значением нижней границы, уменьшенным на 1.)
`public static int LastIndexOf<T>(T[] array, T value, int startIndex)`	Возвращает индекс последнего элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается в обратном порядке с элемента, указываемого по индексу `a[startIndex]`, и оканчивается на элементе `a[0]`. Метод возвращает -1, если искомое значение не найдено
`public static int LastIndexOf(Array array, object value, int startIndex, int count)`	Возвращает индекс последнего элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается в обратном порядке с элемента, указываемого по индексу `array[startIndex]`, и продолжается среди числа элементов, определяемых параметром `count`. Метод возвращает -1, если искомое значение не найдено в заданных пределах. (Если массив имеет ненулевую нижнюю границу, то неудачный исход поиска будет обозначаться значением нижней границы, уменьшенным на 1.)
`public static int LastIndexOf<T>(T[]array, T value, int startIndex, int count)`	Возвращает индекс последнего элемента, имеющего значение `value` в одномерном массиве `array`. Поиск начинается в обратном порядке с элемента, указываемого по индексу `array[startIndex]`, и продолжается среди числа элементов, определяемых параметром `count`. Метод возвращает -1, если искомое значение не найдено в заданных пределах
`public static void Resize<T>(ref T[] array, int newSize)`	Задает длину `newSize` массива `array`
`public static void Reverse(Array array)`	Меняет на обратный порядок следования элементов в массиве `array`
`public static void Reverse(Array array, int index, int length)`	Меняет на обратный порядок следования элементов массива `array` заданных в пределах, начиная с элемента, указываемого по индексу `array[index]`, и включая число элементов, определяемых параметром `length`

Продолжение табл. 21.12

Метод	Назначение
`public void SetValue(object value, int index)`	Устанавливает значение `value` элемента вызывающего массива по индексу `index`. Массив должен быть одномерным
`public void SetValue(object value, long index)`	Устанавливает значение `value` элемента вызывающего массива по индексу `index`. Массив должен быть одномерным
`public void SetValue(object value, int index1, int index2)`	Устанавливает значение `value` элемента вызывающего массива по индексам `[index1, index2]`. Массив должен быть двумерным
`public void SetValue(object value, long index1, long index2)`	Устанавливает значение `value` элемента вызывающего массива по индексам `[index1, index2]`. Массив должен быть двумерным
`public void SetValue(object value, int index1, int index2, int index3)`	Устанавливает значение `value` элемента вызывающего массива по индексам `[index1, index2, index3]`. Массив должен быть трехмерным
`public void SetValue(object value, long index1, long index2, long index3)`	Устанавливает значение `value` элемента вызывающего массива по индексам `[index1, index2, index3]`. Массив должен быть трехмерным
`public void SetValue(object value, int[] indices)`	Устанавливает значение `value` элемента вызывающего массива по указанным индексам. Число измерений массива должно соответствовать числу элементов массива `indices`
`public void SetValue(object value, long[]indices)`	Устанавливает значение `value` элемента вызывающего массива по указанным индексам. Число измерений массива должно соответствовать числу элементов массива `indices`
`public static void Sort(Array array)`	Сортирует массив `array` по нарастающей. Массив должен быть одномерным
`public static void Sort<T>(T[]array)`	Сортирует массив `array` по нарастающей. Массив должен быть одномерным
`public static void Sort(Array array, IComparer comparer)`	Сортирует массив `array` по нарастающей, используя способ сравнения, задаваемый параметром `comparer`. Массив должен быть одномерным
`public static void Sort<T>(T[] array, Comparison<T> comparer)`	Сортирует массив `array` по нарастающей, используя способ сравнения, задаваемый параметром `comparer`. Массив должен быть одномерным
`public static void Sort<T>(T[] array, IComparer<T> comparer)`	Сортирует массив `array` по нарастающей, используя способ сравнения, задаваемый параметром `comparer`. Массив должен быть одномерным
`public static void Sort(Array keys, Array items)`	Сортирует по нарастающей два заданных одномерных массива. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, оба массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются по порядку нарастания ключей

Продолжение табл. 21.12

Метод	Назначение
`public static void Sort<TKey, TValue>(TKey[] keys, TV[] items)`	Сортирует по нарастающей два заданных одномерных массива. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, оба массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются по порядку возрастания ключей
`public static void Sort(Array keys, Array items, Icomparer comparer)`	Сортирует по нарастающей два заданных одномерных массива, используя способ сравнения, задаваемый параметром `comparer`. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, оба массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются по порядку возрастания ключей
`public static void Sort<TKey, TValue>(TKey[] keys, TValue[] items, IComparer<TKey> comparer)`	Сортирует по нарастающей два заданных одномерных массива, используя способ сравнения, задаваемый параметром `comparer`. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, оба массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются по порядку возрастания ключей
`public static void Sort(Array array, int index, int length)`	Сортирует массив `array` по нарастающей в заданных пределах, начиная с элемента, указываемого по индексу `array[index]`, и включая число элементов, определяемых параметром `length`. Массив должен быть одномерным
`public static void Sort<T>(T[] array, int index, int length)`	Сортирует массив `array` по нарастающей в заданных пределах, начиная с элемента, указываемого по индексу `array[index]`, и включая число элементов, определяемых параметром `length`. Массив должен быть одномерным
`public static void Sort(Array array, int index, int length, IComparer comparer)`	Сортирует массив `array` по нарастающей в заданных пределах, начиная с элемента, указываемого по индексу `array[index]`, и включая число элементов, определяемых параметром `length`, а также используя способ сравнения, задаваемый параметром `v`. Массив должен быть одномерным
`public static void Sort<T>(T[] array, int index, int length, Icomparer<T> comparer)`	Сортирует массив `array` по нарастающей в заданных пределах, начиная с элемента, указываемого по индексу `array[index]`, и включая число элементов, определяемых параметром `length`, а также используя способ сравнения, задаваемый параметром `comparer`. Массив должен быть одномерным

Продолжение табл. 21.12

Метод	Назначение
`public static void Sort(Array keys, Array items, int index, int length)`	Сортирует по нарастающей два одномерных массива в заданных пределах, начиная с элемента, указываемого по индексу `index`, и включая число элементов, определяемых параметром `length`. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, оба массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются в заданных пределах по порядку возрастания ключей
`public static void Sort<TKey, TValue>(TKey[] keys, TValue[] items, int index, int length)`	Сортирует по нарастающей два одномерных массива в заданных пределах, начиная с элемента, указываемого по индексу `index`, и включая число элементов, определяемых параметром `length`. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, оба массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются в заданных пределах по порядку возрастания ключей
`public static void Sort(Array keys, Array items, int index, int length, IComparer comparer)`	Сортирует по нарастающей два одномерных массива в заданных пределах, начиная с элемента, указываемого по индексу `index`, и включая число элементов, определяемых параметром `length`, а также используя способ сравнения, задаваемый параметром `comparer`. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются в заданных пределах по порядку возрастания ключей
`public static void Sort<TKey, TValue>(TKey[] keys, TV items, int index, int length, Icomparer<TKey> comparer)`	Сортирует по нарастающей два одномерных массива в заданных пределах, начиная с элемента, указываемого по индексу `index`, и включая число элементов, определяемых параметром `length`, а также используя способ сравнения, задаваемый параметром `comparer`. Массив `keys` содержит ключи сортировки, а массив `items` — значения, связанные с этими ключами. Следовательно, эти два массива должны содержать пары "ключ-значение". После сортировки элементы обоих массивов располагаются в заданных пределах по порядку возрастания ключей

Окончание табл. 21.12

Метод	Назначение
`public static bool TrueForAll<T>(T[] array, Predicate<T> match)`	Возвращает логическое значение `true`, если все элементы массива *`array`* удовлетворяют условию предиката, задаваемого параметром *`match`*. Если один или более элементов этого массива не удовлетворяют заданному условию, то возвращается логическое значение `false`

Сортировка и поиск в массивах

Содержимое массива нередко приходится сортировать. Для этой цели в классе `Array` предусмотрен обширный ряд сортирующих методов. Так, с помощью разных вариантов метода `Sort()` можно отсортировать массив полностью или в заданных пределах либо отсортировать два массива, содержащих соответствующие пары "ключ–значение". После сортировки в массиве можно осуществить эффективный поиск, используя разные варианты метода `BinarySearch()`. В качестве примера ниже приведена программа, в которой демонстрируется применение методов `Sort()` и `BinarySearch()` для сортировки и поиска в массиве значений типа `int`.

```
// Отсортировать массив и найти в нем значение.

using System;

class SortDemo {
  static void Main() {
    int[] nums = { 5, 4, 6, 3, 14, 9, 8, 17, 1, 24, -1, 0 };

    // Отобразить исходный порядок следования.
    Console.Write("Исходный порядок следования: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Отсортировать массив.
    Array.Sort(nums);

    // Отобразить порядок следования после сортировки.
    Console.Write("Порядок следования после сортировки: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Найти значение 14.
    int idx = Array.BinarySearch(nums, 14);
      Console.WriteLine("Индекс элемента массива со значением 14: " +
         idx);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Исходный порядок следования: 5 4 6 3 14 9 8 17 1 24 -1 0
Порядок следования после сортировки: -1 0 1 3 4 5 6 8 9 14 17 24
Индекс элемента массива со значением 14: 9
```

В приведенном выше примере массив состоит из элементов типа `int`, который относится к категории типов значений. Все методы, определенные в классе `Array`, автоматически доступны для обработки массивов всех встроенных в C# типов значений. Но в отношении массивов ссылок на объекты это правило может и не соблюдаться. Так, для сортировки массива ссылок на объекты в классе типа этих объектов должен быть реализован интерфейс `IComparable` или `IComparable<T>`. Если же ни один из этих интерфейсов не реализован в данном классе, то во время выполнения программы может возникнуть исключительная ситуация в связи с попыткой отсортировать подобный массив или осуществить в нем поиск. Правда, реализовать оба интерфейса, `IComparable` и `IComparable<T>`, совсем нетрудно.

В интерфейсе `IComparable` определяется один метод.

```
int CompareTo(object obj)
```

В этом методе значение вызывающего объекта сравнивается со значением объекта, определяемого параметром *obj*. Если значение вызывающего объекта больше, чем у объекта *obj*, то возвращается положительное значение; если оба значения равны — нулевое значение, а если значение вызывающего объекта меньше, чем у объекта *obj*, — отрицательное значение.

Интерфейс `IComparable<T>` является обобщенным вариантом интерфейса `IComparable`. Поэтому в нем определен следующий обобщенный вариант метода `CompareTo()`.

```
int CompareTo(T other)
```

Обобщенный вариант метода `CompareTo()` действует аналогично необобщенному его варианту. В нем значение вызывающего объекта также сравнивается со значением объекта, определяемого параметром *other*. Если значение вызывающего объекта больше, чем у объекта *other*, то возвращается положительное значение; если оба значения равны — нулевое значение, а если значение вызывающего объекта меньше, чем у объекта *other*, — отрицательное значение. Преимущество интерфейса `IComparable<T>` заключается в том, что он обеспечивает типовую безопасность, поскольку в этом случае тип обрабатываемых данных указывается явным образом, а следовательно, никакого приведения типа `object` сравниваемого объекта к нужному типу не требуется. В качестве примера ниже приведена программа, в которой демонстрируются сортировка и поиск в массиве объектов определяемого пользователем класса.

```
// Отсортировать массив объектов и осуществить в нем поиск.

using System;

class MyClass : IComparable<MyClass> {
  public int i;

  public MyClass(int x) { i = x; }

  // Реализовать интерфейс IComparable<MyClass>.
  public int CompareTo(MyClass v) {
    return i - v.i;
  }

  public bool Equals(MyClass v) {
    return i == v.i;
  }
```

```
}

class SortDemo {
  static void Main() {
    MyClass[] nums = new MyClass[5];

    nums[0] = new MyClass(5);
    nums[1] = new MyClass(2);
    nums[2] = new MyClass(3);
    nums[3] = new MyClass(4);
    nums[4] = new MyClass(1);

    // Отобразить исходный порядок следования.
    Console.Write("Исходный порядок следования: ");
    foreach(MyClass o in nums)
      Console.Write(o.i + " ");
    Console.WriteLine();

    // Отсортировать массив.
    Array.Sort(nums);

    // Отобразить порядок следования после сортировки.
    Console.Write("Порядок следования после сортировки: ");
    foreach(MyClass o in nums)
      Console.Write(o.i + " ");
    Console.WriteLine();

    // Найти объект MyClass(2).
    MyClass x = new MyClass(2);
    int idx = Array.BinarySearch(nums, x);

    Console.WriteLine("Индекс элемента массива с объектом MyClass(2): " +
                      idx);
  }
}
```

При выполнении этой программы получается следующий результат.

```
Исходный порядок следования: 5 2 3 4 1
Порядок следования после сортировки: 1 2 3 4 5
Индекс элемента массива с объектом MyClass(2): 1
```

При сортировке или поиске в массиве строк может возникнуть потребность явно указать способ сравнения символьных строк. Так, если массив будет сортироваться с использованием одних настроек культурной среды, а поиск в нем — с помощью других настроек, то во избежание ошибок, скорее всего, придется явно указать способ сравнения. Аналогичная ситуация возникает и в том случае, если требуется отсортировать массив символьных строк при настройках культурной среды, отличающихся от текущих. Для выхода из подобных ситуаций можно передать экземпляр объекта типа `StringComparer` параметру типа `IComparer`, который поддерживается в целом ряде перегружаемых вариантов методов `Sort()` и `BinarySearch()`.

ПРИМЕЧАНИЕ

Более подробно особенности сравнения строк рассматриваются в главе 22.

Класс StringComparer объявляется в пространстве имен System и реализует, среди прочего, интерфейсы IComparer и IComparer<T>. Поэтому экземпляр объекта типа StringComparer может быть передан в качестве аргумента параметру типа IComparer. Кроме того, в классе StringComparer определен ряд доступных только для чтения свойств, возвращающих экземпляр объекта типа StringComparer и поддерживающих различные способы сравнения символьных строк. Все эти свойства перечислены ниже.

Свойство	Способ сравнения
public static StringComparer CurrentCulture {get; }	С учетом регистра и культурной среды
public static StringComparer CurrentCultureIgnoreCase {get; }	Без учета регистра, но с учетом культурной среды
public static StringComparer InvariantCulture {get; }	С учетом регистра и безотносительно к культурной среде
public static StringComparer InvariantCultureIgnoreCase {get; }	Без учета регистра и безотносительно к культурной среде
public static StringComparer Ordinal {get; }	Порядковое сравнение с учетом регистра
public static StringComparer OrdinalIgnoreCase {get; }	Порядковое сравнение без учета регистра

Передавая явным образом экземпляр объекта типа StringComparer, можно совершенно однозначно определить порядок сортировки или поиска в массиве. Например, в приведенном фрагменте кода сортировка и поиск в массиве символьных строк осуществляется с помощью свойства StringComparer.Ordinal.

```
string[] strs = { "xyz", "one" , "beta", "Alpha" };
// ...
Array.Sort(strs, StringComparer.Ordinal);
int idx = Array.BinarySearch(strs, "beta", StringComparer.Ordinal);
```

Обращение содержимого массива

Иногда оказывается полезно обратить содержимое массива и, в частности, отсортировать по убывающей массив, отсортированный по нарастающей. Для такого обращения массива достаточно вызвать метод Reverse(). С его помощью можно обратить содержимое массива полностью или частично. Этот процесс демонстрируется в приведенной ниже программе.

```
// Обратить содержимое массива.

using System;

class ReverseDemo {
  static void Main() {
    int[] nums = { 1, 2, 3, 4, 5 };

    // Отобразить исходный порядок следования.
    Console.Write("Исходный порядок следования: ");
```

```
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Обратить весь массив.
    Array.Reverse(nums);

    // Отобразить обратный порядок следования.
    Console.Write("Обратный порядок следования: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Обратить часть массива.
    Array.Reverse(nums, 1, 3);

    // Отобразить обратный порядок следования.
    Console.Write("Частично обращенный порядок следования: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

Эта программа дает следующий результат.

```
Исходный порядок следования: 1 2 3 4 5
Обратный порядок следования: 5 4 3 2 1
Частично обращенный порядок следования: 5 2 3 4 1
```

Копирование массива

Полное или частичное копирование одного массива в другой — это еще одна весьма распространенная операция с массивами. Для копирования содержимого массива служит метод `Copy()`. В зависимости от его варианта копирование элементов исходного массива осуществляется в начало или в средину целевого массива. Применение метода `Copy()` демонстрируется в приведенном ниже примере программы.

```
// Скопировать массив.

using System;

class CopyDemo {
  static void Main() {
    int[] source = { 1, 2, 3, 4, 5 };
    int[] target = { 11, 12, 13, 14, 15 };
    int[] source2 = { -1, -2, -3, -4, -5 };

    // Отобразить исходный массив.
    Console.Write("Исходный массив: ");
    foreach(int i in source)
      Console.Write(i + " ");
    Console.WriteLine();
```

```
    // Отобразить исходное содержимое целевого массива.
    Console.Write("Исходное содержимое целевого массива: ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();

    // Скопировать весь массив.
    Array.Copy(source, target, source.Length);

    // Отобразить копию.
    Console.Write("Целевой массив после копирования: ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();

    // Скопировать в средину целевого массива.
    Array.Copy(source2, 2, target, 3, 2);

    // Отобразить копию.
    Console.Write("Целевой массив после частичного копирования: ");
    foreach(int i in target)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

Выполнение этой программы дает следующий результат.

```
Исходный массив: 1 2 3 4 5
Исходное содержимое целевого массива: 11 12 13 14 15
Целевой массив после копирования: 1 2 3 4 5
Целевой массив после частичного копирования: 1 2 3 -3 -4
```

Применение предиката

Предикат представляет собой делегат типа `System.Predicate`, возвращающий логическое значение `true` или `false` в зависимости от некоторого условия. Он объявляется следующим образом.

```
public delegate bool Predicate<T> (T obj)
```

Объект, проверяемый по заданному условию, передается в качестве параметра *obj*. Если объект *obj* удовлетворяет заданному условию, то предикат должен возвратить логическое значение `true`, в противном случае — логическое значение `false`. Предикаты используются в ряде методов класса `Array`, включая: `Exists()`, `Find()`, `FindIndex()` и `FindAll()`.

В приведенном ниже примере программы демонстрируется применение предиката с целью определить, содержится ли в целочисленном массиве отрицательное значение. Если такое значение обнаруживается, то данная программа извлекает первое отрицательное значение, найденное в массиве. Для этого в ней используются методы `Exists()` и `Find()`.

```
// Продемонстрировать применение предикатного делегата.

using System;
```

```
class PredDemo {

  // Предикатный метод, возвращающий логическое значение true,
  // если значение переменной v оказывается отрицательным.
  static bool IsNeg(int v) {
    if(v < 0) return true;
    return false;
  }

  static void Main() {
    int[] nums = { 1, 4, -1, 5, -9 };
    Console.Write("Содержимое массива nums: ");
    foreach(int i in nums)
      Console.Write(i + " ");
    Console.WriteLine();

    // Сначала проверить, содержит ли массив nums отрицательное значение.
    if(Array.Exists(nums, PredDemo.IsNeg)) {
      Console.WriteLine("Массив nums содержит отрицательное значение.");

    // Затем найти первое отрицательное значение в массиве.
    int x = Array.Find(nums, PredDemo.IsNeg);
    Console.WriteLine("Первое отрицательное значение: " + x);
    }
    else
      Console.WriteLine("В массиве nums отсутствуют отрицательные значения.");
  }
}
```

Эта программа дает следующий результат.

```
Содержимое массива nums: 1 4 -1 5 -9
Массив nums содержит отрицательное значение.
Первое отрицательное значение: -1
```

В данном примере программы в качестве предиката методам `Exists()` и `Find()` передается метод `IsNeg()`. Обратите внимание на следующее объявление метода `IsNeg()`.

```
static bool IsNeg(int v) {
```

Методы `Exists()` и `Find()` автоматически и по порядку передают элементы массива переменной `v`. Следовательно, после каждого вызова метода `IsNeg()` переменная `v` будет содержать следующий элемент массива.

Применение делегата `Action`

Делегат `Action` применяется в методе `Array.ForEach()` для выполнения заданного действия над каждым элементом массива. Существуют разные формы делегата `Action`, отличающиеся числом параметров типа. Ниже приведена одна из таких форм.

```
public delegate void Action<T> (T obj)
```

В этой форме объект, над которым должно выполняться действие, передается в качестве параметра *obj*. Когда же эта форма делегата `Action` применяется в методе `Array.ForEach()`, то каждый элемент массива передается по порядку объекту *obj*.

Следовательно, используя делегат Action и метод ForEach(), можно в одном операторе выполнить заданную операцию над целым массивом.

В приведенном ниже примере программы демонстрируется применение делегата Action и метода ForEach(). Сначала в ней создается массив объектов класса MyClass, а затем используется метод Show() для отображения значений, извлекаемых из этого массива. Далее эти значения становятся отрицательными с помощью метода Neg(). И наконец, метод Show() используется еще раз для отображения отрицательных значений. Все эти операции выполняются посредством вызовов метода ForEach().

```
// Продемонстрировать применение делегата Action.

using System;

class MyClass {
  public int i;

  public MyClass(int x) { i = x; }
}

class ActionDemo {
  // Метод делегата Action, отображающий значение, которое ему передается.
  static void Show(MyClass o) {
    Console.Write(o.i + " ");
  }

  // Еще один метод делегата Action, делающий
  // отрицательным значение, которое ему передается.
  static void Neg(MyClass o) {
    o.i = -o.i;
  }

static void Main() {
    MyClass[] nums = new MyClass[5];

    nums[0] = new MyClass(5);
    nums[1] = new MyClass(2);
    nums[2] = new MyClass(3);
    nums[3] = new MyClass(4);
    nums[4] = new MyClass(1);

    Console.Write("Содержимое массива nums: ");

   // Выполнить действие для отображения значений.
   Array.ForEach(nums, ActionDemo.Show);

   Console.WriteLine();

    // Выполнить действие для отрицания значений.
    Array.ForEach(nums, ActionDemo.Neg);

    Console.Write("Содержимое массива nums после отрицания: ");

    // Выполнить действие для повторного отображения значений.
    Array.ForEach(nums, ActionDemo.Show);

    Console.WriteLine();
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Содержимое массива nums: 5 2 3 4 1
Содержимое массива nums после отрицания: -5 -2 -3 -4 -1
```

Класс BitConverter

В программировании нередко требуется преобразовать встроенный тип данных в массив байтов. Допустим, что на некоторое устройство требуется отправить целое значение, но сделать это нужно отдельными байтами, передаваемыми по очереди. Часто возникает и обратная ситуация, когда данные получаются из устройства в виде упорядоченной последовательности байтов, которые требуется преобразовать в один из встроенных типов. Для подобных преобразований в среде .NET предусмотрен отдельный класс `BitConverter`.

Класс `BitConverter` является статическим. Он содержит методы, приведенные в табл. 21.13. Кроме того, в нем определено следующее поле.

```
public static readonly bool IsLittleEndian
```

Это поле принимает логическое значение `true`, если в текущей среде сначала сохраняется младший байт слова, а затем старший. Это так называемый формат с *прямым* порядком байтов. А если в текущей среде сначала сохраняется старший байт слова, а затем младший, то поле `IsLittleEndian` принимает логическое значение `false`. Это так называемый формат с *обратным* порядком байтов. В компьютерах с процессором Intel Pentium используется формат с прямым порядком байтов.

Таблица 21.13. Методы, определенные в классе BitConverter

Метод	Назначение
`public static long DoubleToInt64Bits(double value)`	Преобразует значение *value* в целочисленное значение типа `long` и возвращает результат
`public static byte[] GetBytes(bool value)`	Преобразует значение *value* в однобайтовый массив и возвращает результат
`public static byte[] GetBytes(char value)`	Преобразует значение *value* в двухбайтовый массив и возвращает результат
`public static byte[] GetBytes(double value)`	Преобразует значение *value* в восьмибайтовый массив и возвращает результат
`public static byte[] GetBytes(float value)`	Преобразует значение *value* в четырехбайтовый массив и возвращает результат
`public static byte[] GetBytes(int value)`	Преобразует значение *value* в четырехбайтовый массив и возвращает результат
`public static byte[] GetBytes(long value)`	Преобразует значение *value* в восьмибайтовый массив и возвращает результат
`public static byte[] GetBytes(short value)`	Преобразует значение *value* в двухбайтовый массив и возвращает результат
`public static byte[] GetBytes(uint value)`	Преобразует значение *value* в четырехбайтовый массив и возвращает результат
`public static byte[] GetBytes(ulong value)`	Преобразует значение *value* в восьмибайтовый массив и возвращает результат

Продолжение табл. 21.13

Метод	Назначение
`public static byte[] GetBytes(ushort value)`	Преобразует значение `value` в двухбайтовый массив и возвращает результат
`public static double Int64BitsToDouble(long value)`	Преобразует значение `value` в значение типа `double` и возвращает результат
`public static bool ToBoolean(byte[] value, int startIndex)`	Преобразует байт из элемента массива, указываемого по индексу `value[startIndex]`, в эквивалентное значение типа `bool` и возвращает результат. Ненулевое значение преобразуется в логическое значение `true`, а нулевое — в логическое значение `false`
`public static char ToChar(byte[] value, int index)`	Преобразует два байта, начиная с элемента массива `value[index]`, в эквивалентное значение типа `char` и возвращает результат
`public static double ToDouble(byte[] value, int startIndex)`	Преобразует восемь байтов, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `double` и возвращает результат
`public static short ToInt16(byte[] value, int startIndex)`	Преобразует два байта, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `short` и возвращает результат
`public static int ToInt32(byte[] value, int startIndex)`	Преобразует четыре байта, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `int` и возвращает результат
`public static long ToInt64(byte[] value, int startIndex)`	Преобразует восемь байтов, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `long` и возвращает результат
`public static float ToSingle(byte[] value, int startIndex)`	Преобразует четыре байта, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `float` и возвращает результат
`public static string ToString(byte[] value)`	Преобразует байты из массива `value` в символьную строку. Строка содержит шестнадцатеричные значения, связанные с этими байтами и разделенные дефисами
`public static string ToString(byte[] value, int startIndex)`	Преобразует байты из массива `value` в символьную строку, начиная с элемента `value[startIndex]`. Строка содержит шестнадцатеричные значения, связанные с этими байтами и разделенные дефисами
`public static string ToString(byte[] value, int startIndex, int length)`	Преобразует байты из массива `value` в символьную строку, начиная с элемента `value[startIndex]` и включая число элементов, определяемых параметром `length`. Строка содержит шестнадцатеричные значения, связанные с этими байтами и разделенные дефисами
`public static ushort ToUInt16(byte[] value, int startIndex)`	Преобразует два байта, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `ushort` и возвращает результат
`public static uint ToUInt32(byte[] value, int startIndex)`	Преобразует четыре байта, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `uint` и возвращает результат

Окончание табл. 21.13

Метод	Назначение
`public static ulong ToUInt64(byte[] value, int startIndex)`	Преобразует восемь байтов, начиная с элемента массива `value[startIndex]`, в эквивалентное значение типа `ulong` и возвращает результат

Генерирование случайных чисел средствами класса Random

Для генерирования последовательного ряда случайных чисел служит класс `Random`. Такие последовательности чисел оказываются полезными в самых разных ситуациях, включая имитационное моделирование. Начало последовательности случайных чисел определяется некоторым начальным числом, которое может задаваться автоматически или указываться явным образом.

В классе `Random` определяются два конструктора.

```
public Random()
public Random(int seed)
```

Первый конструктор создает объект типа `Random`, использующий системное время для определения начального числа. А во втором конструкторе используется начальное значение *seed*, задаваемое явным образом.

Методы, определенные в классе `Random`, перечислены в табл. 21.14.

Таблица 21.14. Методы, определенные в классе Random

Метод	Назначение
`public virtual int Next()`	Возвращает следующее случайное целое число, которое будет находиться в пределах от 0 до `Int32.MaxValue-1` включительно
`public virtual int Next(int maxValue)`	Возвращает следующее случайное целое число, которое будет находиться в пределах от 0 до *maxValue-1* включительно
`public virtual int Next(int minValue, int maxValue)`	Возвращает следующее случайное целое число, которое будет находиться в пределах от *minValue* до *maxValue-1* включительно
`public virtual void NextBytes(byte[] buffer)`	Заполняет массив *buffer* последовательностью случайных целых чисел. Каждый байт в массиве будет находиться в пределах от 0 до `Byte.MaxValue-1` включительно
`public virtual double NextDouble()`	Возвращает из последовательности следующее случайное число, которое представлено в форме с плавающей точкой, больше или равно 0,0 и меньше 1,0
`protected virtual double Sample()`	Возвращает из последовательности следующее случайное число, которое представлено в форме с плавающей точкой, больше или равно 0,0 и меньше 1,0. Для получения несимметричного или специального распределения случайных чисел этот метод необходимо переопределить в производном классе

Ниже приведена программа, в которой применение класса Random демонстрируется на примере создания компьютерного варианта пары игральных костей.

```
// Компьютерный вариант пары игральных костей.

using System;

class RandDice {
  static void Main() {
    Random ran = new Random();

    Console.Write(ran.Next(1, 7) + " ");
    Console.WriteLine(ran.Next(1, 7));
  }
}
```

При выполнении этой программы три раза подряд могут быть получены, например, следующие результаты.

```
5 2
4 4
1 6
```

Сначала в этой программе создается объект класса Random. А затем в ней запрашиваются два случайных значения в пределах от 1 до 6.

Управление памятью и класс GC

В классе GC инкапсулируются средства "сборки мусора". Методы, определенные в этом классе, перечислены в табл. 21.15.

Таблица 21.15. Методы, определенные в классе GC

Метод	Назначение
`public static voidAddMemoryPressure(long bytesAllocated)`	Задает в качестве параметра *bytesAllocated* количество байтов, распределенных в неуправляемой области памяти
`public static void CancelFullGCNotification()`	Отменяет уведомление о "сборке мусора"
`public static void Collect()`	Инициализирует процесс "сборки мусора"
`public static void Collect(int generation)`	Инициализирует процесс "сборки мусора" в областях памяти с номерами поколений от 0 до *generation*
`public static void Collect(int generation, GCCollectionMode mode)`	Инициализирует процесс "сборки мусора" в областях памяти с номерами поколений от 0 до *generation* в режиме, определяемом параметром mode
`public static int CollectionCount(int generation)`	Возвращает количество операций "сборки мусора", выполненных в области памяти с номером поколения *generation*
`public static int GetGeneration(object obj)`	Возвращает номером поколения для области памяти, доступной по ссылке *obj*

Продолжение табл. 21.15

Метод	Назначение
`public static int GetGeneration(WeakReference wo)`	Возвращает номер поколения для области памяти, доступной по "слабой" ссылке, задаваемой параметром *wo*. Наличие "слабой" ссылки не защищает объект от "сборки мусора"
`public static long GetTotalMemory(bool forceFullCollection)`	Возвращает общий объем памяти (в байтах), выделенной на данный момент. Если параметр *forceFullCollection* имеет логическое значение `true`, то сначала выполняется "сборка мусора"
`public static void KeepAlive(object obj)`	Создает ссылку на объект `obj`, защищая его от "сборки мусора". Действие этой ссылки оканчивается после выполнения метода `KeepAlive()`
`public static void RegisterForFullGCNotification(int maxGenerationThreshold, int largeObjectHeapThreshold)`	Разрешает уведомление о "сборке мусора". Значение параметра *maxGenerationThreshold* обозначает количество объектов второго поколения в обычной "куче", которые будут инициировать уведомление. А значение параметра *largeObjectHeapThreshold* обозначает количество объектов в крупной "куче", которые будут инициировать уведомление. Оба значения должны быть указаны в пределах от 1 до 99
`public static void RemoveMemoryPressure(long bytesAllocated)`	Задает в качестве параметра *bytesAllocated* количество байтов, освобождаемых в неуправляемой области памяти
`public static void ReRegisterForFinalize(object obj)`	Вызывает деструктор для объекта *obj*. Этот метод аннулирует действие метода `SuppressFinalize()`
`public static void SuppressFinalize(object obj)`	Препятствует вызову деструктора для объекта *obj*
`public static GCNotificationStatus WaitForFullGCApproach()`	Ожидает уведомления о том, что должен произойти полный цикл "сборки мусора". Здесь `GCNotificationStatus` — перечисление, определенное в пространстве имен `System`
`public static GCNotificationStatus WaitForFullGCApproach(int millisecondsTimeout)`	Ожидает уведомления о том, что должен произойти полный цикл "сборки мусора", в течение времени, задаваемого параметром *millisecondsTimeout*. Здесь `GCNotificationStatus` — перечисление, определенное в пространстве имен `System`
`public static GCNotificationStatus WaitForFullGCComplete()`	Ожидает уведомления о завершении полного цикла "сборки мусора". Здесь `GCNotificationStatus` — перечисление, определенное в пространстве имен `System`

Окончание табл. 21.15

Метод	Назначение
`public static GCNotificationStatus WaitForFullGCComplete(int millisecondsTimeout)`	Ожидает уведомления о завершении полного цикла "сборки мусора" в течение времени, задаваемого параметром *millisecondsTimeout*. Здесь `GCNotificationStatus` — перечисление, определенное в пространстве имен `System`
`public static void WaitForPendingFinalizers()`	Прекращает выполнение вызывающего потока до тех пор, пока не будут выполнены все вызванные и незавершенные деструкторы

Кроме того, в классе `GC` определяется следующее доступное только для чтения свойство:

```
public static int MaxGeneration { get; }
```

Свойство `MaxGeneration` содержит максимальный номер поколения, доступный для системы. Номер поколения обозначает возраст выделенной области памяти. Чем старше выделенная область памяти, тем больше номер ее поколения. Номера поколений позволяют повысить эффективность работы системы "сборки мусора".

В большинстве приложений возможности класса `GC` не используются. Но в особых случаях они оказываются весьма полезными. Допустим, что требуется организовать принудительную "сборку мусора" с помощью метода `Collect()` в выбранный момент времени. Как правило, "сборка мусора" происходит в моменты, не указываемые специально в программе. А поскольку для ее выполнения требуется некоторое время, то желательно, чтобы она не происходила в тот момент, когда решается критичная по времени задача. С другой стороны, "сборку мусора" и другие вспомогательные операции можно выполнить во время простоя программы. Имеется также возможность регистрировать уведомления о приближении и завершении "сборки мусора".

Для проектов с неуправляемым кодом особое значение имеют два следующих метода из класса `GC`: `AddMemoryPressure()` и `RemoveMemoryPressure()`. С их помощью указывается большой объем неуправляемой памяти, выделяемой или освобождаемой в программе. Особое значение этих методов состоит в том, что система управления памятью не контролирует область неуправляемой памяти. Если программа выделяет большой объем неуправляемой памяти, то это может сказаться на производительности, поскольку системе ничего неизвестно о таком сокращении объема свободно доступной памяти. Если же большой объем неуправляемой памяти выделяется с помощью метода `AddMemoryPressure()`, то система CLR уведомляется о сокращении объема свободно доступной памяти. А если выделенная область памяти освобождается с помощью метода `RemoveMemoryPressure()`, то система CLR уведомляется о соответствующем восстановлении объема свободно доступной памяти. Следует, однако, иметь в виду, что метод `RemoveMemoryPressure()` необходимо вызывать только для уведомления об освобождении области неуправляемой памяти, выделенной с помощью метода `AddMemoryPressure()`.

Класс object

В основу типа object в C# положен класс object. Члены класса Object подробно рассматривались в главе 11, но поскольку он играет главную роль в C#, то его методы ради удобства повторно перечисляются в табл. 21.16. В классе object определен конструктор

```
public Object()
```

который создает пустой объект.

Таблица 21.16. Методы, определенные в классе Object

Метод	Назначение
public virtual bool Equals(object *obj*)	Возвращает логическое значение true, если вызывающий объект оказывается таким же, как и объект, определяемый параметром *obj*. В противном случае возвращается значение false
public static bool Equals(object *objA*, object *objB*)	Возвращает логическое значение true, если объект *objA* оказывается таким же, как и объект *objB*. В противном случае возвращается значение false
protected Finalize()	Выполняет завершающие действия перед процессом "сборки мусора". В C# метод Finalize() доступен через деструктор
public virtual int GetHashCode()	Возвращает хеш-код, связанный с вызывающим объектом
public Type GetType()	Получает тип объекта во время выполнения программы
protected object MemberwiseClone()	Создает "неполную" копию объекта. При этом копируются члены, но не объекты, на которые ссылаются эти члены
public static bool ReferenceEquals(object *objA*, object *objB*)	Возвращает логическое значение true, если объекты *objA* и *objB* ссылаются на один и тот же объект. В противном случае возвращается логическое значение false
public virtual string ToString()	Возвращает строку, описывающую объект

Класс Tuple

В версии .NET Framework 4.0 внедрен удобный способ создания групп объектов (так называемых кортежей). В основу этого способа положен статический класс Tuple, в котором определяется несколько вариантов метода Create() для создания кортежей, а также различные обобщенные классы типа Tuple<...>, в которых инкапсулируются кортежи. В качестве примера ниже приведено объявление варианта метода Create(), возвращающего кортеж с тремя членами.

```
public static Tuple<T1, T2, T3>
  Create<T1, T2, T3>(T1 item1, T2 item2, T3 item3)
```

Следует заметить, что данный метод возвращает объект типа `Tuple<T1, T2, T3>`, в котором инкапсулируются члены кортежа *item1*, *item2* и *item3*. Вообще говоря, кортежи оказываются полезными в том случае, если группу значений нужно интерпретировать как единое целое. В частности, кортежи можно передавать методам, возвращать из методов или же сохранять в коллекции либо в массиве.

Интерфейсы IComparable и IComparable<T>

Во многих классах приходится реализовывать интерфейс `IComparable` или `IComparable<T>`, поскольку он позволяет сравнивать один объект с другим, используя различные методы, определенные в среде .NET Framework. Интерфейсы `IComparable` и `IComparable<T>` были представлены в главе 18, где они использовались в примерах программ для сравнения двух объектов, определяемых параметрами обобщенного типа. Кроме того, они упоминались при рассмотрении класса `Array` ранее в этой главе. Но поскольку эти интерфейсы имеют особое значение и применяются во многих случаях, то ниже приводится их краткое описание.

Интерфейс `IComparable` реализуется чрезвычайно просто, потому что он состоит всего лишь из одного метода.

```
int CompareTo(object obj)
```

В этом методе значение вызывающего объекта сравнивается со значением объекта, определяемого параметром *obj*. Если значение вызывающего объекта больше, чем у объекта *obj*, то возвращается положительное значение; если оба значения равны — нулевое значение, а если значение вызывающего объекта меньше, чем у объекта *obj*, — отрицательное значение.

Обобщенный вариант интерфейса `IComparable` объявляется следующим образом.

```
public interface IComparable<T>
```

В данном варианте тип сравниваемых данных передается параметру `T` в качестве аргумента типа. В силу этого объявление метода `CompareTo()` претерпевает изменения и выглядит так, как показано ниже.

```
int CompareTo(T other)
```

В этом объявлении тип данных, которыми оперирует метод `CompareTo()`, может быть указан явным образом. Следовательно, интерфейс `IComparable<T>` обеспечивает типовую безопасность. Именно по этой причине он теперь считается более предпочтительным в программировании на C#, чем интерфейс `IComparable`.

Интерфейс IEquatable<T>

Интерфейс `IEquatable<T>` реализуется в тех классах, где требуется определить порядок сравнения двух объектов на равенство их значений. В этом интерфейсе определен только один метод, `Equals()`, объявление которого приведено ниже.

```
bool Equals(T other)
```

Этот метод возвращает логическое значение `true`, если значение вызывающего объекта оказывается равным значению другого объекта *other*, в противном случае — логическое значение `false`.

Интерфейс `IEquatable<T>` реализуется в нескольких классах и структурах среды .NET Framework, включая структуры числовых типов и класс `String`. Для реализации интерфейса `IEquatable<T>` обычно требуется также переопределять методы `Equals(Object)` и `GetHashCode()`, определенные в классе `Object`.

Интерфейс IConvertible

Интерфейс `IConvertible` реализуется в структурах всех типов значений, `String` и `DateTime`. В нем определяются различные преобразования типов. Реализовывать этот интерфейс в создаваемых пользователем классах, как правило, не требуется.

Интерфейс ICloneable

Реализовав интерфейс `ICloneable`, можно создать все условия для копирования объекта. В интерфейсе `ICloneable` определен только один метод, `Clone()`, объявление которого приведено ниже.

```
object Clone()
```

В этом методе создается копия вызывающего объекта, а конкретная его реализация зависит от способа создания копии объекта. Вообще говоря, существуют две разновидности копий объектов: полная и неполная. Если создается полная копия, то копия совершенно не зависит от оригинала. Так, если в исходном объекте содержится ссылка на другой объект *O*, то при его копировании создается также копия объекта *O*. А при создании неполной копии осуществляется копирование одних только членов, но не объектов, на которые эти члены ссылаются. Так, после создания неполной копии объекта, ссылающегося на другой объект *O*, копия и оригинал будут ссылаться на один и тот же объект *O*, причем любые изменения в объекте *O* будут оказывать влияние как на копию, так и на оригинал. Как правило, метод `Clone()` реализуется для получения полной копии. А неполные копии могут быть созданы с помощью метода `MemberwiseClone()`, определенного в классе `Object`.

Ниже приведен пример программы, в которой демонстрируется применение интерфейса `ICloneable`. В ней создается класс `Test`, содержащий ссылку на объект класса `X`. В самом классе `Test` используется метод `Clone()` для создания полной копии.

```
// Продемонстрировать применение интерфейса ICloneable.

using System;

class X {
  public int a;

  public X(int x) { a = x; }
}

class Test : ICloneable {
```

```
  public X o;
  public int b;

  public Test(int x, int y) {
    o = new X(x);
    b = y;
  }

  public void Show(string name) {
    Console.Write("Значения объекта " + name + ": ");
    Console.WriteLine("o.a: {0}, b: {1}", o.a, b);
  }

  // Создать полную копию вызывающего объекта.
  public object Clone() {
    Test temp = new Test(o.a, b);
    return temp;
  }
}

class CloneDemo {
  static void Main() {
    Test ob1 = new Test(10, 20);

    ob1.Show("ob1");

    Console.WriteLine("Сделать объект ob2 копией объекта ob1.");
    Test ob2 = (Test) ob1.Clone();

    ob2.Show("ob2");

    Console.WriteLine("Изменить значение ob1.o.a на 99, " +
                      " а значение ob1.b — на 88.");
    ob1.o.a = 99;
    ob1.b = 88;

    ob1.Show("ob1");
    ob2.Show("ob2");
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Значения объекта ob1: o.a: 10, b: 20
Сделать объект ob2 копией объекта ob1.
Значения объекта ob2: o.a: 10, b: 20
Изменить значение ob1.o.a на 99, а значение ob1.b — на 88.
Значения объекта ob1: o.a: 99, b: 88
Значения объекта ob2: o.a: 10, b: 20
```

Как следует из результата выполнения приведенной выше программы, объект `ob2` является копией объекта `ob1`, но это совершенно разные объекты. Изменения в одном из них не оказывают никакого влияния на другой. Это достигается конструированием нового объекта типа `Test`, который выделяет новый объект типа `X` для копирования. При этом новому экземпляру объекта типа `X` присваивается такое же значение, как и у объекта типа `X` в оригинале.

Для получения неполной копии достаточно вызвать метод MemberwiseClone(), определяемый в классе Object из метода Clone(). В качестве упражнения попробуйте заменить метод Clone() в предыдущем примере программы на следующий его вариант.

```
// Сделать неполную копию вызывающего объекта.
public object Clone() {
  Test temp = (Test) MemberwiseClone();
  return temp;
}
```

После этого изменения результат выполнения данной программы будет выглядеть следующим образом.

```
Значения объекта ob1: o.a: 10, b: 20
Сделать объект ob2 копией объекта ob1.
Значения объекта ob2: o.a: 10, b: 20
Изменить значение ob1.o.a на 99, а значение ob1.b — на 88.
Значения объекта ob1: o.a: 99, b: 88
Значения объекта ob2: o.a: 99, b: 20
```

Как видите, обе переменные экземпляра o в объектах ob1 и ob2 ссылаются на один и тот же объект типа X. Поэтому изменения в одном объекте оказывают влияние на другой. Но в то же время поля b типа int в каждом из них разделены, поскольку типы значений недоступны по ссылке.

Интерфейсы IFormatProvider и IFormattable

В интерфейсе IFormatProvider определен единственный метод GetFormat(), который возвращает объект, определяющий форматирование данных в удобочитаемой форме текстовой строки. Ниже приведена общая форма метода GetFormat():

```
object GetFormat(Type formatType)
```

где *formatType* — это объект, получаемый для форматирования.

Интерфейс IFormattable поддерживает форматирование выводимых результатов в удобочитаемой форме. В нем определен следующий метод:

```
string ToString(string format, IFormatProvider formatProvider)
```

где *format* обозначает инструкции для форматирования, а *formatProvider* — поставщик формата.

ПРИМЕЧАНИЕ

Подробнее о форматировании речь пойдет в главе 22.

Интерфейсы IObservable<T> и IObserver<T>

В версию .NET Framework 4.0 добавлены еще два интерфейса, поддерживающие шаблон наблюдателя: IObservable<T> и IObserver<T>. В шаблоне наблюдателя один класс (в роли наблюдаемого) предоставляет уведомления другому классу (в роли

наблюдателя). С этой целью объект наблюдаемого класса регистрирует объект наблюдающего класса. Для регистрации наблюдателя вызывается метод `Subscribe()`, который определен в интерфейсе `IObservable<T>` и которому передается объект типа `IObserver<T>`, принимающий уведомление. Для получения уведомлений можно зарегистрировать несколько наблюдателей. А для отправки уведомлений всем зарегистрированным наблюдателям применяются три метода, определенные в интерфейсе `IObserver<T>`. Так, метод `OnNext()` отправляет данные наблюдателю, метод `OnError()` сообщает об ошибке, а метод `OnCompleted()` указывает на то, что наблюдаемый объект прекратил отправку уведомлений.

22
ГЛАВА

Строки и форматирование

В этой главе рассматривается класс `String`, положенный в основу встроенного в C# типа `string`. Как известно, обработка символьных строк является неотъемлемой частью практически всех программ. Именно по этой причине в классе `String` определяется обширный ряд методов, свойств и полей, обеспечивающих наиболее полное управление процессом построения символьных строк и манипулирования ими. С обработкой строк тесно связано форматирование данных в удобочитаемой форме. Используя подсистему форматирования, можно отформатировать данные всех имеющихся в C# числовых типов, а также дату, время и перечисления.

Строки в C#

Вопросы обработки строк уже обсуждались в главе 7, и поэтому не стоит повторяться. Вместо этого целесообразно дать краткий обзор реализации символьных строк в C#, прежде чем переходить к рассмотрению класса `String`.

Во всех языках программирования *строка* представляет собой последовательность символов, но конкретная ее реализация отличается в разных языках. В некоторых языках программирования, например в C++, строки представляют собой массивы символов, тогда как в C# они являются объектами встроенного типа данных `string`. Следовательно, `string` является ссылочным типом. Более того, `string` — это имя стандартного для среды .NET строкового типа `System.String`. Это означает, что в C# строке как объекту доступны все методы, свойства, поля и операторы, определенные в классе `String`.

После создания строки последовательность составляющих ее символов не может быть изменена. Благодаря этому ограничению строки реализуются в C# более эффективно. И хотя такое ограничение кажется на первый взгляд серьезным препятствием, на самом деле оно таковым не является. Когда требуется получить строку как разновидность уже существующей строки, достаточно создать новую строку, содержащую требующиеся изменения, и "отвергнуть" исходную строку, если она больше не нужна. А поскольку ненужные строковые объекты автоматически утилизируются средствами "сборки мусора", то беспокоиться о дальнейшей судьбе "отвергнутых" строк не приходится. Следует, однако, подчеркнуть, что переменные ссылок на строки могут, безусловно, изменить объект, на который они ссылаются. Но сама последовательность символов в конкретном строковом объекте не подлежит изменению после его создания.

Для создания строк, которые нельзя изменить, в C# предусмотрен класс `StringBuilder`, находящийся в пространстве имен `System.Text`. Но на практике для этой цели чаще используется тип `string`, а не класс `StringBuilder`.

Класс `String`

Класс `String` определен в пространстве имен `System`. В нем реализуются следующие интерфейсы: `IComparable`, `IComparable<string>`, `ICloneable`, `IConvertible`, `IEnumerable`, `IEnumerable<char>` и `IEquatable<string>`. Кроме того, `String` — герметичный класс, а это означает, что он не может наследоваться. В классе `String` предоставляются все необходимые функциональные возможности для обработки символьных строк в C#. Он служит основанием для встроенного в C# типа `string` и является составной частью среды .NET Framework. В последующих разделах представлено подробное описание класса `String`.

Конструкторы класса `String`

В классе `String` определено несколько конструкторов, позволяющих создавать строки самыми разными способами. Для создания строки из символьного массива служит один из следующих конструкторов.

```
public String(char[ ] value)
public String(char[ ] value, int startIndex, int length)
```

Первая форма конструктора позволяет создать строку, состоящую из символов массива *value*. А во второй форме для этой цели из массива *value* извлекается определенное количество символов (*length*), начиная с элемента, указываемого по индексу *startIndex*.

С помощью приведенного ниже конструктора можно создать строку, состоящую из отдельного символа, повторяющегося столько раз, сколько потребуется:

```
public String(char c, int count)
```

где *c* обозначает повторяющийся символ; а *count* — количество его повторений.

Кроме того, строку можно создать по заданному указателю на символьный массив, используя один из следующих конструкторов.

```
public String(char* value)
public String(char* value, int startIndex, int length)
```

Первая форма конструктора позволяет создать строку из символов, доступных из массива по указателю *value*. При этом предполагается, что массив, доступный по указателю *value*, завершается пустым символом, обозначающим конец строки. А во второй форме конструктора для этой цели из массива, доступного по указателю *value*, извлекается определенное количество символов (*length*), начиная с элемента, указываемого по индексу *startIndex*. В этих конструкторах применяются указатели, поэтому их можно использовать только в небезопасном коде.

И наконец, строку можно построить по заданному указателю на байтовый массив, используя один из следующих конструкторов.

```
public String(sbyte* value)
public String(sbyte* value, int startIndex, int length)
public String(sbyte* value, int startIndex, int length, Encoding enc)
```

Первая форма конструктора позволяет построить строку из отдельных байтов символов, доступных из массива по указателю *value*. При этом предполагается, что массив, доступный по указателю *value*, завершается признаком конца строки. Во второй форме конструктора для этой цели из массива, доступного по указателю *value*, извлекается определенное количество байтов символов (*length*), начиная с элемента, указываемого по индексу *startIndex*. А третья форма конструктора позволяет указать количество кодируемых байтов. Класс `Encoding` находится в пространстве имен `System.Text`. В этих конструкторах применяются указатели, и поэтому их можно использовать только в небезопасном коде.

При объявлении строкового литерала автоматически создается строковый объект. Поэтому для инициализации строкового объекта зачастую оказывается достаточно присвоить ему строковый литерал, как показано ниже.

```
string str = "новая строка";
```

Поле, индексатор и свойство класса `String`

В классе `String` определено единственное поле.

```
public static readonly string Empty
```

Поле `Empty` обозначает пустую строку, т.е. такую строку, которая не содержит символы. Этим оно отличается от пустой ссылки типа `String`, которая просто делается на несуществующий объект.

Помимо этого, в классе `String` определен единственный индексатор, доступный только для чтения.

```
public char this[int index] { get; }
```

Этот индексатор позволяет получить символ по указанному индексу. Индексация строк, как и массивов, начинается с нуля. Объекты типа `String` отличаются постоянством и не изменяются, поэтому вполне логично, что в классе `String` поддерживается индексатор, доступный только для чтения.

И наконец, в классе `String` определено единственное свойство, доступное только для чтения.

```
public int Length { get; }
```

Свойство `Length` возвращает количество символов в строке.

Операторы класса `String`

В классе `String` перегружаются два следующих оператора: == и !=. Оператор == служит для проверки двух символьных строк на равенство. Когда оператор == применяется к ссылкам на объекты, он обычно проверяет, делаются ли обе ссылки на один и тот же объект. А когда оператор == применяется к ссылкам на объекты типа `String`, то на предмет равенства сравнивается содержимое самих строк. Это же относится и к оператору !=. Когда он применяется к ссылкам на объекты типа `String`, то на предмет неравенства сравнивается содержимое самих строк. В то же время другие операторы отношения, в том числе < и >=, сравнивают ссылки на объекты типа `String` таким же образом, как и на объекты других типов. А для того чтобы проверить, является ли одна строка больше другой, следует вызвать метод `Compare()`, определенный в классе `String`.

Как станет ясно дальше, во многих видах сравнения символьных строк используются сведения о культурной среде. Но это не относится к операторам = = и !=. Ведь они просто сравнивают порядковые значения символов в строках. (Иными словами, они сравнивают двоичные значения символов, не видоизмененные нормами культурной среды, т.е. региональными стандартами.) Следовательно, эти операторы выполняют сравнение строк без учета регистра и настроек культурной среды.

Сравнение строк

Вероятно, из всех операций обработки символьных строк чаще всего выполняется сравнение одной строки с другой. Прежде чем рассматривать какие-либо методы сравнения строк, следует подчеркнуть следующее: сравнение строк может быть выполнено в среде .NET Framework двумя основными способами. Во-первых, сравнение может отражать обычаи и нормы отдельной культурной среды, которые зачастую представляют собой настройки культурной среды, вступающие в силу при выполнении программы. Это стандартное поведение некоторых, хотя и не всех методов сравнения. И во-вторых, сравнение может быть выполнено независимо от настроек культурной среды только по порядковым значениям символов, составляющих строку. Вообще говоря, при сравнении строк без учета культурной среды используется лексикографический порядок (и лингвистические особенности), чтобы определить, является ли одна строка больше, меньше или равной другой строке. При порядковом сравнении строки просто упорядочиваются на основании невидоизмененного значения каждого символа.

ПРИМЕЧАНИЕ

В силу отличий способов сравнения строк с учетом культурной среды и порядкового сравнения, а также последствий каждого такого сравнения настоятельно рекомендуется руководствоваться лучшими методиками, предлагаемыми в настоящее время корпорацией Microsoft. Ведь выбор неверного способа сравнения строк может привести к неправильной работе программы, когда она эксплуатируется в среде, отличающей от той, в которой она разработана.

Выбор способа сравнения символьных строк представляет собой весьма ответственное решение. Как правило и без всякий исключений, следует выбирать сравнение строк с учетом культурной среды, если это делается для целей отображения результата пользователю (например, для вывода на экран ряда строк, отсортированных в лексикографическом порядке). Но если строки содержат фиксированную информацию, не предназначенную для видоизменения с учетом отличий в культурных средах, на-

пример, имя файла, ключевое слово, адрес веб-сайта или значение, связанное с обеспечением безопасности, то следует выбрать порядковое сравнение строк. Разумеется, особенности конкретного разрабатываемого приложения будут диктовать выбор подходящего способа сравнения символьных строк.

В классе `String` предоставляются самые разные методы сравнения строк, перечисленные в табл. 22.1. Наиболее универсальным среди них является метод `Compare()`. Он позволяет сравнивать две строки полностью или частично, с учетом или без учета регистра, способа сравнения, определяемого параметром типа `StringComparison`, а также сведений о культурной среде, предоставляемых с помощью параметра типа `CultureInfo`. Те перегружаемые варианты метода `Compare()`, которые не содержат параметр типа `StringComparison`, выполняют сравнение символьных строк с учетом регистра и культурной среды. А в тех перегружаемых его вариантах, которые не содержат параметр типа `CultureInfo`, сведения о культурной среде определяются текущей средой выполнения. В примерах программ, приведенных в этой главе, параметр типа `CultureInfo` не используется, а большее внимание уделяется использованию параметра типа `StringComparison`.

Таблица 22.1. Методы сравнения символьных строк

Метод	Назначение
`public static int Compare(string strA, string strB)`	Сравнивает строку *strA* со строкой *strB*. Возвращает положительное значение, если строка *strA* больше строки *strB*; отрицательное значение, если строка *strA* меньше строки *strB*; и нуль, если строки *strA* и *strB* равны. Сравнение выполняется с учетом регистра и культурной среды
`public static int Compare(string strA, string strB, bool ignoreCase)`	Сравнивает строку *strA* со строкой *strB*. Возвращает положительное значение, если строка *strA* больше строки *strB*; отрицательное значение, если строка *strA* меньше строки *strB*; и нуль, если строки *strA* и *strB* равны. Если параметр *ignoreCase* принимает логическое значение `true`, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Сравнение выполняется с учетом культурной среды
`public static int Compare(string strA, string strB, StringComparison comparisonType)`	Сравнивает строку *strA* со строкой *strB*. Возвращает положительное значение, если строка *strA* больше строки *strB*; отрицательное значение, если строка *strA* меньше строки *strB*; и нуль, если строки *strA* и *strB* равны. Параметр *comparisonType* определяет конкретный способ сравнения строк
`public static int Compare(string strA, string strB, bool ignoreCase, CultureInfo culture)`	Сравнивает строку *strA* со строкой *strB*, используя информацию о культурной среде, определяемую параметром *culture*. Возвращает положительное значение, если строка *strA* больше строки *strB*; отрицательное значение, если строка *strA* меньше строки *strB*; и нуль, если строки *strA* и *strB* равны. Если параметр *ignoreCase* принимает логическое значение `true`, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Класс `CultureInfo` определен в пространстве имен `System.Globalization`

Продолжение табл. 22.1

Метод	Назначение
`public static int Compare(string strA, int indexA, string strB, int indexB, int length)`	Сравнивает части строк *strA* и *strB*. Сравнение начинается со строковых элементов *strA*[*indexA*] и *strB*[*indexB*] и включает количество символов, определяемых параметром *length*. Метод возвращает положительное значение, если часть строки *strA* больше части строки *strB*; отрицательное значение, если часть строки *strA* меньше части строки *strB*; и нуль, если сравниваемые части строк *strA* и *strB* равны. Сравнение выполняется с учетом регистра и культурной среды
`public static int Compare(string strA, int indexA, string strB, int indexB, int length, bool ignoreCase)`	Сравнивает части строк *strA* и *strB*. Сравнение начинается со строковых элементов *strA*[*indexA*] и *strB*[*indexB*] и включает количество символов, определяемых параметром *length*. Метод возвращает положительное значение, если часть строки *strA* больше части строки *strB*; отрицательное значение, если часть строки *strA* меньше части строки *strB*; и нуль, если сравниваемые части строк *strA* и *strB* равны. Если параметр *ignoreCase* принимает логическое значение `true`, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Сравнение выполняется с учетом культурной среды
`public static int Compare(string strA, int indexA, string strB, int indexB, int length, StringComparison comparisonType)`	Сравнивает части строк *strA* и *strB*. Сравнение начинается со строковых элементов *strA*[*indexA*] и *strB*[*indexB*] и включает количество символов, определяемых параметром *length*. Метод возвращает положительное значение, если часть строки *strA* больше части строки *strB*; отрицательное значение, если часть строки *strA* меньше части строки *strB*; и нуль, если сравниваемые части строк *strA* и *strB* равны. Параметр *comparisonType* определяет конкретный способ сравнения строк
`public static int Compare(string strA, int indexA, string strB, int indexB, int length, bool ignoreCase, CultureInfo culture)`	Сравнивает части строк *strA* и *strB*, используя информацию о культурной среде, определяемую параметром *culture*. Сравнение начинается со строковых элементов *strA*[*indexA*] и *strB*[*indexB*] и включает количество символов, определяемых параметром *length*. Метод возвращает положительное значение, если часть строки *strA* больше части строки *strB*; отрицательное значение, если часть строки *strA* меньше части строки *strB*; и нуль, если сравниваемые части строк *strA* и *strB* равны. Если параметр *ignoreCase* принимает логическое значение `true`, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Класс `CultureInfo` определен в пространстве имен `System.Globalization`

Продолжение табл. 22.1

Метод	Назначение
`public static int Compare(string strA, string strB, CultureInfo culture, CompareOptions options)`	Сравнивает строку `strA` со строкой `strB`, используя информацию о культурной среде, обозначаемую параметром `culture`, а также варианты сравнения, передаваемые в качестве параметра `options`. Возвращает положительное значение, если строка `strA` больше строки `strB`; отрицательное значение, если строка `strA` меньше строки `strB`; и нуль, если строки `strA` и `strB` равны. Классы `CultureInfo` и `CompareOptions` определены в пространстве имен `System.Globalization`
`public static int Compare(string strA, int indexA, string strB, int indexB, int length, CultureInfo culture, CompareOptions options)`	Сравнивает части строк `strA` и `strB`, используя информацию о культурной среде, обозначаемую параметром `culture`, а также варианты сравнения, передаваемые в качестве параметра `options`. Сравнение начинается со строковых элементов `strA[indexA]` и `strB[indexB]` и включает количество символов, определяемых параметром `length`. Метод возвращает положительное значение, если часть строки `strA` больше части строки `strB`; отрицательное значение, если часть строки `strA` меньше части строки `strB`; и нуль, если сравниваемые части строк `strA` и `strB` равны. Классы `CultureInfo` и `CompareOptions` определены в пространстве имен `System.Globalization`
`public static int CompareOrdinal(string strA, string strB)`	Сравнивает строку `strA` со строкой `strB` независимо от культурной среды, языка и региональных стандартов. Возвращает положительное значение, если строка `strA` больше строки `strB`; отрицательное значение, если строка `strA` меньше строки `strB`; и нуль, если строки `strA` и `strB` равны
`public static int CompareOrdinal(string strA, int indexA, string strB, int indexB, int count)`	Сравнивает части строк `strA` и `strB` независимо от культурной среды, языка и региональных стандартов. Сравнение начинается со строковых элементов `strA[indexA]` и `strB[indexB]` и включает количество символов, определяемых параметром `count`. Метод возвращает положительное значение, если часть строки `strA` больше части строки `strB`; отрицательное значение, если часть строки `strA` меньше части строки `strB`; и нуль, если сравниваемые части строк `strA` и `strB` равны
`public int CompareTo(object value)`	Сравнивает вызывающую строку со строковым представлением объекта `value`. Возвращает положительное значение, если вызывающая строка больше строки `value`; отрицательное значение, если вызывающая строка меньше строки `value`; и нуль, если сравниваемые строки равны
`public int CompareTo(string strB)`	Сравнивает вызывающую строку со строкой `strB`. Возвращает положительное значение, если вызывающая строка больше строки `strB`; отрицательное значение, если вызывающая строка меньше строки `strB`; и нуль, если сравниваемые строки равны

Окончание табл. 22.1

Метод	Назначение
`public override bool Equals(object obj)`	Возвращает логическое значение `true`, если вызывающая строка содержит ту же последовательность символов, что и строковое представление объекта `obj`. Выполняется порядковое сравнение с учетом регистра, но без учета культурной среды
`public bool Equals(string value)`	Возвращает логическое значение `true`, если вызывающая строка содержит ту же последовательность символов, что и строка `value`. Выполняется порядковое сравнение с учетом регистра, но без учета культурной среды
`public bool Equals(string value, StringComparison comparisonType)`	Возвращает логическое значение `true`, если вызывающая строка содержит ту же последовательность символов, что и строка `value`. Параметр `comparisonType` определяет конкретный способ сравнения строк
`public static bool Equals(string a, string b)`	Возвращает логическое значение `true`, если строка `a` содержит ту же последовательность символов, что и строка `b`. Выполняется порядковое сравнение с учетом регистра, но без учета культурной среды
`public static bool Equals(string a, string b, StringComparison comparisonType)`	Возвращает логическое значение `true`, если строка `a` содержит ту же последовательность символов, что и строка `b`. Параметр `comparisonType` определяет конкретный способ сравнения строк

Тип `StringComparison` представляет собой перечисление, в котором определяются значения, приведенные в табл. 22.2. Используя эти значения, можно организовать сравнение строк, удовлетворяющее потребностям конкретного приложения. Следовательно, добавление параметра типа `StringComparison` расширяет возможности метода `Compare()` и других методов сравнения, например, `Equals()`. Это дает также возможность однозначно указывать способ предполагаемого сравнения строк. В силу имеющих отличий между сравнением строк с учетом культурной среды и порядковым сравнением очень важно быть предельно точным в этом отношении. Именно по этой причине в примерах программ, приведенных в данной книге, параметр типа `StringComparison` явно указывается в вызовах тех методов, в которых он поддерживается.

Таблица 22.2. Значения, определяемые в перечислении `StringComparison`

Значение	Описание
`CurrentCulture`	Сравнение строк производится с использованием текущих настроек параметров культурной среды
`CurrentCultureIgnoreCase`	Сравнение строк производится с использованием текущих настроек параметров культурной среды, но без учета регистра
`InvariantCulture`	Сравнение строк производится с использованием неизменяемых, т.е. универсальных данных о культурной среде

Окончание табл. 22.2

Значение	Описание
`InvariantCultureIngoreCase`	Сравнение строк производится с использованием неизменяемых, т.е. универсальных данных о культурной среде и без учета регистра
`Ordinal`	Сравнение строк производится с использованием порядковых значений символов в строке. При этом лексикографический порядок может нарушиться, а условные обозначения, принятые в отдельной культурной среде, игнорируются
`OrdinalIgnoreCase`	Сравнение строк производится с использованием порядковых значений символов в строке, но без учета регистра. При этом лексикографический порядок может нарушиться, а условные обозначения, принятые в отдельной культурной среде, игнорируются

В любом случае метод `Compare()` возвращает отрицательное значение, если первая сравниваемая строка оказывается меньше второй; положительное значение, если первая сравниваемая строка больше второй; и наконец, нуль, если обе сравниваемые строки равны. Несмотря на то что метод `Compare()` возвращает нуль, если сравниваемые строки равны, для определения равенства символьных строк, как правило, лучше пользоваться методом `Equals()` или же оператором `==`. Дело в том, что метод `Compare()` определяет равенство сравниваемых строк на основании порядка их сортировки. Так, если выполняется сравнение строк с учетом культурной среды, то обе строки могут оказаться одинаковыми по порядку их сортировки, но не равными по существу. По умолчанию равенство строк определяется в методе `Equals()`, исходя из порядковых значений символов и без учета культурной среды. Следовательно, по умолчанию обе строки сравниваются в этом методе на абсолютное, посимвольное равенство подобно тому, как это делается в операторе `==`.

Несмотря на большую универсальность метода `Compare()`, для простого порядкового сравнения символьных строк проще пользоваться методом `CompareOrdinal()`. И наконец, следует иметь в виду, что метод `CompareTo()` выполняет сравнение строк только с учетом культурной среды. На момент написания этой книги отсутствовали перегружаемые варианты этого метода, позволявшие указывать другой способ сравнения символьных строк.

В приведенной ниже программе демонстрируется применение методов `Compare()`, `Equals()`, `CompareOrdinal()`, а также операторов `==` и `!=` для сравнения символьных строк. Обратите внимание на то, что два первых примера сравнения наглядно демонстрируют отличия между сравнением строк с учетом культурной среды и порядковым сравнением в англоязычной среде.

```
// Продемонстрировать разные способы сравнения символьных строк.

using System;

class CompareDemo {
  static void Main() {
    string str1 = "alpha";
    string str2 = "Alpha";
```

```
string str3 = "Beta";
string str4 = "alpha";
string str5 = "alpha, beta";
int result;

// Сначала продемонстрировать отличия между сравнением строк
// с учетом культурной среды и порядковым сравнением.
result = String.Compare(str1, str2, StringComparison.CurrentCulture);
Console.Write("Сравнение строк с учетом культурной среды: ");
if(result < 0)
  Console.WriteLine(str1 + " меньше " + str2);
else if(result > 0)
  Console.WriteLine(str1 + " больше " + str2);
else
  Console.WriteLine(str1 + " равно " + str2);

result = String.Compare(str1, str2, StringComparison.Ordinal);
Console.Write("Порядковое сравнение строк: ");
if(result < 0)
 Console.WriteLine(str1 + " меньше " + str2);
else if(result > 0)
 Console.WriteLine(str1 + " больше " + str2);
else
  Console.WriteLine(str1 + " равно " + str4);

// Использовать метод CompareOrdinal().
result = String.CompareOrdinal(str1, str2);
Console.Write("Сравнение строк методом CompareOrdinal():\n");
if(result < 0)
  Console.WriteLine(str1 + " меньше " + str2);
else if(result > 0)
  Console.WriteLine(str1 + " больше " + str2);
else
  Console.WriteLine(str1 + " равно " + str4);

Console.WriteLine();

// Определить равенство строк с помощью оператора = = .
// Это порядковое сравнение символьных строк.
 if(str1 == str4) Console.WriteLine(str1 + " == " + str4);

// Определить неравенство строк с помощью оператора !=.
if(str1 != str3) Console.WriteLine(str1 + " != " + str3);
if(str1 != str2) Console.WriteLine(str1 + " != " + str2);

Console.WriteLine();

// Выполнить порядковое сравнение строк без учета регистра,
// используя метод Equals().
if(String.Equals(str1, str2, StringComparison.OrdinalIgnoreCase))
  Console.WriteLine("Сравнение строк методом Equals() с " +
                    "параметром OrdinalIgnoreCase:\n" +
                     str1 + " равно " + str2);
```

```
    Console.WriteLine();

    // Сравнить части строк.
     if(String.Compare(str2, 0, str5, 0, 3,
       StringComparison.CurrentCulture) > 0) {
       Console.WriteLine("Сравнение строк с учетом текущей культурной среды:" +
                         "\n3 первых символа строки " + str2 +
                         " больше, чем 3 первых символа строки " + str5);
   }
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Сравнение строк с учетом культурной среды: alpha меньше Alpha
Порядковое сравнение строк: alpha больше Alpha
Сравнение строк методом CompareOrdinal():
alpha больше Alpha

alpha == alpha
alpha != Beta
alpha != Alpha

Сравнение строк методом Equals() с параметром OrdinalIgnoreCase:
alpha равно Alpha

Сравнение строк с учетом текущей культурной среды:
3 первых символа строки Alpha больше, чем 3 первых символа строки alpha, beta
```

Сцепление строк

Строки можно сцеплять, т.е. объединять вместе, двумя способами. Во-первых, с помощью оператора +, как было показано в главе 7. И во-вторых, с помощью одного из методов сцепления, определенных в классе `String`. Конечно, для этой цели проще всего воспользоваться оператором +, тем не менее методы сцепления служат неплохой альтернативой такому подходу.

Метод, выполняющий сцепление строк, называется `Concat()`. Ниже приведена одна из самых распространенных его форм.

```
public static string Concat(string str0, string str1)
```

Этот метод возвращает строку, состоящую из строки *str1*, присоединяемой путем сцепления в конце строки *str0*. Ниже приведена еще одна форма метода `Concat()`, в которой сцепляются три строки.

```
public static string Concat(string str0, string str1,string str2)
```

В данной форме метод `Concat()` возвращает строку, состоящую из последовательно сцепленных строк *str0*, *str1* и *str2*.

Имеется также форма метода `Concat()`, в которой сцепляются четыре строки.

```
public static string Concat(string str0, string str1, string str2, string str3)
```

В этой форме метод `Concat()` возвращает строку, состоящую из четырех последовательно сцепленных строк.

А в приведенной ниже еще одной форме метода `Concat()` сцепляется произвольное количество строк:

```
public static string Concat(params string[] values)
```

где *values* обозначает переменное количество аргументов, сцепляемых для получения возвращаемого результата. Если в этой форме метода `Concat()` допускается сцепление произвольного количества строк, то зачем нужны все остальные его формы? Они существуют ради повышения эффективности. Ведь передача методу от одного до четырех аргументов оказывается намного эффективнее, чем использование для этой цели переменного списка аргументов.

В приведенном ниже примере программы демонстрируется применение метода `Concat()` в форме с переменным списком аргументов.

```
// Продемонстрировать применение метода Concat().

using System;

class ConcatDemo {
  static void Main() {
    string result = String.Concat("Это ", "тест ", "метода ",
                                  "сцепления ", "строк ",
                                  "из класса ", "String." );
    Console.WriteLine("Результат: " + result);
}
}
```

Эта программа дает следующий результат.

```
Результат: Это тест метода сцепления строк из класса String.
```

Кроме того, существуют варианты метода `Concat()`, в которых он принимает в качестве параметров ссылки на объекты, а не на строки. В этих вариантах метод `Concat()` получает строковые представления вызывающих объектов, а возвращает объединенную строку, сцепленную из этих представлений. (Строковые представления объектов получаются с помощью метода `ToString()`, вызываемого для этих объектов.) Ниже приведены все подобные варианты и формы метода `Concat()`.

```
public static string Concat(object arg0)
public static string Concat(object arg0, object arg1)
public static string Concat(object arg0, object arg1, object arg2)
public static string Concat(object arg0, object arg1, object arg2, object arg3)
public static string Concat(params object[] args)
```

В первой форме метод `Concat()` возвращает строку, эквивалентную объекту *arg0*, а в остальных формах — строку, получаемую в результате сцепления всех аргументов данного метода. Объектные формы метода `Concat()`, т.е. относящиеся к типу `object`, очень удобны, поскольку они исключают получение вручную строковых представлений объектов перед их сцеплением. В приведенном ниже примере программы наглядно демонстрируется польза от подобных форм метода `Concat()`.

```
// Продемонстрировать применение объектной формы метода Concat().

using System;

class MyClass {
```

```
  public static int Count = 0;

  public MyClass() { Count++; }
}

class ConcatDemo {
  static void Main() {
    string result = String.Concat("значение равно " + 19);
    Console.WriteLine("Результат: " + result);

    result = String.Concat("привет ", 88, " ", 20.0,
                           " ", false, " ", 23.45M);
    Console.WriteLine("Результат: " + result);

    MyClass mc = new MyClass();

    result = String.Concat(mc, " текущий счет равен ",
                           MyClass.Count);
    Console.WriteLine("Результат: " + result);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Результат: значение равно 19
Результат: привет 88 20 False 23.45
Результат: MyClass текущий счет равен 1
```

В данном примере метод `Concat()` сцепляет строковые представления различных типов данных. Для каждого аргумента этого метода вызывается соответствующий метод `ToString()`, с помощью которого получается строковое представление аргумента. Следовательно, в следующем вызове метода `Concat()`:

```
string result = String.Concat("значение равно " + 19);
```

метод `Int32.ToString()` вызывается для получения строкового представления целого значения 19, а затем метод `Concat()` сцепляет строки и возвращает результат.

Обратите также внимание на применение объекта определяемого пользователем класса `MyClass` в следующем вызове метода `Concat()`.

```
result = String.Concat(mc, " текущий счет равен ",
                       MyClass.Count);
```

В данном случае возвращается строковое представление объекта типа `MyClass`, сцепленное с указываемой строкой. По умолчанию это просто имя класса. Но если переопределить метод `ToString()`, то вместо строки с именем класса `MyClass` может быть возвращена другая строка. В качестве упражнения попробуйте ввести в приведенный выше пример программы следующий фрагмент кода.

```
public override string ToString() {
  return "Объект типа MyClass";
}
```

В этом случае последняя строка результата выполнения программы будет выглядеть так, как показано ниже.

```
Результат: Объект типа MyClass текущий счет равен 1
```

В версию 4.0 среды .NET Framework добавлены еще две формы метода `Concat()`, приведенные ниже.

```
public static string Concat<T>(IEnumerable<T> values)
public static string Concat(IEnumerable<string> values)
```

В первой форме этого метода возвращается символьная строка, состоящая из сцепленных строковых представлений ряда значений, имеющихся в объекте, который обозначается параметром *values* и может быть объектом любого типа, реализующего интерфейс `IEnumerable<T>`. А во второй форме данного метода сцепляются строки, обозначаемые параметром *values*. (Следует, однако, иметь в виду, что если приходится выполнять большой объем операций сцепления символьных строк, то для этой цели лучше воспользоваться средствами класса `StringBuilder`.)

Поиск в строке

В классе `String` предоставляется немало методов для поиска в строке. С их помощью можно, например, искать в строке отдельный символ, строку, первое или последнее вхождение того и другого в строке. Следует, однако, иметь в виду, что поиск может осуществляться либо с учетом культурной среды либо порядковым способом.

Для обнаружения первого вхождения символа или подстроки в исходной строке служит метод `IndexOf()`. Для него определено несколько перегружаемых форм. Ниже приведена одна из форм для поиска первого вхождения символа в исходной строке.

```
public int IndexOf(char value)
```

В этой форме метода `IndexOf()` возвращается первое вхождение символа *value* в вызывающей строке. Если символ *value* в ней не найден, то возвращается значение –1. При таком поиске символа настройки культурной среды игнорируются. Следовательно, в данном случае осуществляется порядковый поиск первого вхождения символа.

Ниже приведены еще две формы метода `IndexOf()`, позволяющие искать первое вхождение одной строки в другой.

```
public int IndexOf(String value)
public int IndexOf(String value, StringComparison comparisonType)
```

В первой форме рассматриваемого здесь метода поиск первого вхождения строки, обозначаемой параметром *value*, осуществляется с учетом культурной среды. А во второй форме предоставляется возможность указать значение типа `StringComparison`, обозначающее способ поиска. В если искомая строка не найдена, то в обеих формах данного метода возвращается значение –1.

Для обнаружения последнего вхождения символа или строки в исходной строке служит метод `LastIndexOf()`. И для этого метода определено несколько перегружаемых форм. Ниже приведена одна из форм для поиска последнего вхождения символа в вызывающей строке.

```
public int LastIndexOf(char value)
```

В этой форме метода `LastIndexOf()` осуществляется порядковый поиск, а в итоге возвращается последнее вхождение символа *value* в вызывающей строке или же значение –1, если искомый символ не найден.

Ниже приведены еще две формы метода `LastIndexOf()`, позволяющие искать последнее вхождение одной строки в другой.

```
public int LastIndexOf(string value)
public int LastIndexOf(string value, StringComparison comparisonType)
```

В первой форме рассматриваемого здесь метода поиск последнего вхождения строки, обозначаемой параметром *value*, осуществляется с учетом культурной среды. А во второй форме предоставляется возможность указать значение типа `StringComparison`, обозначающее способ поиска. Если же искомая строка не найдена, то в обеих формах данного метода возвращается значение –1.

В классе `String` предоставляются еще два интересных метода поиска в строке: `IndexOfAny()` и `LastIndexOfAny()`. Оба метода обнаруживают первый символ, совпадающий с любым набором символов. Ниже приведены простейшие формы этих методов.

```
public int IndexOfAny(char[] anyOf)
public int LastIndexOfAny(char[] anyOf)
```

Метод `IndexOfAny()` возвращает индекс первого вхождения любого символа из массива *anyOf*, обнаруженного в вызывающей строке, а метод `LastIndexOfAny()` — индекс последнего вхождения любого символа из массива *anyOf*, обнаруженного в вызывающей строке. Если совпадение символов не обнаружено, то в обоих случаях возвращается значение –1. Кроме того, в обоих рассматриваемых здесь методах осуществляется порядковый поиск.

При обработке символьных строк нередко оказывается полезно знать, начинается ли строка заданной подстрокой или же оканчивается ею. Для этой цели служат методы `StartsWith()` и `EndsWith()`. Ниже приведены их простейшие формы.

```
public bool StartsWith(string value)
public bool EndsWith(string value)
```

Метод `StartsWith()` возвращает логическое значение `true`, если вызывающая строка начинается с подстроки, переданной ему в качестве аргумента *value*. А метод `EndsWith()` возвращает логическое значение `true`, если вызывающая строка оканчивается подстрокой, переданной ему в качестве аргумента *value*. В противном случае оба метода возвращают логическое значение `false`.

В обоих рассматриваемых здесь методах поиск осуществляется с учетом культурной среды. Для того чтобы указать конкретный способ поиска подстроки, можно воспользоваться приведенными ниже вариантами этих методов с дополнительным параметром типа `StringComparison`.

```
public bool StartsWith(string value, StringComparison comparisonType)
public bool EndsWith(string value, StringComparison comparisonType)
```

Оба варианта рассматриваемых здесь методов поиска действуют таким же образом, как и предыдущие их варианты. Но в то же время они позволяют явно указать конкретный способ поиска.

В приведенном ниже примере программы демонстрируется применение нескольких методов поиска в строке.

```
// Продемонстрировать поиск в строке.

using System;

class StringSearchDemo {
  static void Main() {
```

```
    string str = "C# обладает эффективными средствами обработки строк.";
    int idx;

    Console.WriteLine("Строка str: " + str);

    idx = str.IndexOf('о');
    Console.WriteLine("Индекс первого вхождения символа 'о': " + idx);

    idx = str.LastIndexOf('о');
    Console.WriteLine("Индекс последнего вхождения символа 'о': " + idx);

    idx = str.IndexOf("ми", StringComparison.Ordinal);
    Console.WriteLine("Индекс первого вхождения подстроки \"ми\": " + idx);

    idx = str.LastIndexOf("ми", StringComparison.Ordinal);
    Console.WriteLine("Индекс последнего вхождения подстроки \"ми\": " + idx);

    char[] chrs = { 'а', 'б', 'в' };
    idx = str.IndexOfAny(chrs);
    Console.WriteLine("Индекс первого вхождения символов " +
                      " 'а', 'б' или 'в': " + idx);

    if(str.StartsWith("C# обладает", StringComparison.Ordinal))
      Console.WriteLine("Строка str начинается с подстроки \"C# обладает\"");

    if(str.EndsWith("строк.", StringComparison.Ordinal))
      Console.WriteLine("Строка str оканчивается подстрокой \"строк.\"");
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Строка str: C# обладает эффективными средствами обработки строк.
Индекс первого вхождения символа 'о': 3
Индекс последнего вхождения символа 'о': 49
Индекс первого вхождения подстроки "ми": 22
Индекс последнего вхождения подстроки "ми": 33
Индекс первого вхождения символов 'а', 'б' или 'в': 4
Строка str начинается с подстроки "C# обладает"
Строка str оканчивается подстрокой "строк."
```

Во многих случаях полезным для поиска в строке оказывается метод `Contains()`. Его общая форма выглядит следующим образом.

```
public bool Contains(string value)
```

Метод `Contains()` возвращает логическое значение `true`, если вызывающая строка содержит подстроку, обозначаемую параметром *value*, в противном случае — логическое значение `false`. Поиск указываемой подстроки осуществляется порядковым способом. Этот метод особенно полезен, если требуется только выяснить, находится ли конкретная подстрока в другой строке. В приведенном ниже примере программы демонстрируется применение метода `Contains()`.

```
// Продемонстрировать применение метода Contains().

using System;
```

```
class ContainsDemo {
  static void Main() {
    string str = "C# сочетает эффективность с производительностью.";

    if(str.Contains("эффективность"))
      Console.WriteLine("Обнаружена подстрока \"эффективность\".");

    if(str.Contains("эффе"))
     Console.WriteLine("Обнаружена подстрока \"эффе\".");

    if(!str.Contains("эффективный"))
      Console.WriteLine("Подстрока \"эффективный\" не обнаружена.");
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Обнаружена подстрока "эффективность".
Обнаружена подстрока "эффе".
Подстрока "эффективный" не обнаружена.
```

Как следует из результата выполнения приведенной выше программы, метод `Contains()` осуществляет поиск на совпадение произвольной последовательности символов, а не только целых слов. Поэтому в вызывающей строке обнаруживается и подстрока `"эффективность"`, и подстрока `"эффе"`. Но поскольку в вызывающей строке отсутствует подстрока `"эффективный"`, то она и не обнаруживается.

У некоторых методов поиска в строке имеются дополнительные формы, позволяющие начинать поиск по указанному индексу или указывать пределы для поиска в строке. В табл. 22.3 сведены все варианты методов поиска в строке, которые поддерживаются в классе `String`.

Таблица 22.3. Методы поиска в строке, поддерживаемые в классе `String`

Метод	Назначение
`public bool Contains(string value)`	Возвращает логическое значение `true`, если вызывающая строка содержит подстроку *`value`*. Если же подстрока *`value`* не обнаружена, возвращается логическое значение `false`
`public bool EndsWith(string value)`	Возвращает логическое значение `true`, если вызывающая строка оканчивается подстрокой *`value`*. В противном случае возвращает логическое значение `false`
`public bool EndsWith(string value, StringComparison comparisonType)`	Возвращает логическое значение `true`, если вызывающая строка оканчивается подстрокой *`value`*. В противном случае возвращает логическое значение `false`. Параметр *`comparisonType`* определяет конкретный способ поиска
`public bool EndsWith(string value, bool ignoreCase, CultureInfo culture)`	Возвращает логическое значение `true`, если вызывающая строка оканчивается подстрокой *`value`*, иначе возвращает

Продолжение табл. 22.3

Метод	Назначение
	логическое значение `false`. Если параметр *`ignoreCase`* принимает логическое значение `true`, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Поиск осуществляется с использованием информации о культурной среде, обозначаемой параметром *`culture`*
`public int IndexOf(char value)`	Возвращает индекс первого вхождения символа *`value`* в вызывающей строке. Если искомый символ не обнаружен, то возвращается значение `-1`
`public int IndexOf(string value)`	Возвращает индекс первого вхождения подстроки *`value`* в вызывающей строке. Если искомая подстрока не обнаружена, то возвращается значение `-1`
`public int IndexOf(char value, int startIndex)`	Возвращает индекс первого вхождения символа *`value`* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *`startIndex`*. Метод возвращает значение `-1`, если искомый символ не обнаружен
`public int IndexOf(string value, int startIndex)`	Возвращает индекс первого вхождения подстроки *`value`* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *`startIndex`*. Метод возвращает значение `-1`, если искомая подстрока не обнаружена
`public int IndexOf(char value, int startIndex, int count)`	Возвращает индекс первого вхождения символа *`value`* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *`startIndex`*, и охватывает число элементов, определяемых параметром *`count`*. Метод возвращает значение `-1`, если искомый символ не обнаружен
`public int IndexOf(string value, int startIndex, int count)`	Возвращает индекс первого вхождения подстроки *`value`* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *`startIndex`*, и охватывает число элементов, определяемых параметром *`count`*. Метод возвращает значение `-1`, если искомая подстрока не обнаружена
`public int IndexOf(string value, StringComparison comparisonType)`	Возвращает индекс первого вхождения подстроки *`value`* в вызывающей строке.

Продолжение табл. 22.3

Метод	Назначение
	Параметр *comparisonType* определяет конкретный способ выполнения поиска. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int IndexOf(string value, int startIndex, StringComparison comparisonType)`	Возвращает индекс первого вхождения подстроки *value* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *startIndex*. Параметр *comparisonType* определяет конкретный способ выполнения поиска. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int IndexOf(string value, int startIndex, int count, StringComparison comparisonType)`	Возвращает индекс первого вхождения подстроки *value* в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *startIndex*, и охватывает число элементов, определяемых параметром *count*. Параметр *comparisonType* определяет конкретный способ выполнения поиска. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOf(char value)`	Возвращает индекс последнего вхождения символа *value* в вызывающей строке. Если искомый символ не обнаружен, возвращается значение -1
`public int IndexOfAny(char[] anyOf)`	Возвращает индекс первого вхождения любого символа из массива *anyOf*, обнаруженного в вызывающей строке. Метод возвращает значение -1, если не обнаружено совпадение ни с одним из символов из массива *anyOf*. Поиск осуществляется порядковым способом
`public int IndexOfAny(char[] anyOf, int startIndex)`	Возвращает индекс первого вхождения любого символа из массива *anyOf*, обнаруженного в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *startIndex* Метод возвращает значение -1, если не обнаружено совпадение ни с одним из символов из массива *anyOf*. Поиск осуществляется порядковым способом
`public int IndexOfAny(char[] anyOf, int startIndex, int count)`	Возвращает индекс первого вхождения любого символа из массива *anyOf*, обнаруженного в вызывающей строке. Поиск начинается с элемента, указываемого по индексу *startIndex*, и охватывает число

Продолжение табл. 22.3

Метод	Назначение
	элементов, определяемых параметром *count*. Метод возвращает значение -1, если не обнаружено совпадение ни с одним из символов из массива *anyOf*. Поиск осуществляется порядковым способом
`public int LastIndexOf(string value)`	Возвращает индекс последнего вхождения подстроки *value* в вызывающей строке. Если искомая подстрока не обнаружена, возвращается значение -1
`public int LastIndexOf(char value, int startIndex)`	Возвращает индекс последнего вхождения символа *value* в части вызывающей строки. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу *startIndex*, и заканчивая элементом с нулевым индексом. Метод возвращает значение -1, если искомый символ не обнаружен
`public int LastIndexOf(string value, int startIndex)`	Возвращает индекс последнего вхождения подстроки *value* в части вызывающей строки. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу *startIndex*, и заканчивая элементом с нулевым индексом. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOf(char value, int startIndex, int count)`	Возвращает индекс последнего вхождения символа *value* в части вызывающей строки. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу *startIndex*, и охватывает число элементов, определяемых параметром *count*. Метод возвращает значение -1, если искомый символ не обнаружен
`public int LastIndexOf(string value, int startIndex, int count)`	Возвращает индекс последнего вхождения подстроки *value* в части вызывающей строки. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу *startIndex*, и охватывает число элементов, определяемых параметром *count*. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOf(string value, StringComparison comparisonType)`	Возвращает индекс последнего вхождения подстроки `value` в вызывающей строке. Параметр `comparisonType` определяет конкретный способ выполнения поиска. Метод возвращает значение -1, если искомая подстрока не обнаружена

Продолжение табл. 22.3

Метод	Назначение
`public int LastIndexOf(string value, int startIndex, StringComparison comparisonType)`	Возвращает индекс последнего вхождения подстроки `value` в части вызывающей строки. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу `startIndex`, и заканчивая элементом с нулевым индексом. Параметр `comparisonType` определяет конкретный способ выполнения поиска. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOf(string value, int startIndex, int count, StringComparison comparisonType)`	Возвращает индекс последнего вхождения подстроки `value` в части вызывающей строки. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу `startIndex`, и охватывает число элементов, определяемых параметром `count`. Параметр `comparisonType` определяет конкретный способ выполнения поиска. Метод возвращает значение -1, если искомая подстрока не обнаружена
`public int LastIndexOfAny(char[] anyOf)`	Возвращает индекс последнего вхождения любого символа из массива `anyOf`, обнаруженного в вызывающей строке. Метод возвращает значение -1, если не обнаружено совпадение ни с одним из символов из массива `anyOf`. Поиск осуществляется порядковым способом
`public int LastIndexOfAny(char[] anyOf, int startIndex)`	Возвращает индекс последнего вхождения любого символа из массива `anyOf`, обнаруженного в вызывающей строке. Поиск начинается в обратном порядке с элемента, указываемого по индексу `startIndex`, и заканчивая элементом с нулевым индексом. Метод возвращает значение -1, если не обнаружено совпадение ни с одним из символов из массива `anyOf`. Поиск осуществляется порядковым способом
`public int LastIndexOfAny(char[] anyOf, int startIndex, int count)`	Возвращает индекс последнего вхождения любого символа из массива `anyOf`, обнаруженного в вызывающей строке. Поиск осуществляется в обратном порядке, начиная с элемента, указываемого по индексу `startIndex`, и охватывает число элементов, определяемых параметром `count`. число элементов, определяемых параметром `count`. Метод возвращает значение -1,

Окончание табл. 22.3

Метод	Назначение
	если не обнаружено совпадение ни с одним из символов из массива *anyOf*. Поиск осуществляется порядковым способом
`public bool StartsWith(string value)`	Возвращает логическое значение `true`, если вызывающая строка начинается с подстроки *value*. В противном случае возвращается логическое значение `false`
`public bool StartsWith(string value, StringComparison comparisonType)`	Возвращает логическое значение `true`, если вызывающая строка начинается с подстроки *value*. В противном случае возвращается логическое значение `false`. Параметр *comparisonType* определяет конкретный способ выполнения поиска
`public bool StartsWith(string value, bool ignoreCase, CultureInfo culture)`	Возвращает логическое значение `true`, если вызывающая строка начинается с подстроки *value*. В противном случае возвращается логическое значение `false`. Если параметр *ignoreCase* принимает логическое значение `true`, то при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. Поиск осуществляется с использованием информации о культурной среде, обозначаемой параметром *culture*

Разделение и соединение строк

К основным операциям обработки строк относятся разделение и соединение. При *разделении* строка разбивается на составные части, а при соединении строка составляется из отдельных частей. Для разделения строк в классе `String` определен метод `Split()`, а для соединения — метод `Join()`.

Существует несколько вариантов метода `Split()`. Ниже приведены две формы этого метода, ставшие наиболее часто используемыми, начиная с версии C# 1.0.

```
public string[ ] Split(params char[ ] separator)
public string[ ] Split(params char[ ] separator, int count)
```

В первой форме метода `Split()` вызывающая строка разделяется на составные части. В итоге возвращается массив, содержащий подстроки, полученные из вызывающей строки. Символы, ограничивающие эти подстроки, передаются в массиве *separator*. Если массив *separator* пуст или ссылается на пустую строку, то в качестве разделителя подстрок используется пробел. А во второй форме данного метода возвращается количество подстрок, определяемых параметром *count*.

Существует несколько форм метода `Join()`. Ниже приведены две формы, ставшие доступными, начиная с версии 2.0 среды .NET Framework.

```
public static string Join(string separator, string[] value)
public static string Join(string separator, string[] value,
                          int startIndex, int count)
```

В первой форме метода `Join()` возвращается строка, состоящая из сцепляемых подстрок, передаваемых в массиве *value*. Во второй форме также возвращается строка, состоящая из подстрок, передаваемых в массиве `value`, но они сцепляются в определенном количестве *count*, начиная с элемента массива *value*[*startIndex*]. В обеих формах каждая последующая строка отделяется от предыдущей разделительной строкой, определяемой параметром *separator*.

В приведенном ниже примере программы демонстрируется применение методов `Split()` и `Join()`.

```
// Разделить и соединить строки.

using System;

class SplitAndJoinDemo {
  static void Main() {
    string str = "Один на суше, другой на море.";
    char[] seps = {' ', '.', ',' };

    // Разделить строку на части.
    string[] parts = str.Split(seps);
    Console.WriteLine("Результат разделения строки: ");
    for(int i=0; i < parts.Length; i++)
      Console.WriteLine(parts[i]);

    // А теперь соединить части строки.
    string whole = String.Join(" | ", parts);
    Console.WriteLine("Результат соединения строки: ");
    Console.WriteLine(whole);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Результат разделения строки:
Один
на
суше

другой
на
море

Результат соединения строки:
Один | на | суше |  | другой | на | море
```

Обратите внимание на пустую строку между словами "суше" и "другой". Дело в том, что в исходной строке после слова "суше" следует запятая и пробел, как в подстроке "суше, другой". Но запятая и пробел указаны в качестве разделителей. Поэтому при разделении данной строки между двумя разделителями (запятой и пробелом) оказывается пустая строка.

Существует ряд других форм метода `Split()`, принимающих параметр типа `StringSplitOptions`. Этот параметр определяет, являются ли пустые строки частью разделяемой в итоге строки. Ниже приведены все эти формы метода `Split()`.

```
public string[] Split(params char[] separator,StringSplitOptions options)
public string[] Split(string[] separator, StringSplitOptions options)
public string[] Split(params char[] separator, int count,
                      StringSplitOptions options)
public string[] Split(string[]separator, int count,
                      StringSplitOptions options)
```

В двух первых формах метода `Split()` вызывающая строка разделяется на части и возвращается массив, содержащий подстроки, полученные из вызывающей строки. Символы, разделяющие эти подстроки, передаются в массиве *separator*. Если массив *separator* пуст, то в качестве разделителя используется пробел. А в третьей и четвертой формах данного метода возвращается количество строк, ограничиваемое параметром *count*. Но во всех формах параметр *options* обозначает конкретный способ обработки пустых строк, которые образуются в том случае, если два разделителя оказываются рядом. В перечислении `StringSplitOptions` определяются только два значения: `None` и `RemoveEmptyEntries`. Если параметр *options* принимает значение `None`, то пустые строки включаются в конечный результат разделения исходной строки, как показано в предыдущем примере программы. А если параметр *options* принимает значение `RemoveEmptyEntries`, то пустые строки исключаются из конечного результата разделения исходной строки.

Для того чтобы стали понятнее последствия исключения пустых строк, попробуем заменить в предыдущем примере программы строку кода

```
string[] parts = str.Split(seps);
```

следующим фрагментом кода.

```
string[] parts = str.Split(seps, StringSplitOptions.RemoveEmptyEntries);
```

При выполнении данной программы получится следующий результат.

```
Результат разделения строки:
Один
на
суше
другой
на
море
Результат соединения строки:
Один | на | суше | другой | на | море
```

Как видите, пустая строка, появлявшаяся ранее из-за того, что после слова "суше" следовали запятая и пробел, теперь исключена.

Разделение является очень важной процедурой обработки строк, поскольку с его помощью нередко получают отдельные *лексемы*, составляющие исходную строку. Так, в программе ведения базы данных может возникнуть потребность разделить с помощью метода `Split()` строку запроса "показать все остатки больше 100" на отдельные части, включая подстроки "показать" и "100". В процессе разделения исключаются разделители, поэтому в итоге получается подстрока "показать" (без начальных и конечных пробелов), а не подстрока " показать". Этот принцип демонстрируется

в приведенном ниже примере программы, где строки, содержащие такие бинарные математические операции, как 10 + 5, преобразуются в лексемы, а затем эти операции выполняются и выводится конечный результат.

```
// Преобразовать строки в лексемы.

using System;

class TokenizeDemo {
  static void Main() {
    string[] input = {
                      "100 + 19",
                      "100 / 3,3",
                      "-3 * 9",
                      "100 - 87"
                     };
    char[] seps = {' '};

    for(int i=0; i < input.Length; i++) {
      // разделить строку на части
      string[] parts = input[i].Split(seps);
      Console.Write("Команда: ");
      for(int j=0; j < parts.Length; j++)
        Console.Write(parts[j] + " ");

      Console.Write(", результат: ");
      double n = Double.Parse(parts[0]);
      double n2 = Double.Parse(parts[2]);

      switch(parts[1]) {
        case "+":
          Console.WriteLine(n + n2);
          break;
        case "-":
          Console.WriteLine(n - n2);
          break;
        case "*":
          Console.WriteLine(n * n2);
          break;
        case "/":
          Console.WriteLine(n / n2);
          break;
      }
    }
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Команда: 100 + 19 , результат: 119
Команда: 100 / 3,3 , результат: 30,3030303030303
Команда: -3 * 9 , результат: -27
Команда: 100 - 87 , результат: 13
```

Начиная с версии 4.0, в среде .NET Framework стали доступными следующие дополнительные формы метода `Join()`.

```
public static string Join(string separator, params object[] values)
public static string Join(string separator, IEnumerable<string>[] values)
public static string Join<T>(string separator, IEnumerable<T>[] values)
```

В первой форме рассматриваемого здесь метода возвращается строка, содержащая строковое представление объектов из массива `values`. Во второй форме возвращается строка, содержащая результат сцепления коллекции строк, обозначаемой параметром `values`. И в третьей форме возвращается строка, содержащая результат сцепления строковых представлений объектов из коллекции, обозначаемой параметром `values`. Во всех трех случаях каждая предыдущая строка отделяется от последующей разделителем, определяемым параметром `separator`.

Заполнение и обрезка строк

Иногда в строке требуется удалить начальные и конечные пробелы. Такая операция называется *обрезкой* и нередко требуется в командных процессорах. Например, программа ведения базы данных способна распознавать команду "print", но пользователь может ввести эту команду с одним или несколькими начальными и конечными пробелами. Поэтому перед распознаванием введенной команды необходимо удалить все подобные пробелы. С другой стороны, строку иногда требуется заполнить пробелами, чтобы она имела необходимую минимальную длину. Так, если подготавливается вывод результатов в определенном формате, то каждая выводимая строка должна иметь определенную длину, чтобы сохранить выравнивание строк. Для упрощения подобных операций в C# предусмотрены соответствующие методы.

Для обрезки строк используется одна из приведенных ниже форм метода `Trim()`.

```
public string Trim()
public string Trim(params char[] trimChars)
```

В первой форме метода `Trim()` из вызывающей строки удаляются начальные и конечные пробелы. А во второй форме этого метода удаляются начальные и конечные вхождения в вызывающей строке символов из массива `trimChars`. В обеих формах возвращается получающаяся в итоге строка.

Строку можно заполнить символами слева или справа. Для заполнения строки слева служат такие формы метода `PadLeft()`.

```
public string PadLeft(int totalWidth)
public string PadLeft(int totalWidth, char paddingChar)
```

В первой форме метода `PadLeft()` вводятся пробелы с левой стороны вызывающей строки, чтобы ее общая длина стала равной значению параметра `totalWidth`. А во второй форме данного метода символы, обозначаемые параметром `paddingChar`, вводятся с левой стороны вызывающей строки, чтобы ее общая длина стала равной значению параметра `totalWidth`. В обеих формах возвращается получающаяся в итоге строка. Если значение параметра `totalWidth` меньше длины вызывающей строки, то возвращается копия неизмененной вызывающей строки.

Для заполнения строки справа служат следующие формы метода `PadRight()`.

```
public string PadRight(int totalWidth)
public string PadRight(int totalWidth, char paddingChar)
```

В первой форме метода `PadLeft()` вводятся пробелы с правой стороны вызывающей строки, чтобы ее общая длина стала равной значению параметра `totalWidth`.

А во второй форме данного метода символы, обозначаемые параметром *paddingChar*, вводятся с правой стороны вызывающей строки, чтобы ее общая длина стала равной значению параметра *totalWidth*. В обеих формах возвращается получающаяся в итоге строка. Если значение параметра *totalWidth* меньше длины вызывающей строки, то возвращается копия неизмененной вызывающей строки.

В приведенном ниже примере программы демонстрируются обрезка и заполнение строк.

```
// Пример обрезки и заполнения строк.

using System;

class TrimPadDemo {
  static void Main() {
    string str = "тест";

    Console.WriteLine("Исходная строка: " + str);

    // Заполнить строку пробелами слева.
    str = str.PadLeft(10);
    Console.WriteLine("|" + str + "|");

    // Заполнить строку пробелами справа.
    str = str.PadRight(20);
    Console.WriteLine("|" + str + "|");

    // Обрезать пробелы.
    str = str.Trim();
    Console.WriteLine("|" + str + "|");

    // Заполнить строку символами # слева.
    str = str.PadLeft(10, '#');
    Console.WriteLine("|" + str + "|");

    // Заполнить строку символами # справа.
    str = str.PadRight(20, '#');
    Console.WriteLine("|" + str + "|");

    // Обрезать символы #.
    str = str.Trim('#');
    Console.WriteLine("|" + str + "|");
  }
}
```

Эта программа дает следующий результат.

```
Исходная строка: тест
|      тест|
|      тест          |
|тест|
|######тест|
|######тест##########|
|тест|
```

Вставка, удаление и замена строк

Для вставки одной строки в другую служит приведенный ниже метод `Insert()`:

```
public string Insert(int startIndex, string value)
```

где *value* обозначает строку, вставляемую в вызывающую строку по индексу *startIndex*. Метод возвращает получившуюся в итоге строку.

Для удаления части строки служит метод `Remove()`. Ниже приведены две его формы.

```
public string Remove(int startIndex)
public string Remove(int startIndex, int count)
```

В первой форме метода `Remove()` удаление выполняется, начиная с места, указываемого по индексу *startIndex*, и продолжается до конца строки. А во второй форме данного метода из строки удаляется количество символов, определяемое параметром *count*, начиная с места, указываемого по индексу *startIndex*. В обеих формах возвращается получающаяся в итоге строка.

Для замены части строки служит метод `Replace()`. Ниже приведены две его формы.

```
public string Replace(char oldChar, char newChar)
public string Replace(string oldValue, string newValue)
```

В первой форме метода `Replace()` все вхождения символа *oldChar* в вызывающей строке заменяются символом *newChar*. А во второй форме данного метода все вхождения строки *oldValue* в вызывающей строке заменяются строкой *newValue*. В обеих формах возвращается получающаяся в итоге строка.

В приведенном ниже примере демонстрируется применение методов `Insert()`, `Remove()` и `Replace()`.

```
// Пример вставки, замены и удаления строк.

using System;

class InsRepRevDemo {
  static void Main() {
    string str = "Это тест";

    Console.WriteLine("Исходная строка: " + str);

    // Вставить строку.
    str = str.Insert(4, "простой ");
    Console.WriteLine(str);

    // Заменить строку.
    str = str.Replace("простой", "непростой ");
    Console.WriteLine(str);

    // Заменить символы в строке
    str = str.Replace('т', 'Х');
    Console.WriteLine(str);

    // Удалить строку.
    str = str.Remove(4, 5);
    Console.WriteLine(str);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Исходная строка: Это тест
Это простой тест
Это непростой тест
ЭХо непросХой ХесХ
ЭХо сХой ХесХ
```

Смена регистра

В классе `String` предоставляются два удобных метода, позволяющих сменить регистр букв в строке, — `ToUpper()` и `ToLower()`. Ниже приведены их простейшие формы.

```
public string ToLower()
public string ToUpper()
```

Метод `ToLower()` делает строчными все буквы в вызывающей строке, а метод `ToUpper()` делает их прописными. В обоих случаях возвращается получающаяся в итоге строка. Имеются также следующие формы этих методов, в которых можно указывать информацию о культурной среде и способы преобразования символов.

```
public string ToLower(CultureInfo culture)
public string ToUpper(CultureInfo culture)
```

С помощью этих форм можно избежать неоднозначности в исходном коде по отношению к правилам смены регистра. Именно для таких целей эти формы и рекомендуется применять.

Кроме того, имеются следующие методы `ToUpperInvariant()` и `ToLowerInvariant()`.

```
public string ToUpperInvariant()
public string ToLowerInvariant()
```

Эти методы аналогичны методам `ToUpper()` и `ToLower()`, за исключением того, что они изменяют регистр букв в вызывающей строке безотносительно к настройкам культурной среды.

Применение метода `Substring()`

Для получения части строки служит метод `Substring()`. Ниже приведены две его формы.

```
public string Substring(int startIndex)
public string Substring(int startIndex, int length)
```

В первой форме метода `Substring()` подстрока извлекается, начиная с места, обозначаемого параметром *`startIndex`*, и до конца вызывающей строки. А во второй форме данного метода извлекается подстрока, состоящая из количества символов, определяемых параметром *`length`*, начиная с места, обозначаемого параметром *`startIndex`*. В обеих формах возвращается получающаяся в итоге подстрока.

В приведенном ниже примере программы демонстрируется применение метода `Substring()`.

```
// Использовать метод Substring().

using System;

class SubstringDemo {
  static void Main() {
    string str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    Console.WriteLine("Строка str: " + str);

    Console.Write("Подстрока str.Substring(15): ");
    string substr = str.Substring(15);
    Console.WriteLine(substr);

    Console.Write("Подстрока str.Substring(0, 15): ");
    substr = str.Substring(0, 15);
    Console.WriteLine(substr);
  }
}
```

Эта программа дает следующий результат.

```
Строка str: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Подстрока str.Substring(15): PQRSTUVWXYZ
Подстрока' str.Substring(0, 15): ABCDEFGHIJKLMNO
```

Методы расширения класса `String`

Как упоминалось ранее, в классе `String` реализуется обобщенный интерфейс `IEnumerable<T>`. Это означает, что, начиная с версии C# 3.0, для объекта класса `String` можно вызывать методы расширения, определенные в классах `Enumerable` и `Queryable`, которые находятся в пространстве имен `System.Linq`. Эти методы расширения служат главным образом для поддержки LINQ, хотя некоторые из них могут использоваться в иных целях, в том числе и в определенных видах обработки строк. Подробнее о методах расширения см. в главе 19.

Форматирование

Когда данные встроенных в C# типов, например `int` или `double`, требуется отобразить в удобочитаемой форме, приходится формировать их строковое представление. Несмотря на то что в C# для такого представления данных автоматически предоставляется формат, используемый по умолчанию, имеется также возможность указать выбранный формат вручную. Так, в части I этой книги было показано, что числовые данные можно выводить в формате выбранной денежной единицы. Для форматирования данных числовых типов в C# предусмотрен целый ряд методов, включая методы `Console.WriteLine()`, `String.Format()` и `ToString()`. Во всех этих методах применяется один и тот же подход к форматированию. Поэтому освоив один из них, вы сможете без особого труда применять и другие.

Общее представление о форматировании

Форматирование осуществляется с помощью двух компонентов: *спецификаторов формата* и *поставщиков формата*. Конкретная форма строкового представления

отдельного значения зависит от спецификатора формата. Следовательно, спецификатор формата определяет, в какой именно удобочитаемой форме будут представлены данные. Например, для вывода числового значения в экспоненциальном представлении (т.е. в виде мантиссы и порядка числа) используется спецификатор формата `E`.

Как правило, конкретный формат значения зависит от культурных и языковых особенностей локализации программного обеспечения. Например, в Соединенных Штатах Америки денежные суммы указываются в долларах, а в странах ЕС — в евро. Для учета культурных и языковых отличий в C# предусмотрены поставщики формата. В частности, поставщик формата определяет порядок интерпретации спецификатора формата. Поставщик формата создается путем реализации интерфейса `IFormatProvider`, в котором определяется метод `GetFormat()`. Для всех встроенных числовых типов и многих других типов данных в среде .NET Framework предопределены соответствующие поставщики формата. Вообще говоря, данные можно отформатировать, не указывая конкретный поставщик формата, поэтому поставщики формата не рассматриваются далее в этой книге.

Для того чтобы отформатировать данные, достаточно включить спецификатор формата в метод, поддерживающий форматирование. О применении спецификаторов формата речь уже шла в главе 3, тем не менее к этому вопросу стоит вернуться вновь. Применение спецификаторов формата рассматривается далее на примере метода `Console.WriteLine()`, хотя аналогичный подход применим и к другим методам, поддерживающим форматирование.

Для форматирования выводимых данных служит следующая форма метода `WriteLine()`.

```
WriteLine("форматирующая строка", arg0, arg1, ... , argN);
```

В этой форме аргументы метода `WriteLine()` разделяются запятой, а не знаком +. А `форматирующая строка` состоит из двух следующих элементов: обычных печатаемых символов, отображаемых в исходном виде, а также команд форматирования.

Ниже приведена общая форма команд форматирования:

```
{argnum, width: fmt}
```

где *argnum* — это номер отображаемого аргумента, начиная с нуля; *width* — минимальная ширина поля, а *fmt* — спецификатор формата. Параметры *width* и *fmt* не являются обязательными. Поэтому в своей простейшей форме команда форматирования просто указывает конкретные аргументы для отображения. Например, команда `{0}` указывает аргумент *arg0*, команда `{1}` — аргумент *arg1* и т.д.

Если во время выполнения программы в форматирующей строке встречается команда форматирования, то вместо нее подставляется и затем отображается соответствующий аргумент, определяемый параметром *argnum*. Следовательно, от положения спецификатора формата в форматирующей строке зависит, где именно будут отображаться соответствующие данные. А номер аргумента определяет конкретный форматируемый аргумент.

Если в команде форматирования указывается параметр *fmt*, то данные отображаются в указываемом формате. В противном случае используется формат, выбираемый по умолчанию. Если же в команде форматирования указывается параметр *width*, то выводимые данные дополняются пробелами для достижения минимально необходимой ширины поля. При положительном значении параметра *width* выводимые данные выравниваются по правому краю, а при отрицательном значении — по левому краю.

Оставшаяся часть данной главы посвящена вопросам форматирования и отдельным спецификаторам формата.

Спецификаторы формата числовых данных

Для числовых данных определено несколько спецификаторов формата, сведенных в табл. 22.4. Каждый спецификатор формата может включать в себя дополнительный, но необязательный спецификатор точности. Так, если числовое значение требуется указать в формате с фиксированной точкой и двумя десятичными разрядами в дробной части, то для этой цели служит спецификатор `F2`.

Таблица 22.4. Спецификаторы формата числовых данных

Спецификатор	Формат	Назначение спецификатора точности
`C`	Денежная единица	Задает количество десятичных разрядов
`c`	То же, что и `C`	
`D`	Целочисленный (используется только с целыми числами)	Задает минимальное количество цифр. При необходимости результат дополняется начальными нулями
`d`	То же, что и `D`	
`E`	Экспоненциальное представление чисел (в обозначении используется прописная буква E)	Задает количество десятичных разрядов. По умолчанию используется шесть рязрядов
`e`	Экспоненциальное представление чисел (в обозначении используется строчная буква e)	Задает количество десятичных разрядов. По умолчанию используется шесть рязрядов
`F`	Представление чисел с фиксированной точкой	Задает количество десятичных разрядов
`f`	То же, что и `F`	
`G`	Используется более короткий из двух форматов: `E` или `F`	См. спецификаторы `E` и `F`
`g`	Используется более короткий из двух форматов: `e` или `f`	См. спецификаторы `e` и `f`
`N`	Представление чисел с фиксированной точкой (и запятой в качестве разделителя групп разрядов)	Задает количество десятичных разрядов
`n`	То же, что и `N`	
`P`	Проценты	Задает количество десятичных разрядов
`p`	То же, что и `P`	
`R` или `r`	Числовое значение, которое преобразуется с помощью метода `Parse()` в эквивалентную внутреннюю форму. (Это так называемый "круговой" формат)	Не используется
`X`	Шестнадцатеричный (в обозначении используются прописные буквы A–F)	Задает минимальное количество цифр. При необходимости результат дополняется начальными нулями
`x`	Шестнадцатеричный (в обозначении используются строчные буквы A–F)	Задает минимальное количество цифр. При необходимости результат дополняется начальными нулями

Как пояснялось выше, конкретное действие спецификаторов формата зависит от текущих настроек параметров культурной среды. Например, спецификатор денежной единицы C автоматически отображает числовое значение в формате денежной единицы, выбранном для локализации программного обеспечения в конкретной культурной среде. Для большинства пользователей используемая по умолчанию информация о культурной среде соответствует их региональным стандартам и языковым особенностям. Поэтому один и тот же спецификатор формата может использоваться без учета культурного контекста, в котором выполняется программа.

В приведенной ниже программе демонстрируется применение нескольких спецификаторов формата числовых данных.

```
// Продемонстрировать применение различных
// спецификаторов формата числовых данных.

using System;

class FormatDemo {
   static void Main() {
   double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    Console.WriteLine("{0:F2}", v);

    Console.WriteLine("{0:N5}", v);

    Console.WriteLine("{0:e}", v);

    Console.WriteLine("{0:r}", v);

    Console.WriteLine("{0:p}", v2);

    Console.WriteLine("{0:X}", x);

    Console.WriteLine("{0:D12}", x);

    Console.WriteLine("{0:C}", 189.99);
  }
}
```

Эта программа дает следующий результат.

```
17688.66
17,688.65849
1.768866e+004
17688.65849
15.00 %
15
000000000021
$189.99
```

Обратите внимание на действие спецификатора точности в нескольких форматах.

Представление о номерах аргументов

Следует иметь в виду, что аргумент, связанный со спецификатором формата, определяется номером аргумента, а не его позицией в списке аргументов. Это означает,

что один и тот же аргумент может указываться неоднократно в одном вызове метода `WriteLine()`. Эта также означает, что аргументы могут отображаться в той последовательности, в какой они указываются в списке аргументов. В качестве примера рассмотрим следующую программу.

```
using System;

class FormatDemo2 {
  static void Main() {

    // Форматировать один и тот же аргумент тремя разными способами.
    Console.WriteLine("{0:F2} {0:F3} {0:e}", 10.12345);

    // Отобразить аргументы не по порядку.
    Console.WriteLine("{2:d} {0:d} {1:d}", 1, 2, 3);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
10.12 10.123 1.012345e+001
3 1 2
```

В первом операторе вызова метода `WriteLine()` один и тот же аргумент `10.12345` форматируется тремя разными способами. Это вполне допустимо, поскольку каждый спецификатор формата в этом вызове обозначает первый и единственный аргумент. А во втором вызове метода `WriteLine()` три аргумента отображаются не по порядку. Не следует забывать, что каких-то особых правил, предписывающих обозначать аргументы в спецификаторах формата в определенной последовательности, не существует. Любой спецификатор формата может обозначать какой угодно аргумент.

Применение методов `String.Format()` и `ToString()` для форматирования данных

Несмотря на все удобства встраивания команд форматирования выводимых данных в вызовы метода `WriteLine()`, иногда все же требуется сформировать строку, содержащую отформатированные данные, но не отображать ее сразу. Это дает возможность отформатировать данные заранее, чтобы вывести их в дальнейшем на выбранное устройство. Такая возможность особенно полезна для организации работы в среде с графическим пользовательским интерфейсом, подобной Windows, где ввод-вывод на консоль применяется редко, а также для подготовки вывода на веб-страницу.

Вообще говоря, отформатированное строковое представление отдельного значения может быть получено двумя способами. Один из них состоит в применении метода `String.Format()`, а другой — в передаче спецификатора формата методу `ToString()`, относящемуся к одному из встроенных в C# числовых типов данных. Оба способа рассматриваются далее по порядку.

Применение метода `String.Format()` для форматирования значений

Для получения отформатированного значения достаточно вызвать метод `Format()`, определенный в классе `String`, в соответствующей его форме. Все формы этого метода

перечислены в табл. 22.5. Метод `Format()` аналогичен методу `WriteLine()`, за исключением того, что он возвращает отформатированную строку, а не выводит ее на консоль.

Таблица 22.5. Формы метода `Format()`

Метод	Описание
`public static string Format(string format, object arg0)`	Форматирует объект *arg0* в соответствии с первой командой форматирования, которая содержится в строке *format*. Возвращает копию строки *format*, в которой команда форматирования заменена отформатированными данными
`public static string Format(string format, object arg0, object arg1)`	Форматирует объект *arg0* в соответствии с первой командой форматирования, содержащейся в строке *format*, а объект *arg1* — в соответствии со второй командой. Возвращает копию строки *format*, в которой команды форматирования заменены отформатированными данными
`public static string Format(string format, object arg0, object arg1, object arg2)`	Форматирует объекты *arg0*, *arg1* и *arg2* по соответствующим командам форматирования, содержащимся в строке *format*. Возвращает копию строки *format*, в которой команды форматирования заменены отформатированными данными
`public static string Format(string format, params object[] args)`	Форматирует значения, передаваемые в массиве *args*, в соответствии с командами форматирования, содержащимися в строке *format*. Возвращает копию строки *format*, в которой команды форматирования заменены отформатированными данными
`public static string Format(IFormatProvider provider, string format, params object[] args)`	Форматирует значения, передаваемые в массиве *args*, в соответствии с командами форматирования, содержащимися в строке *format*, используя поставщик формата *provider*. Возвращает копию строки *format*, в которой команды форматирования заменены отформатированными данными

Ниже приведен вариант предыдущего примера программы форматирования, измененный с целью продемонстрировать применение метода `String.Format()`. Этот вариант дает такой же результат, как и предыдущий.

```
// Использовать метод String.Format() для форматирования значений.

using System;

class FormatDemo {
  static void Main() {
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    string str = String.Format("{0:F2}", v);
```

```
    Console.WriteLine(str);

    str = String.Format("{0:N5}", v);
    Console.WriteLine(str);

    str = String.Format("{0:e}", v);
    Console.WriteLine(str);

    str = String.Format("{0:r}", v);
    Console.WriteLine(str);

    str = String.Format("{0:p}", v2);
    Console.WriteLine(str);

    str = String.Format("{0:X}", x);
    Console.WriteLine(str);

    str = String.Format("{0:D12}", x);
    Console.WriteLine(str);

    str = String.Format("{0:C}", 189.99);
    Console.WriteLine(str);
  }
}
```

Аналогично методу `WriteLine()`, метод `String.Format()` позволяет встраивать в свой вызов обычный текст вместе со спецификаторами формата, причем в вызове данного метода может быть указано несколько спецификаторов формата и значений. В качестве примера рассмотрим еще одну программу, отображающую текущую сумму и произведение чисел от 1 до 10.

```
// Еще один пример применения метода Format().

using System;

class FormatDemo2 {
  static void Main() {
    int i;
    int sum = 0;
    int prod = 1;
    string str;

    /* Отобразить текущую сумму и произведение чисел
       от 1 до 10. */
    for(i=1; i <= 10; i++) {
      sum += i;
      prod *= i;
      str = String.Format("Сумма:{0,3:D} Произведение:{1,8:D}",
                          sum, prod);
      Console.WriteLine(str);
    }
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Сумма:  1  Произведение:       1
Сумма:  3  Произведение:       2
Сумма:  6  Произведение:       6
Сумма: 10  Произведение:      24
Сумма: 15  Произведение:     120
Сумма: 21  Произведение:     720
Сумма: 28  Произведение:    5040
Сумма: 36  Произведение:   40320
Сумма: 45  Произведение:  362880
Сумма: 55  Произведение: 3628800
```

Обратите особое внимание в данной программе на следующий оператор.

```
str = String.Format("Сумма:{0,3:D} Произведение:{1,8:D}", sum, prod);
```

В этом операторе содержится вызов метода `Format()` с двумя спецификаторами формата: одним — для суммы (в переменной `sum`), а другим — для произведения (в переменной `prod`). Обратите также внимание на то, что номера аргументов указываются таким же образом, как и в вызове метода `WriteLine()`, и что в вызов метода `Format()` включается обычный текст, как, например, строка `"Сумма: "`. Этот текст передается данному методу и становится частью выводимой строки.

Применение метода `ToString()` для форматирования данных

Для получения отформатированного строкового представления отдельного значения любого числового типа, которому соответствует встроенная структура, например `Int32` или `Double`, можно воспользоваться методом `ToString()`. Этой цели служит приведенная ниже форма метода `ToString()`.

```
public string ToString("форматирующая строка")
```

В этой форме метод `ToString()` возвращает строковое представление вызывающего объекта в том формате, который определяет спецификатор `"форматирующая строка"`, передаваемый данному методу. Например, в следующей строке кода формируется строковое представление значения 188.99 в формате денежной единицы с помощью спецификатора формата `C`.

```
string str = 189.99.ToString("C");
```

Обратите внимание на то, что спецификатор формата передается методу `ToString()` непосредственно. В отличие от встроенных команд форматирования, используемых в вызовах методов `WriteLine()` и `Format()`, где для этой цели дополнительно указываются номер аргумента и ширина поля, в вызове метода `ToString()` достаточно указать только спецификатор формата.

Ниже приведен вариант примера предыдущей программы форматирования, измененный с целью продемонстрировать применение метода `ToString()` для получения отформатированных строк. Этот вариант дает такой же результат, как и предыдущий.

```
// Использовать метод ToString() для форматирования значений.

using System;

class ToStringDemo {
  static void Main() {
```

```
    double v = 17688.65849;
    double v2 = 0.15;
    int x = 21;

    string str = v.ToString("F2");
    Console.WriteLine(str);

    str = v.ToString("N5");
    Console.WriteLine(str);

    str = v.ToString("e");
    Console.WriteLine(str);

    str = v.ToString("r");
    Console.WriteLine(str);

    str = v2.ToString("p");
    Console.WriteLine(str);

    str = x.ToString("X");
    Console.WriteLine(str);

    str = x.ToString("D12");
    Console.WriteLine(str);

    str = 189.99.ToString("C");
    Console.WriteLine(str);
  }
}
```

Определение пользовательского формата числовых данных

Несмотря на всю полезность предопределенных спецификаторов формата числовых данных, в C# предоставляется также возможность определить пользовательский, т.е. свой собственный, формат, используя средство, называемое *форматом изображения*. Своим происхождением термин *формат изображения* обязан тому обстоятельству, что специальный формат пользователь определяет, задавая пример внешнего вида (т.е. изображение) выводимых данных. Такой подход вкратце упоминался в части I этой книги, а здесь он рассматривается более подробно.

Символы-заполнители специального формата числовых данных

Когда пользователь определяет специальный формат, он задает этот формат в виде примера (или изображения) того, как должны выглядеть выводимые данные. Для этой цели используются символы, перечисленные в табл. 22.6. Они служат в качестве заполнителей и рассматриваются далее по очереди.

Символ точки обозначает местоположение десятичной точки.

Символ-заполнитель # обозначает цифровую позицию, или разряд числа. Этот символ может указываться слева или справа от десятичной точки либо отдельно. Так, если справа от десятичной точки указывается несколько символов #, то они обозначают количество отображаемых десятичных цифр в дробной части числа. При

необходимости форматируемое числовое значение округляется. Когда же символы # указываются слева от десятичной точки, то они обозначают количество отображаемых десятичных цифр в целой части числа. При необходимости форматируемое числовое значение дополняется начальными нулями. Если целая часть числового значения состоит из большего количества цифр, чем количество указываемых символов #, то она отображается полностью, но в любом случае целая часть числового значения не усекается. В отсутствие десятичной точки наличие символа # обусловливает округление соответствующего целого значения. А нулевое значение, которое не существенно, например конечный нуль, не отображается. Правда, это обстоятельство несколько усложняет дело, поскольку при указании такого формата, как #.##, вообще ничего не отображается, если форматируемое числовое значение равно нулю. Для вывода нулевого значения служит рассматриваемый далее символ-заполнитель 0.

Таблица 22.6. Символы-заполнители специального формата числовых данных

Символ-заполнитель	Назначение
#	Цифра
.	Десятичная точка
,	Разделитель групп разрядов
%	Процент
0	Используется для дополнения начальными и конечными нулями
;	Выделяет разделы, описывающие формат для положительных, отрицательных и нулевых значений
E0 E+0 E-0 e0 e+0 e-0	Экспоненциальное представление чисел

Символ-заполнитель 0 обусловливает дополнение форматируемого числового значения начальными или конечными нулями, чтобы обеспечить минимально необходимое количество цифр в строковом представлении данного значения. Этот символ может указываться как слева, как и справа от десятичной точки. Например, следующая строка кода:

```
Console.WriteLine("{0:00##.#00}", 21.3);
```

выводит такой результат.

```
0021.300
```

Значения, состоящие из большего количества цифр, будут полностью отображаться слева от десятичной точки, а округленные — справа.

При отображении больших числовых значений отдельные группы цифр могут отделяться друг от друга запятыми, для чего достаточно вставить запятую в шаблон, состоящий из символов #. Например, следующая строка кода:

```
Console.WriteLine("{0:#,###.#}", 3421.3);
```

выводит такой результат.

```
3,421.3.
```

Указывать запятую на каждой позиции совсем не обязательно. Если указать запятую в шаблоне один раз, то она будет автоматически вставляться в форматируемом числовом значении через каждые три цифры слева от десятичной запятой. Например, следующая строка кода:

```
Console.WriteLine("{0:#,###.#}", 8763421.3);
```

дает такой результат.

```
8,763,421.3.
```

У запятой имеется и другое назначение. Если запятая вставляется непосредственно перед десятичной точкой, то она выполняет роль масштабного коэффициента. Каждая запятая делит форматируемое числовое значение на 1000. Например, следующая строка кода:

```
Console.WriteLine("Значение в тысячах: {0:#,###,.#}", 8763421.3);
```

дает такой результат.

```
Значение в тысячах: 8,763.4
```

Как показывает приведенный выше результат, числовое значение выводится масштабированным в тысячах.

Помимо символов-заполнителей, пользовательский спецификатор формата может содержать любые другие символы, которые появляются в отформатированной строке без изменения на тех местах, где они указаны в спецификаторе формата. Например, при выполнении следующего фрагмента кода:

```
Console.WriteLine("КПД топлива: {0:##.# миль на галлон }", 21.3);
```

выводится такой результат.

```
КПД топлива: 21.3 миль на галлон
```

При необходимости в форматируемой строке можно также указывать такие управляющие последовательности, как `\t` или `\n`.

Символы-заполнители `E` и `e` обусловливают отображение числовых значений в экспоненциальном представлении. В этом случае после символа `E` или `e` должен быть указан хотя бы один нуль, хотя их может быть и больше. Нули обозначают количество отображаемых десятичных цифр. Дробная часть числового значения округляется в соответствии с заданным форматом отображения. Если указывается символ `E`, то он отображается прописной буквой "E". А если указывается символ `e`, то он отображается строчной буквой "e". Для того чтобы знак порядка отображался всегда, используются формы `E+` или `e+`. А для отображения знака порядка только при выводе отрицательных значений служат формы `E`, `e`, `E-` или `e-`.

Знак `;` служит разделителем в различных форматах вывода положительных, отрицательных и нулевых значений. Ниже приведена общая форма пользовательского спецификатора формата, в котором используется знак `;`.

положительный_формат; отрицательный_формат; нулевой_формат

Рассмотрим следующий пример.

```
Console.WriteLine("{0:#.##;(#.##);0.00}", num);
```

Если значение переменной `num` положительно, то оно отображается с двумя разрядами после десятичной точки. Если же значение переменной `num` отрицательно, то оно также отображается с двумя разрядами после десятичной точки, но в круглых скобках. А если значение переменной `num` равно нулю, то оно отображается в виде строки `0.00`. Когда используются разделители, указывать все части приведенной выше формы пользовательского спецификатора формата совсем не обязательно. Так, если

требуется вывести только положительные или отрицательные значения, *нулевой_формат* можно опустить. (В данном случае нуль форматируется как положительное значение.) С другой стороны, можно опустить *отрицательный_формат*. И в этом случае *положительный_формат* и *нулевой_формат* должны разделяться точкой с запятой. А в итоге *положительный_формат* будет использоваться для форматирования не только положительных, но и отрицательных значений.

В приведенном ниже примере программы демонстрируется лишь несколько специальных форматов, которые могут быть определены пользователем.

```
// Пример применения специальных форматов.

using System;

class PictureFormatDemo {
  static void Main() {
    double num = 64354.2345;

    Console.WriteLine("Формат по умолчанию: " + num);

    // Отобразить числовое значение с 2 разрядами после десятичной точки.
    Console.WriteLine("Значение с 2 десятичными разрядами: " +
                      "{0:#.##}", num);

    // Отобразить числовое значение с 2 разрядами после
    // десятичной точки и запятыми перед ней.
    Console.WriteLine("Добавить запятые: {0:#,###.##}", num);

    // Отобразить числовое значение в экспоненциальном представлении.
    Console.WriteLine("Использовать экспоненциальное представление: " +
                      "{0:#.###e+00}", num);

    // Отобразить числовое значение, масштабированное в тысячах.
Console.WriteLine("Значение в тысячах: " + "{0:#0,}", num);

    /* Отобразить по-разному положительные,
       отрицательные и нулевые значения. */
    Console.WriteLine("Отобразить по-разному положительные," +
                      "отрицательные и нулевые значения.");
    Console.WriteLine("{0:#.#;(#.##);0.00}", num);
    num = -num;
    Console.WriteLine("{0:#.##;(#.##);0.00}", num);
    num = 0.0;
    Console.WriteLine("{0:#.##;(#.##);0.00}", num);

    // Отобразить числовое значение в процентах.
    num = 0.17;
    Console.WriteLine("Отобразить в процентах: {0:#%}", num);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Формат по умолчанию: 64354.2345
Значение с 2 десятичными разрядами: 64354.23
Добавить запятые: 64,354.23
```

```
Использовать экспоненциальное представление: 6.435e+04
Значение в тысячах: 64
Отобразить по-разному положительные, отрицательные и нулевые значения.
64354.2
(64354.23)
0.00
Отобразить в процентах: 17%
```

Форматирование даты и времени

Помимо числовых значений, форматированию нередко подлежит и другой тип данных: `DateTime`. Это структура, представляющая дату и время. Значения даты и времени могут отображаться самыми разными способами. Ниже приведены лишь некоторые примеры их отображения.

```
06/05/2005
Friday, January 1, 2010
12:59:00
12:59:00 PM
```

Кроме того, дата и время могут быть по-разному представлены в отдельных странах. Для этой цели в среде .NET Framework предусмотрена обширная подсистема форматирования значений даты и времени.

Форматирование даты и времени осуществляется с помощью спецификаторов формата. Спецификаторы формата даты и времени сведены в табл. 22.7. Конкретное представление даты и времени может отличаться в силу региональных и языковых особенностей и поэтому зависит от настройки параметров культурной среды.

Таблица 22.7. Спецификаторы формата даты и времени

Спецификатор	Формат
`D`	Дата в длинной форме
`d`	Дата в краткой форме
`F`	Дата и время в длинной форме
`f`	Дата и время в краткой форме
`G`	Дата — в краткой форме, время — в длинной
`gg`	Дата и время — в краткой форме
`M`	Месяц и день
`m`	То же, что и `M`
`O`	Формат даты и времени, включая часовой пояс. Строка, составленная в формате `O`, может быть преобразована обратно в эквивалентную форму вывода даты и времени. Это так называемый "круговой" формат
`o`	То же, что и `O`
`R`	Дата и время в стандартной форме по Гринвичу
`r`	То же, что и `R`
`s`	Сортируемый формат представления даты и времени
`T`	Время в длинной форме
`t`	Время в краткой форме

Окончание табл. 22.7

Спецификатор	Формат
U	Длинная форма универсального представления даты и времени; время отображается как универсальное синхронизированное время (UTC)
u	Краткая форма универсального представления даты и времени
Y	Месяц и год
y	То же, что и Y

В приведенном ниже примере программы демонстрируется применение спецификаторов формата даты и времени.

```
// Отформатировать дату и время, используя стандартные форматы.

using System;

class TimeAndDateFormatDemo {
  static void Main() {
    DateTime dt = DateTime.Now; // получить текущее время

    Console.WriteLine("Формат d: {0:d}", dt);
    Console.WriteLine("Формат D: {0:D}", dt);

    Console.WriteLine("Формат t: {0:t}", dt);
    Console.WriteLine("Формат T: {0:T}", dt);

    Console.WriteLine("Формат f: {0:f}", dt);
    Console.WriteLine("Формат F: {0:F}", dt);

    Console.WriteLine("Формат g: {0:g}", dt);
    Console.WriteLine("Формат G: {0:G}", dt);

    Console.WriteLine("Формат m: {0:m}", dt);
     Console.WriteLine("Формат M: {0:M}", dt);

    Console.WriteLine("Формат o: {0:o}", dt);
    Console.WriteLine("Формат O: {0:O}", dt);

    Console.WriteLine("Формат r: {0:r}", dt);
    Console.WriteLine("Формат R: {0:R}", dt);

    Console.WriteLine("Формат s: {0:s}", dt);

    Console.WriteLine("Формат u: {0:u}", dt);
    Console.WriteLine("Формат U: {0:U}", dt);

    Console.WriteLine("Формат y: {0:y}", dt);
    Console.WriteLine("Y format: {0:Y}", dt);
  }
}
```

Эта программа дает следующий результат, который, впрочем, зависит от настроек языковых и региональных параметров локализации базового программного обеспечения.

```
Формат d: 2/11/2010
Формат D: Thursday, February 11, 2010
Формат t: 11:21 AM
Формат T: 11:21:23 AM
Формат f: Thursday, February 11, 2010 11:21 AM
Формат F: Thursday, February 11, 2010 11:21:23 AM
Формат g: 2/11/2010 11:21 AM
Формат G: 2/11/2010 11:21:23 AM
Формат m: February 11
Формат M: February 11
Формат o: 2010-02-11T11:21:23.3768153-06:00
Формат O: 2010-02-11T11:21:23.3768153-06:00
Формат r: Thu, 11 Feb 2010 11:21:23 GMT
Формат R: Thu, 11 Feb 2010 11:21:23 GMT
Формат s: 2010-02-11T11:21:23
Формат u: 2010-02-11 11:21:23Z
Формат U: Thursday, February 11, 2010 5:21:23 PM
Формат y: February, 2010
Формат Y: February, 2010
```

В следующем примере программы воспроизводятся очень простые часы. Время обновляется каждую секунду, и каждый час компьютер издает звонок. Для получения отформатированного строкового представления времени перед его выводом в этой программе используется метод `ToString()` из структуры `DateTime`. Через каждый час символ звукового предупреждающего сигнала присоединяется к отформатированной строке, представляющей время, в результате чего звенит звонок.

```
// Пример простых часов.

using System;

class SimpleClock {
  static void Main() {
    string t;
    int seconds;

    DateTime dt = DateTime.Now;
    seconds = dt.Second;

    for(;;) {
      dt = DateTime.Now;

      // обновлять время через каждую секунду
      if(seconds != dt.Second) {
        seconds = dt.Second;

        t = dt.ToString("T");

        if(dt.Minute==0 && dt.Second==0)
          t = t + "\a"; // производить звонок через каждый час

        Console.WriteLine(t);
      }
    }
  }
}
```

Определение пользовательского формата даты и времени

Несмотря на то что стандартные спецификаторы формата даты и времени предусмотрены практически на все случаи жизни, пользователь может определить свои собственные спёциальные форматы. Процесс определения пользовательских форматов даты и времени мало чем отличается от описанного выше для числовых типов значений. По существу, пользователь создает пример (т.е. изображение) того, как должны выглядеть выводимые данные даты и времени. Для определения пользовательского формата даты и времени служат символы-заполнители, перечисленные в табл. 22.8.

Таблица 22.8. Символы-заполнители специального формата даты и времени

Символ-заполнитель	Назначение
`d`	День месяца в виде числа в пределах от 1 до 31
`dd`	День месяца в виде числа в пределах от 1 до 31. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
`ddd`	Сокращенное название дня недели
`dddd`	Полное название дня недели
`f, ff, fff, ffff, fffff, ffffff, fffffff`	Дробная часть числового значения, обозначающего секунды. Количество десятичных разрядов определяется числом заданных символов `f`
`g`	Эра
`h`	Часы в виде числа в пределах от 1 до 12
`hh`	Часы в виде числа в пределах от 1 до 12. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
`H`	Часы в виде числа в пределах от 0 до 23
`HH`	Часы в виде числа в пределах от 0 до 23. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
`K`	Часовой пояс, указываемый в часах. Для автоматической коррекции местного времени и универсального синхронизированного времени (UTC) используется значение свойства `DateTime.Kind`. (Этот спецификатор формата рекомендуется теперь вместо спецификаторов с символами-заполнителями `z`.)
`m`	Минуты
`mm`	Минуты. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
`M`	Месяц в виде числа в пределах от 1 до 12
`MM`	Месяц в виде числа в пределах от 1 до 12. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
`MMM`	Сокращенное название месяца
`MMMM`	Полное название месяца
`s`	Секунды
`ss`	Секунды. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
`t`	Символ "A" или "P", обозначающий время A.M. (до полудня) или P.M. (после полудня) соответственно

Окончание табл. 22.8

Символ-заполнитель	Назначение
tt	A.M. или P.M.
y	Год в виде двух цифр, если недостаточно одной
yy	Год в виде двух цифр. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
yyy	Год в виде трех цифр
yyyy	Год в виде четырех цифр
yyyyy	Год в виде пяти цифр
z	Смещение часового пояса в часах
zz	Смещение часового пояса в часах. Числовые значения в пределах от 1 до 9 дополняются начальным нулем
zzz	Смещение часового пояса в часах и минутах
:	Разделитель для составляющих значения времени
/	Разделитель для составляющих значения даты
%fmt	Стандартный формат, соответствующий спецификатору формата fmt

Глядя на табл. 22.8, можно заметить, что символы-заполнители d, f, g, m, M, s и t выполняют ту же функцию, что и аналогичные символы-заполнители из табл. 22.7. Вообще говоря, если один из этих символов указывается отдельно, то он интерпретируется как спецификатор формата. В противном случае он считается символом-заполнителем. Поэтому если требуется указать несколько таких символов отдельно, но интерпретировать их как символы-заполнители, то перед каждым из них следует поставить знак %.

В приведенном ниже примере программы демонстрируется применение нескольких форматов даты и времени.

```
// Отформатировать дату и время, используя специальные форматы.

using System;

class CustomTimeAndDateFormatsDemo {
  static void Main() {
    DateTime dt = DateTime.Now;

    Console.WriteLine("Время: {0:hh:mm tt}", dt);
    Console.WriteLine("Время в 24-часовом формате: {0:HH:mm}", dt);
    Console.WriteLine("Дата: {0:ddd MMM dd, yyyy}", dt);

    Console.WriteLine("Эра: {0:gg}", dt);

    Console.WriteLine("Время в секундах: " +
                      "{0:HH:mm:ss tt}", dt);

    Console.WriteLine("День месяца в формате m: {0:m}", dt);
    Console.WriteLine("Минуты в формате m: {0:%m}", dt);
  }
}
```

Вот к какому результату приводит выполнение этой программы (опять же все зависит от конкретных настроек языковых и региональных параметров локализации базового программного обеспечения).

```
Время: 11:19 AM
Время 24-часовом формате: 11:19
Дата: Thu Feb 11, 2010
Эра: A.D.
Время в секундах: 11:19:40 AM
День месяца в формате m: February 11
Минуты в формате m: 19
```

Форматирование промежутков времени

Начиная с версии 4.0, в среде .NET Framework появилась возможность форматировать объекты типа `TimeSpan` — структуры, представляющей промежуток времени. Объект типа `TimeSpan` может быть получен самыми разными способами, в том числе и в результате вычитания одного объекта типа `DateTime` из другого. И хотя форматировать объекты типа `TimeSpan` приходится нечасто, о такой возможности все же стоит упомянуть вкратце.

По умолчанию в структуре `TimeSpan` поддерживаются три стандартных спецификатора формата даты и времени: `c`, `g` и `G`. Они обозначают инвариантную форму промежутка времени, короткую и длинную форму с учетом культурной среды соответственно (последняя форма всегда включает в себя дни). Кроме того, в структуре `TimeSpan` поддерживаются специальные спецификаторы формата даты и времени, приведенные в табл. 22.9. Вообще говоря, если один из этих спецификаторов используется в отдельности, его нужно предварить символом `%`.

Таблица 22.9. Символы-заполнители специального формата промежутка времени

Символ-заполнитель	Назначение
`d, dd, ddd, dddd, ddddd, dddddd, ddddddd`	Целые дни. Если указано несколько символов-заполнителей `d`, то отображается, по крайней мере, указанное количество цифр с начальными нулями, если требуется
`h, hh`	Часы (не считая тех, что составляют часть целого дня). Если указано `hh`, то отображаются две цифры с начальными нулями, если требуется
`m, mm`	Минуты (не считая тех, что составляют часть целого часа). Если указано `mm`, то отображаются две цифры с начальными нулями, если требуется
`s, ss`	Секунды (не считая тех, что составляют часть целой минуты). Если указано `ss`, то отображаются две цифры с начальными нулями, если требуется
`f, ff, fff, ffff, fffff, ffffff, fffffff`	Дробные доли секунды. Количество символов-заполнителей `f` обозначает точность представления, а остальные цифры отбрасываются
`F, FF, FFF, FFFF, FFFFF, FFFFFF, FFFFFFF`	Дробные доли секунды. Количество символов-заполнителей `F` обозначает точность представления, а остальные цифры отбрасываются и конечные нули не отображаются

В приведенной ниже программе демонстрируется форматирование объектов типа TimeSpan на примере отображения времени, которое приблизительно требуется для вывода на экран 1000 целых значений в цикле for.

```
// Отформатировать объект типа TimeSpan.

using System;

class TimeSpanDemo {
  static void Main() {
    DateTime start = DateTime.Now;

    // Вывести числа от 1 до 1000.
    for(int i = 1; i <= 1000; i++) {
      Console.Write(i + " ");
      if((i % 10) == 0) Console.WriteLine();
    }

    Console.WriteLine();

    DateTime end = DateTime.Now;

    TimeSpan span = end - start;

    Console.WriteLine("Время выполнения: {0:c}", span);
    Console.WriteLine("Время выполнения: {0:g}", span);
    Console.WriteLine("Время выполнения: {0:G}", span);
    Console.WriteLine("Время выполнения: 0.{0:fff} секунды", span);
  }
}
```

Выполнение этой программы приводит к следующему результату, который и в этом случае зависит от конкретных настроек языковых и региональных параметров локализации базового программного обеспечения, а также от загрузки системы задачами и ее быстродействия.

```
. . .
981 982 983 984 985 986 987 988 989 990
991 992 993 994 995 996 997 998 999 1000
Время выполнения: 00:00:00.0140000
Время выполнения: 0:00:00.014
Время выполнения: 0:00:00:00.0140000
Время выполнения: 0.014 секунды
```

Форматирование перечислений

В C# допускается также форматировать значения, определяемые в перечислении. Вообще говоря, значения из перечисления могут отображаться как по имени, так и по значению. Спецификаторы формата перечислений сведены в табл. 22.10. Обратите особое внимание на форматы G и F. Перед перечислениями, которые должны представлять битовые поля, следует указывать атрибут Flags. Как правило, в битовых полях хранятся значения, обозначающие отдельные двоичные разряды и упорядоченные

по степени числа 2. При наличии атрибута `Flags` имена всех битовых составляющих форматируемого значения, если, конечно, это действительное значение, отображаются с помощью спецификатора `G`. А с помощью спецификатора `F` отображаются имена всех битовых составляющих форматируемого значения, если оно составляется путем логического сложения по ИЛИ двух или более полей, определяемых в перечислении.

Таблица 22.10. Спецификаторы формата перечислений

Спецификатор	Назначение
`D`	Отображает значение в виде десятичного целого числа
`d`	То же, что и `D`
`F`	Отображает имя значения. Если это значение можно создать путем логического сложения по ИЛИ двух или более полей, определенных в перечислении, то данный спецификатор отображает имена всех битовых составляющих заданного значения, причем независимо от того, задан атрибут `Flags` или нет
`f`	То же, что и `F`
`G`	Отображает имя значения. Если перед форматируемым перечислением указывается атрибут `Flags`, то данный спецификатор отображает имена всех битовых составляющих заданного значения, если, конечно, это допустимое значение
`g`	То же, что и `G`
`X`	Отображает значение в виде шестнадцатеричного целого числа. Для отображения как минимум восьми цифр форматируемое значение дополняется (при необходимости) начальными нулями
`x`	То же, что и `X`

В приведенной ниже программе демонстрируется применение спецификаторов формата перечислений.

```
// Отформатировать перечисление.

using System;

class EnumFmtDemo {
  enum Direction { North, South, East, West }
  [Flags] enum Status { Ready=0x1, OffLine=0x2,
                        Waiting=0x4, TransmitOK=0x8,
                        ReceiveOK=0x10, OnLine=0x20 }

  static void Main() {
    Direction d = Direction.West;

    Console.WriteLine("{0:G}", d);
    Console.WriteLine("{0:F}", d);

    Console.WriteLine("{0:D}", d);
    Console.WriteLine("{0:X}", d);

    Status s = Status.Ready | Status.TransmitOK;
```

```
    Console.WriteLine("{0:G}", s);
    Console.WriteLine("{0:F}", s);
    Console.WriteLine("{0:D}", s);
    Console.WriteLine("{0:X}", s);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
West
West
3
00000003
Ready, TransmitOK
Ready, TransmitOK
9
00000009
```

23
ГЛАВА

Многопоточное программирование. Часть первая: основы

Среди многих замечательных свойств языка C# особое место принадлежит поддержке *многопоточного программирования*. Многопоточная программа состоит из двух или более частей, выполняемых параллельно. Каждая часть такой программы называется *потоком* и определяет отдельный путь выполнения команд. Таким образом, многопоточная обработка является особой формой многозадачности.

Многопоточное программирование опирается на целый ряд средств, предусмотренных для этой цели в самом языке C#, а также на классы, определенные в среде .NET Framework. Благодаря встроенной в C# поддержке многопоточной обработки сводятся к минимуму или вообще устраняются многие трудности, связанные с организацией многопоточной обработки в других языках программирования. Как станет ясно из дальнейшего, поддержка в C# многопоточной обработки четко организована и проста для понимания.

С выпуском версии 4.0 в среде .NET Framework появились два важных дополнения, имеющих отношение к многопоточным приложениям. Первым из них является TPL (Task Parallel Library — Библиотека распараллеливания задач), а вторым — PLINQ (Parallel LINQ — Параллельный язык интегрированных запросов). Оба дополнения поддерживают параллельное программирование и позволяют использовать преимущества, предоставляемые многопроцессорными (многоядерными) компьютерами в отношении обработки данных. Кроме того, библиотека TPL упрощает создание многопоточных приложений и управление ими. В силу этого многопоточная обработка, опирающаяся на

TPL, рекомендуется теперь как основной подход к разработке многопоточных приложений. Тем не менее накопленный опыт создания исходной многопоточной подсистемы по-прежнему имеет значение по целому ряду причин. Во-первых, уже существует немалый объем унаследованного кода, в котором применяется первоначальный подход к многопоточной обработке. Если приходится работать с таким кодом или сопровождать его, то нужно знать, как работает исходная многопоточная система. Во-вторых, в коде, опирающемся на TPL, могут по-прежнему использоваться элементы исходной многопоточной системы, и особенно ее средства синхронизации. И в-третьих, несмотря на то что сама библиотека TPL основывается на абстракции, называемой *задачей*, она по-прежнему неявно опирается на потоки и потоковые средства, описываемые в этой главе. Поэтому для полного усвоения и применения TPL потребуются твердые знания материала, излагаемого в этой главе.

И наконец, следует особо подчеркнуть, что многопоточная обработка представляет собой довольно обширную тему, и поэтому подробное ее изложение выходит за рамки этой книги. В этой и последующей главах представлен лишь беглый обзор данной темы и демонстрируется ряд основополагающих методик. Следовательно, материал этих глав может служить введением в эту важную тему и основанием для дальнейшего ее самостоятельного изучения.

Основы многопоточной обработки

Различают две разновидности многозадачности: на основе процессов и на основе потоков. В связи с этим важно понимать отличия между ними. *Процесс* фактически представляет собой исполняемую программу. Поэтому *многозадачность на основе процессов* — это средство, благодаря которому на компьютере могут параллельно выполняться две программы и более. Так, многозадачность на основе процессов позволяет одновременно выполнять программы текстового редактора, электронных таблиц и просмотра содержимого в Интернете. При организации многозадачности на основе процессов программа является наименьшей единицей кода, выполнение которой может координировать планировщик задач.

Поток представляет собой координируемую единицу исполняемого кода. Своим происхождением этот термин обязан понятию "поток исполнения". При организации многозадачности на основе потоков у каждого процесса должен быть по крайней мере один поток, хотя их может быть и больше. Это означает, что в одной программе одновременно могут решаться две задачи и больше. Например, текст может форматироваться в редакторе текста одновременно с его выводом на печать, при условии, что оба эти действия выполняются в двух отдельных потоках.

Отличия в многозадачности на основе процессов и потоков могут быть сведены к следующему: многозадачность на основе процессов организуется для параллельного выполнения программ, а многозадачность на основе потоков — для параллельного выполнения отдельных частей одной программы.

Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря возможности выгодно использовать время простоя, неизбежно возникающее в ходе выполнения большинства программ. Как известно, большинство устройств ввода-вывода, будь то устройства, подключенные к сетевым портам, накопители на дисках или клавиатура, работают намного медленнее, чем центральный процессор (ЦП). Поэтому большую

часть своего времени программе приходится ожидать отправки данных на устройство ввода-вывода или приема информации из него. А благодаря многопоточной обработке программа может решать какую-нибудь другую задачу во время вынужденного простоя. Например, в то время как одна часть программы отправляет файл через соединение с Интернетом, другая ее часть может выполнять чтение текстовой информации, вводимой с клавиатуры, а третья — осуществлять буферизацию очередного блока отправляемых данных.

Поток может находиться в одном из нескольких состояний. В целом, поток может быть *выполняющимся*; *готовым к выполнению*, как только он получит время и ресурсы ЦП; *приостановленным*, т.е. временно не выполняющимся; *возобновленным* в дальнейшем; *заблокированным* в ожидании ресурсов для своего выполнения; а также *завершенным*, когда его выполнение окончено и не может быть возобновлено.

В среде .NET Framework определены две разновидности потоков: *приоритетный* и *фоновый*. По умолчанию создаваемый поток автоматически становится приоритетным, но его можно сделать фоновым. Единственное отличие приоритетных потоков от фоновых заключается в том, что фоновый поток автоматически завершается, если в его процессе остановлены все приоритетные потоки.

В связи с организацией многозадачности на основе потоков возникает потребность в особого рода режиме, который называется *синхронизацией* и позволяет координировать выполнение потоков вполне определенным образом. Для такой синхронизации в C# предусмотрена отдельная подсистема, основные средства которой рассматриваются в этой главе.

Все процессы состоят хотя бы из одного потока, который обычно называют *основным*, поскольку именно с него начинается выполнение программы. Следовательно, в основном потоке выполнялись все приведенные ранее примеры программ. Из основного потока можно создать другие потоки.

В языке C# и среде .NET Framework поддерживаются обе разновидности многозадачности: на основе процессов и на основе потоков. Поэтому средствами C# можно создавать как процессы, так и потоки, а также управлять и теми и другими. Для того чтобы начать новый процесс, от программирующего требуется совсем немного усилий, поскольку каждый предыдущий процесс совершенно обособлен от последующего. Намного более важной оказывается поддержка в C# многопоточной обработки, благодаря которой упрощается написание высокопроизводительных, многопоточных программ на C# по сравнению с некоторыми другими языками программирования.

Классы, поддерживающие многопоточное программирование, определены в пространстве имен `System.Threading`. Поэтому любая многопоточная программа на C# включает в себя следующую строку кода.

```
using System.Threading;
```

Класс `Thread`

Система многопоточной обработки основывается на классе `Thread`, который инкапсулирует поток исполнения. Класс `Thread` является *герметичным*, т.е. он не может наследоваться. В классе `Thread` определен ряд методов и свойств, предназначенных для управления потоками. На протяжении всей этой главы будут рассмотрены наиболее часто используемые члены данного класса.

Создание и запуск потока

Для создания потока достаточно получить экземпляр объекта типа `Thread`, т.е. класса, определенного в пространстве имен `System.Threading`. Ниже приведена простейшая форма конструктора класса `Thread`:

```
public Thread(ThreadStart запуск)
```

где *запуск* — это имя метода, вызываемого с целью начать выполнение потока, а `ThreadStart` — делегат, определенный в среде .NET Framework, как показано ниже.

```
public delegate void ThreadStart()
```

Следовательно, метод, указываемый в качестве точки входа в поток, должен иметь возвращаемый тип `void` и не принимать никаких аргументов.

Вновь созданный новый поток не начнет выполняться до тех пор, пока не будет вызван его метод `Start()`, определяемый в классе `Thread`. Существуют две формы объявления метода `Start()`. Ниже приведена одна из них.

```
public void Start()
```

Однажды начавшись, поток будет выполняться до тех пор, пока не произойдет возврат из метода, на который указывает *запуск*. Таким образом, после возврата из этого метода поток автоматически прекращается. Если же попытаться вызвать метод `Start()` для потока, который уже начался, это приведет к генерированию исключения `ThreadStateException`.

В приведенном ниже примере программы создается и начинает выполняться новый поток.

```
// Создать поток исполнения.

using System;
using System.Threading;

class MyThread {
  public int Count;
  string thrdName;

  public MyThread(string name) {
    Count = 0;
    thrdName = name;
  }

  // Точка входа в поток.
  public void Run() {
    Console.WriteLine(thrdName + " начат.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + thrdName + ", Count = " + Count);
      Count++;
    } while(Count < 10);
    Console.WriteLine(thrdName + " завершен.");
  }
```

```
}

class MultiThread {
  static void Main() {
    Console.WriteLine("Основной поток начат.");

    // Сначала сконструировать объект типа MyThread.
    MyThread mt = new MyThread("Потомок #1");

    // Далее сконструировать поток из этого объекта.
    Thread newThrd = new Thread(mt.Run);

    // И наконец, начать выполнение потока.
    newThrd.Start();
    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt.Count != 10);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Рассмотрим приведенную выше программу более подробно. В самом ее начале определяется класс `MyThread`, предназначенный для создания второго потока исполнения. В методе `Run()` этого класса организуется цикл для подсчета от 0 до 9. Обратите внимание на вызов статического метода `Sleep()`, определенного в классе `Thread`. Этот метод обусловливает приостановление того потока, из которого он был вызван, на определенный период времени, указываемый в миллисекундах. Когда приостанавливается один поток, может выполняться другой. В данной программе используется следующая форма метода `Sleep()`:

```
public static void Sleep(int миллисекунд_простоя)
```

где *миллисекунд_простоя* обозначает период времени, на который приостанавливается выполнение потока. Если указанное количество *миллисекунд_простоя* равно нулю, то вызывающий поток приостанавливается лишь для того, чтобы предоставить возможность для выполнения потока, ожидающего своей очереди.

В методе `Main()` новый объект типа `Thread` создается с помощью приведенной ниже последовательности операторов.

```
// Сначала сконструировать объект типа MyThread.
MyThread mt = new MyThread("Потомок #1");

// Далее сконструировать поток из этого объекта.
Thread newThrd = new Thread(mt.Run);

// И наконец, начать выполнение потока.
newThrd.Start();
```

Как следует из комментариев к приведенному выше фрагменту кода, сначала создается объект типа `MyThread`. Затем этот объект используется для создания объекта типа `Thread`, для чего конструктору этого объекта в качестве точки входа передается метод `mt.Run()`. И наконец, выполнение потока начинается с вызова метода `Start()`.

Благодаря этому метод `mt.Run()` выполняется в своем собственном потоке. После вызова метода `Start()` выполнение основного потока возвращается к методу `Main()`, где начинается цикл `do-while`. Оба потока продолжают выполняться, совместно используя ЦП, вплоть до окончания цикла. Ниже приведен результат выполнения данной программы. (Он может отличаться в зависимости от среды выполнения, операционной системы и степени загрузки задач.)

```
Основной поток начат.
Потомок #1 начат.
....В потоке Потомок #1, Count = 0
....В потоке Потомок #1, Count = 1
....В потоке Потомок #1, Count = 2
....В потоке Потомок #1, Count = 3
....В потоке Потомок #1, Count = 4
....В потоке Потомок #1, Count = 5
....В потоке Потомок #1, Count = 6
....В потоке Потомок #1, Count = 7
....В потоке Потомок #1, Count = 8
....В потоке Потомок #1, Count = 9
Потомок #1 завершен.
Основной поток завершен.
```

Зачастую в многопоточной программе требуется, чтобы основной поток был последним потоком, завершающим ее выполнение. Формально программа продолжает выполняться до тех пор, пока не завершатся все ее приоритетные потоки. Поэтому требовать, чтобы основной поток завершал выполнение программы, совсем не обязательно. Тем не менее этого правила принято придерживаться в многопоточном программировании, поскольку оно явно определяет конечную точку программы. В рассмотренной выше программе предпринята попытка сделать основной поток завершающим ее выполнение. Для этой цели значение переменной `Count` проверяется в цикле `do-while` внутри метода `Main()`, и как только это значение оказывается равным 10, цикл завершается и происходит поочередный возврат из методов `Sleep()`. Но такой подход далек от совершенства, поэтому далее в этой главе будут представлены более совершенные способы организации ожидания одного потока до завершения другого.

Простые способы усовершенствования многопоточной программы

Рассмотренная выше программа вполне работоспособна, но ее можно сделать более эффективной, внеся ряд простых усовершенствований. Во-первых, можно сделать так, чтобы выполнение потока начиналось сразу же после его создания. Для этого достаточно получить экземпляр объекта типа `Thread` в конструкторе класса `MyThread`. И во-вторых, в классе `MyThread` совсем не обязательно хранить имя потока, поскольку для этой цели в классе `Thread` специально определено свойство `Name`.

```
public string Name { get; set; }
```

Свойство `Name` доступно для записи и чтения и поэтому может служить как для запоминания, так и для считывания имени потока.

Ниже приведена версия предыдущей программы, в которую внесены упомянутые выше усовершенствования.

```
// Другой способ запуска потока.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name; // задать имя потока
    Thrd.Start(); // начать поток
  }

  // Точка входа в поток.
  void Run() {
    Console.WriteLine(Thrd.Name + " начат.");
    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + Thrd.Name + ", Count = " + Count);
      Count++;
    } while(Count < 10);
    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

class MultiThreadImproved {
  static void Main() {
    Console.WriteLine("Основной поток начат.");

    // Сначала сконструировать объект типа MyThread.
    MyThread mt = new MyThread("Потомок #1");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt.Count != 10);
    Console.WriteLine("Основной поток завершен.");
  }
}
```

Эта версия программы дает такой же результат, как и предыдущая. Обратите внимание на то, что объект потока сохраняется в переменной `Thrd` из класса `MyThread`.

Создание нескольких потоков

В предыдущих примерах программ был создан лишь один порожденный поток. Но в программе можно породить столько потоков, сколько потребуется. Например, в следующей программе создаются три порожденных потока.

```
// Создать несколько потоков исполнения.
```

```
using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
    Console.WriteLine(Thrd.Name + " начат.");
    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + Thrd.Name + ", Count = " + Count);
      Count++;
    } while(Count < 10);

    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

class MoreThreads {
  static void Main() {
    Console.WriteLine("Основной поток начат.");

    // Сконструировать три потока.
    MyThread mt1 = new MyThread("Потомок #1");
    MyThread mt2 = new MyThread("Потомок #2");
    MyThread mt3 = new MyThread("Потомок #3");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt1.Count < 10 ||
             mt2.Count < 10 ||
             mt3.Count < 10);
    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен один из возможных результатов выполнения этой программы

```
Основной поток начат.
.Потомок #1 начат.
Потомок #2 начат.
Потомок #3 начат.
....В потоке Потомок #1, Count = 0
В потоке Потомок #2, Count = 0
В потоке Потомок #3, Count = 0
```

```
.....В потоке Потомок #1, Count = 1
В потоке Потомок #2, Count = 1
В потоке Потомок #3, Count = 1
.....В потоке Потомок #1, Count = 2
В потоке Потомок #2, Count = 2
В потоке Потомок #3, Count = 2
.....В потоке Потомок #1, Count = 3
В потоке Потомок #2, Count = 3
В потоке Потомок #3, Count = 3
..... В потоке Потомок #1, Count = 4
В потоке Потомок #2, Count = 4
В потоке Потомок #3, Count = 4
.....В потоке Потомок #1, Count = 5
В потоке Потомок #2, Count = 5
В потоке Потомок #3, Count = 5
..... В потоке Потомок #1, Count = 6
В потоке Потомок #2, Count = 6
В потоке Потомок #3, Count = 6
..... В потоке Потомок #1, Count = 7
В потоке Потомок #2, Count = 7
В потоке Потомок #3, Count = 7
..... В потоке Потомок #1, Count = 8
В потоке Потомок #2, Count = 8
В потоке Потомок #3, Count = 8
..... В потоке Потомок #1, Count = 9
Поток #1 завершен.
В потоке Потомок #2, Count = 9
Поток #2 завершен.
В потоке Потомок #3, Count = 9
Поток #3 завершен.
Основной поток завершен.
```

Как видите, после того как все три потока начнут выполняться, они будут совместно использовать ЦП. Приведенный выше результат может отличаться в зависимости от среды выполнения, операционной системы и других внешних факторов, влияющих на выполнение программы.

Определение момента окончания потока

Нередко оказывается полезно знать, когда именно завершается поток. В предыдущих примерах программ для этой цели отслеживалось значение переменной `Count`. Но ведь это далеко не лучшее и не совсем пригодное для обобщения решение. Правда, в классе `Thread` имеются два других средства для определения момента окончания потока. С этой целью можно, прежде всего, опросить доступное только для чтения свойство `IsAlive`, определяемое следующим образом.

```
public bool IsAlive { get; }
```

Свойство `IsAlive` возвращает логическое значение `true`, если поток, для которого оно вызывается, по-прежнему выполняется. Для "опробования" свойства `IsAlive` подставьте приведенный ниже фрагмент кода вместо кода в классе `MoreThread` из предыдущей версии многопоточной программы, как показано ниже.

```
// Использовать свойство IsAlive для отслеживания момента окончания потоков.
class MoreThreads {
  static void Main() {
    Console.WriteLine("Основной поток начат.");

    // Сконструировать три потока.
    MyThread mt1 = new MyThread("Поток #1");
    MyThread mt2 = new MyThread("Поток #2");
    MyThread mt3 = new MyThread("Поток #3");

    do {
      Console.Write(".");
      Thread.Sleep(100);
    } while (mt1.Thrd.IsAlive &&
             mt2.Thrd.IsAlive &&
             mt3.Thrd.IsAlive);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

При выполнении этой версии программы результат получается таким же, как и прежде. Единственное отличие заключается в том, что в ней используется свойство `IsAlive` для отслеживания момента окончания порожденных потоков.

Еще один способ отслеживания момента окончания состоит в вызове метода `Join()`. Ниже приведена его простейшая форма.

```
public void Join()
```

Метод `Join()` ожидает до тех пор, пока поток, для которого он был вызван, не завершится. Его имя отражает принцип ожидания до тех пор, пока вызывающий поток не *присоединится* к вызванному методу. Если же данный поток не был начат, то генерируется исключение `ThreadStateException`. В других формах метода `Join()` можно указать максимальный период времени, в течение которого следует ожидать завершения указанного потока.

В приведенном ниже примере программы метод `Join()` используется для того, чтобы основной поток завершился последним.

```
// Использовать метод Join().

using System;
using System.Threading;

class MyThread {
  public int Count;
   public Thread Thrd;

  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
```

```
    Console.WriteLine(Thrd.Name + " начат.");

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + Thrd.Name + ", Count = " + Count);
      Count++;
    } while(Count < 10);

    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

// Использовать метод Join() для ожидания до тех пор,
// пока потоки не завершатся.
class JoinThreads {
  static void Main() {
    Console.WriteLine("Основной поток начат.");

    // Сконструировать три потока.
    MyThread mt1 = new MyThread("Потомок #1");
    MyThread mt2 = new MyThread("Потомок #2");
    MyThread mt3 = new MyThread("Потомок #3");

    mt1.Thrd.Join();
    Console.WriteLine("Потомок #1 присоединен.");

    mt2.Thrd.Join();
    Console.WriteLine("Потомок #2 присоединен.");

    mt3.Thrd.Join();
    Console.WriteLine("Потомок #3 присоединен.");

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен один из возможных результатов выполнения этой программы. Напомним, что он может отличаться в зависимости от среды выполнения, операционной системы и прочих факторов, влияющих на выполнение программы.

```
Основной поток начат.
Потомок #1 начат.
Потомок #2 начат.
Потомок #3 начат.
В потоке Потомок #1, Count = 0
В потоке Потомок #2, Count = 0
В потоке Потомок #3, Count = 0
В потоке Потомок #1, Count = 1
В потоке Потомок #2, Count = 1
В потоке Потомок #3, Count = 1
В потоке Потомок #1, Count = 2
В потоке Потомок #2, Count = 2
В потоке Потомок #3, Count = 2
В потоке Потомок #1, Count = 3
В потоке Потомок #2, Count = 3
```

```
В потоке Потомок #3, Count = 3
В потоке Потомок #1, Count = 4
В потоке Потомок #2, Count = 4
В потоке Потомок #3, Count = 4
В потоке Потомок #1, Count = 5
В потоке Потомок #2, Count = 5
В потоке Потомок #3, Count = 5
В потоке Потомок #1, Count = 6
В потоке Потомок #2, Count = 6
В потоке Потомок #3, Count = 6
В потоке Потомок #1, Count = 7
В потоке Потомок #2, Count = 7
В потоке Потомок #3, Count = 7
В потоке Потомок #1, Count = 8
В потоке Потомок #2, Count = 8
В потоке Потомок #3, Count = 8
В потоке Потомок #1, Count = 9
Потомок #1 завершен.
В потоке Потомок #2, Count = 9
Потомок #2 завершен.
В потоке Потомок #3, Count = 9
Потомок #3 завершен.
Потомок #1 присоединен.
Потомок #2 присоединен.
Потомок #3 присоединен.
Основной поток завершен.
```

Как видите, выполнение потоков завершилось после возврата из последовательного ряда вызовов метода `Join()`.

Передача аргумента потоку

Первоначально в среде .NET Framework нельзя было передавать аргумент потоку, когда он начинался, поскольку у метода, служившего в качестве точки входа в поток, не могло быть параметров. Если же потоку требовалось передать какую-то информацию, то к этой цели приходилось идти различными обходными путями, например использовать общую переменную. Но этот недостаток был впоследствии устранен, и теперь аргумент может быть передан потоку. Для этого придется воспользоваться другими формами метода `Start()`, конструктора класса `Thread`, а также метода, служащего в качестве точки входа в поток.

Аргумент передается потоку в следующей форме метода `Start()`.

```
public void Start(object параметр)
```

Объект, указываемый в качестве аргумента *параметр*, автоматически передается методу, выполняющему роль точки входа в поток. Следовательно, для того чтобы передать аргумент потоку, достаточно передать его методу `Start()`.

Для применения параметризированной формы метода `Start()` потребуется следующая форма конструктора класса `Thread`:

```
public Thread(ParameterizedThreadStart запуск)
```

где *запуск* обозначает метод, вызываемый с целью начать выполнение потока. Обратите внимание на то, что в этой форме конструктора `запуск` имеет тип

ParameterizedThreadStart, а не ThreadStart, как в форме, использовавшейся в предыдущих примерах. В данном случае ParameterizedThreadStart является делегатом, объявляемым следующим образом.

```
public delegate void ParameterizedThreadStart(object obj)
```

Как видите, этот делегат принимает аргумент типа object. Поэтому для правильного применения данной формы конструктора класса Thread у метода, служащего в качестве точки входа в поток, должен быть параметр типа object.

В приведенном ниже примере программы демонстрируется передача аргумента потоку.

```
// Пример передачи аргумента методу потока.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  // Обратите внимание на то, что конструктору класса
  // MyThread передается также значение типа int.
  public MyThread(string name, int num) {
    Count = 0;

    // Вызвать конструктор типа ParameterizedThreadStart
    // явным образом только ради наглядности примера.
    Thrd = new Thread(this.Run);

    Thrd.Name = name;

    // Здесь переменная num передается методу Start()
    // в качестве аргумента.
    Thrd.Start(num);
  }

  // Обратите внимание на то, что в этой форме метода Run()
  // указывается параметр типа object.
  void Run(object num) {
    Console.WriteLine(Thrd.Name + " начат со счета " + num);

    do {
      Thread.Sleep(500);
      Console.WriteLine("В потоке " + Thrd.Name + ", Count = " + Count);
      Count++;
    } while(Count < (int) num);

    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

class PassArgDemo {
  static void Main() {
```

```
    // Обратите внимание на то, что число повторений
    // передается этим двум объектам типа MyThread.
    MyThread mt = new MyThread("Потомок #1", 5);
    MyThread mt2 = new MyThread("Потомок #2", 3);

    do {
      Thread.Sleep(100);
    } while (mt.Thrd.IsAlive | mt2.Thrd.IsAlive);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен результат выполнения данной программы, хотя у вас он может оказаться несколько иным.

```
Потомок#1 начат со счета 5
Потомок #2 начат со счета 3
В потоке Потомок #2, Count = 0
В потоке Потомок #1, Count = 0
В потоке Потомок #1, Count = 1
В потоке Потомок #2, Count = 1
В потоке Потомок #2, Count = 2
Потомок #2 завершен.
В потоке Потомок #1, Count = 2
В потоке Потомок #1, Count = 3
В потоке Потомок #1, Count = 4
Потомок #1 завершен.
Основной поток завершен.
```

Как следует из приведенного выше результата, первый поток повторяется пять раз, а второй — три раза. Число повторений указывается в конструкторе класса `MyThread` и затем передается методу `Run()`, служащему в качестве точки входа в поток, с помощью параметризированной формы `ParameterizedThreadStart` метода `Start()`.

Свойство `IsBackground`

Как упоминалось выше, в среде .NET Framework определены две разновидности потоков: приоритетный и фоновый. Единственное отличие между ними заключается в том, что процесс не завершится до тех пор, пока не окончится приоритетный поток, тогда как фоновые потоки завершаются автоматически по окончании всех приоритетных потоков. По умолчанию создаваемый поток становится приоритетным. Но его можно сделать фоновым, используя свойство `IsBackground`, определенное в классе `Thread`, следующим образом.

```
public bool IsBackground { get; set; }
```

Для того чтобы сделать поток фоновым, достаточно присвоить логическое значение `true` свойству `IsBackground`. А логическое значение `false` указывает на то, что поток является приоритетным.

Приоритеты потоков

У каждого потока имеется свой приоритет, который отчасти определяет, насколько часто поток получает доступ к ЦП. Вообще говоря, низкоприоритетные потоки получают доступ к ЦП реже, чем высокоприоритетные. Таким образом, в течение заданного промежутка времени низкоприоритетному потоку будет доступно меньше времени ЦП, чем высокоприоритетному. Как и следовало ожидать, время ЦП, получаемое потоком, оказывает определяющее влияние на характер его выполнения и взаимодействия с другими потоками, исполняемыми в настоящий момент в системе.

Следует иметь в виду, что, помимо приоритета, на частоту доступа потока к ЦП оказывают влияние и другие факторы. Так, если высокоприоритетный поток ожидает доступа к некоторому ресурсу, например для ввода с клавиатуры, он блокируется, а вместо него выполняется низкоприоритетный поток. В подобной ситуации низкоприоритетный поток может получать доступ к ЦП чаще, чем высокоприоритетный поток в течение определенного периода времени. И наконец, конкретное планирование задач на уровне операционной системы также оказывает влияние на время ЦП, выделяемое для потока.

Когда порожденный поток начинает выполняться, он получает приоритет, устанавливаемый по умолчанию. Приоритет потока можно изменить с помощью свойства `Priority`, являющегося членом класса `Thread`. Ниже приведена общая форма данного свойства:

```
public ThreadPriority Priority{ get; set; }
```

где `ThreadPriority` обозначает перечисление, в котором определяются приведенные ниже значения приоритетов.

```
ThreadPriority.Highest
ThreadPriority.AboveNormal
ThreadPriority.Normal
ThreadPriority.BelowNormal
ThreadPriority.Lowest
```

По умолчанию для потока устанавливается значение приоритета `ThreadPriority.Normal`.

Для того чтобы стало понятнее влияние приоритетов на исполнение потоков, обратимся к примеру, в котором выполняются два потока: один с более высоким приоритетом. Оба потока создаются в качестве экземпляров объектов класса `MyThread`. В методе `Run()` организуется цикл, в котором подсчитывается определенное число повторений. Цикл завершается, когда подсчет достигает величины 1000000000 или когда статическая переменная `stop` получает логическое значение `true`. Первоначально переменная `stop` получает логическое значение `false`. В первом потоке, где производится подсчет до 1000000000, устанавливается логическое значение `true` переменной `stop`. В силу этого второй поток оканчивается на следующем своем интервале времени. На каждом шаге цикла строка в переменной `currentName` проверяется на наличие имени исполняемого потока. Если имена потоков не совпадают, это означает, что произошло переключение исполняемых задач. Всякий раз, когда происходит переключение задач, имя нового потока отображается и присваивается переменной `currentName`. Это дает возможность отследить частоту доступа потока к ЦП. По окончании обоих потоков отображается число повторений цикла в каждом из них.

```
// Продемонстрировать влияние приоритетов потоков.

using System;
using System.Threading;

class MyThread {
  public int Count;
  public Thread Thrd;

  static bool stop = false;
  static string currentName;

  /* Сконструировать новый поток. Обратите внимание на то, что
     данный конструктор еще не начинает выполнение потоков. */
  public MyThread(string name) {
    Count = 0;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    currentName = name;
  }

  // Начать выполнение нового потока.
  void Run() {
    Console.WriteLine("Поток " + Thrd.Name + " начат.");
    do {
      Count++;
      if(currentName != Thrd.Name) {
        currentName = Thrd.Name;
        Console.WriteLine("В потоке " + currentName);
      }
    } while(stop == false && Count < 1000000000);
    stop = true;

    Console.WriteLine("Поток " + Thrd.Name + " завершен.");
  }
}

class PriorityDemo {
  static void Main() {
    MyThread mt1 = new MyThread("с высоким приоритетом");
    MyThread mt2 = new MyThread("с низким приоритетом");

    // Установить приоритеты для потоков.
    mt1.Thrd.Priority = ThreadPriority.AboveNormal;
    mt2.Thrd.Priority = ThreadPriority.BelowNormal;

    // Начать потоки.
    mt1.Thrd.Start();
    mt2.Thrd.Start();

    mt1.Thrd.Join();
    mt2.Thrd.Join();

    Console.WriteLine();
    Console.WriteLine("Поток " + mt1.Thrd.Name +
```

```
                    " досчитал до " + mt1.Count);
    Console.WriteLine("Поток " + mt2.Thrd.Name +
                    " досчитал до " + mt2.Count);
  }
}
```

Вот к какому результату может привести выполнение этой программы.

```
Поток с высоким приоритетом начат.
В потоке с высоким приоритетом
Поток с низким приоритетом начат.
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
В потоке с низким приоритетом
В потоке с высоким приоритетом
Поток с высоким приоритетом завершен.
Поток с низким приоритетом завершен.

Поток с высоким приоритетом досчитал до 1000000000
Поток с низким приоритетом досчитал до 23996334
```

Судя по результату, высокоприоритетный поток получил около 98% всего времени, которое было выделено для выполнения этой программы. Разумеется, конкретный результат может отличаться в зависимости от быстродействия ЦП и числа других задач, решаемых в системе, а также от используемой версии Windows.

Многопоточный код может вести себя по-разному в различных средах, поэтому никогда не следует полагаться на результаты его выполнения только в одной среде. Так, было бы ошибкой полагать, что низкоприоритетный поток из приведенного выше примера будет всегда выполняться лишь в течение небольшого периода времени до тех пор, пока не завершится высокоприоритетный поток. В другой среде высокоприоритетный поток может, например, завершиться еще до того, как низкоприоритетный поток выполнится хотя бы один раз.

Синхронизация

Когда используется несколько потоков, то иногда приходится координировать действия двух или более потоков. Процесс достижения такой координации называется *синхронизацией*. Самой распространенной причиной применения синхронизации служит необходимость разделять среди двух или более потоков общий ресурс, который может быть одновременно доступен только одному потоку. Например, когда в одном потоке выполняется запись информации в файл, второму потоку должно быть запрещено делать это в тот же самый момент времени. Синхронизация требуется и в том случае, если один поток ожидает событие, вызываемое другим потоком. В подобной ситуации требуются какие-то средства, позволяющие приостановить один из потоков до тех пор, пока не произойдет событие в другом потоке. После этого ожидающий поток может возобновить свое выполнение.

В основу синхронизации положено понятие *блокировки*, посредством которой организуется управление доступом к кодовому блоку в объекте. Когда объект заблокирован одним потоком, остальные потоки не могут получить доступ к заблокированному кодовому блоку. Когда же блокировка снимается одним потоком, объект становится доступным для использования в другом потоке.

Средство блокировки встроено в язык C#. Благодаря этому все объекты могут быть синхронизированы. Синхронизация организуется с помощью ключевого слова `lock`. Она была предусмотрена в C# с самого начала, и поэтому пользоваться ею намного проще, чем кажется на первый взгляд. В действительности синхронизация объектов во многих программах на C# происходит практически незаметно.

Ниже приведена общая форма блокировки:

```
lock(lockObj) {
  // синхронизируемые операторы
}
```

где *lockObj* обозначает ссылку на синхронизируемый объект. Если же требуется синхронизировать только один оператор, то фигурные скобки не нужны. Оператор `lock` гарантирует, что фрагмент кода, защищенный блокировкой для данного объекта, будет использоваться только в потоке, получающем эту блокировку. А все остальные потоки блокируются до тех пор, пока блокировка не будет снята. Блокировка снимается по завершении защищаемого ею фрагмента кода.

Блокируемым считается такой объект, который представляет синхронизируемый ресурс. В некоторых случаях им оказывается экземпляр самого ресурса или же произвольный экземпляр объекта, используемого для синхронизации. Следует, однако, иметь в виду, что блокируемый объект не должен быть общедоступным, так как в противном случае он может быть заблокирован из другого, неконтролируемого в программе фрагмента кода и в дальнейшем вообще не разблокируется. В прошлом для блокировки объектов очень часто применялась конструкция `lock(this)`. Но она пригодна только в том случае, если `this` является ссылкой на закрытый объект. В связи с возможными программными и концептуальными ошибками, к которым может привести конструкция `lock(this)`, применять ее больше не рекомендуется. Вместо нее лучше создать закрытый объект, чтобы затем заблокировать его. Именно такой подход принят в примерах программ, приведенных далее в этой главе. Но в унаследованном коде C# могут быть обнаружены примеры применения конструкции `lock(this)`. В одних случаях такой код оказывается безопасным, а в других — требует изменений во избежание серьезных осложнений при его выполнении.

В приведенной ниже программе синхронизация демонстрируется на примере управления доступом к методу `SumIt()`, суммирующему элементы целочисленного массива.

```
// Использовать блокировку для синхронизации доступа к объекту.

using System;
using System.Threading;

class SumArray {
  int sum;
  object lockOn = new object(); // закрытый объект, доступный
                                // для последующей блокировки
  public int SumIt(int[] nums) {
```

```
    lock(lockOn) { // заблокировать весь метод
      sum = 0; // установить исходное значение суммы

      for(int i=0; i < nums.Length; i++) {
        sum += nums[i];
        Console.WriteLine("Текущая сумма для потока " +
                          Thread.CurrentThread.Name + " равна " + sum);
        Thread.Sleep(10); // разрешить переключение задач
      }
      return sum;
    }
  }
}

class MyThread {
  public Thread Thrd;
  int[] a;
  int answer;

  // Создать один объект типа SumArray для всех
  // экземпляров класса MyThread.
  static SumArray sa = new SumArray();

  // Сконструировать новый поток.
  public MyThread(string name, int[] nums) {
    a = nums;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start(); // начать поток
  }

  // Начать выполнение нового потока.
  void Run() {
    Console.WriteLine(Thrd.Name + " начат.");

    answer = sa.SumIt(a);

    Console.WriteLine("Сумма для потока " + Thrd.Name + " равна " + answer);

    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

class Sync {
  static void Main() {
    int[] a = {1, 2, 3, 4, 5};
    MyThread mt1 = new MyThread("Потомок #1", a);
    MyThread mt2 = new MyThread("Потомок #2", a);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

Ниже приведен результат выполнения данной программы, хотя у вас он может оказаться несколько иным.

```
Потомок #1 начат.
Текущая сумма для потока Потомок #1 равна 1
Потомок #2 начат.
Текущая сумма для потока Потомок #1 равна 3
Текущая сумма для потока Потомок #1 равна 6
Текущая сумма для потока Потомок #1 равна 10
Текущая сумма для потока Потомок #1 равна 15
Текущая сумма для потока Потомок #2 равна 1
Сумма для потока Потомок #1 равна 15
Потомок #1 завершен.
Текущая сумма для потока Потомок #2 равна 3
Текущая сумма для потока Потомок #2 равна 6
Текущая сумма для потока Потомок #2 равна 10
Текущая сумма для потока Потомок #2 равна 15
Сумма для потока Потомок #2 равна 15
Потомок #2 завершен.
```

Как следует из приведенного выше результата, в обоих потоках правильно подсчитывается сумма, равная 15.

Рассмотрим эту программу более подробно. Сначала в ней создаются три класса. Первым из них оказывается класс `SumArray`, в котором определяется метод `SumIt()`, суммирующий элементы целочисленного массива. Вторым создается класс `MyThread`, в котором используется статический объект `sa` типа `SumArray`. Следовательно, единственный объект типа `SumArray` используется всеми объектами типа `MyThread`. С помощью этого объекта получается сумма элементов целочисленного массива. Обратите внимание на то, что текущая сумма запоминается в поле `sum` объекта типа `SumArray`. Поэтому если метод `SumIt()` используется параллельно в двух потоках, то оба потока попытаются обратиться к полю `sum`, чтобы сохранить в нем текущую сумму. А поскольку это может привести к ошибкам, то доступ к методу `SumIt()` должен быть синхронизирован. И наконец, в третьем классе, `Sync`, создаются два потока, в которых подсчитывается сумма элементов целочисленного массива.

Оператор `lock` в методе `SumIt()` препятствует одновременному использованию данного метода в разных потоках. Обратите внимание на то, что в операторе `lock` объект `lockOn` используется в качестве синхронизируемого. Это закрытый объект, предназначенный исключительно для синхронизации. Метод `Sleep()` намеренно вызывается для того, чтобы произошло переключение задач, хотя в данном случае это невозможно. Код в методе `SumIt()` заблокирован, и поэтому он может быть одновременно использован только в одном потоке. Таким образом, когда начинает выполняться второй порожденный поток, он не сможет войти в метод `SumIt()` до тех пор, пока из него не выйдет первый порожденный поток. Благодаря этому гарантируется получение правильного результата.

Для того чтобы полностью уяснить принцип действия блокировки, попробуйте удалить из рассматриваемой здесь программы тело метода `SumIt()`. В итоге метод `SumIt()` перестанет быть синхронизированным, а следовательно, он может параллельно использоваться в любом числе потоков для одного и того же объекта. Поскольку текущая сумма сохраняется в поле `sum`, она может быть изменена в каждом потоке, вызывающем метод `SumIt()`. Это означает, что если два потока одновременно вызывают метод `SumIt()` для одного и того же объекта, то конечный результат получается

неверным, поскольку содержимое поля sum отражает смешанный результат суммирования в обоих потоках. В качестве примера ниже приведен результат выполнения рассматриваемой здесь программы после снятия блокировки с метода SumIt().

```
Потомок #1 начат.
Текущая сумма для потока Потомок #1 равна 1
Потомок #2 начат.
Текущая сумма для потока Потомок #2 равна 1
Текущая сумма для потока Потомок #1 равна 3
Текущая сумма для потока Потомок #2 равна 5
Текущая сумма для потока Потомок #1 равна 8
Текущая сумма для потока Потомок #2 равна 11
Текущая сумма для потока Потомок #1 равна 15
Текущая сумма для потока Потомок #2 равна 19
Текущая сумма для потока Потомок #1 равна 24
Текущая сумма для потока Потомок #2 равна 29
Сумма для потока Потомок #1 равна 29
Потомок #1 завершен.
Текущая сумма для потока Потомок #2 равна 29
Потомок #2 завершен.
```

Как следует из приведенного выше результата, в обоих порожденных потоках метод SumIt() используется одновременно для одного и того же объекта, а это приводит к искажению значения в поле sum.

Ниже подведены краткие итоги использования блокировки.

- Если блокировка любого заданного объекта получена в одном потоке, то после блокировки объекта она не может быть получена в другом потоке.
- Остальным потокам, пытающимся получить блокировку того же самого объекта, придется ждать до тех пор, пока объект не окажется в разблокированном состоянии.
- Когда поток выходит из заблокированного фрагмента кода, соответствующий объект разблокируется.

Другой подход к синхронизации потоков

Несмотря на всю простоту и эффективность блокировки кода метода, как показано в приведенном выше примере, такое средство синхронизации оказывается пригодным далеко не всегда. Допустим, что требуется синхронизировать доступ к методу класса, который был создан кем-то другим и сам не синхронизирован. Подобная ситуация вполне возможна при использовании чужого класса, исходный код которого недоступен. В этом случае оператор lock нельзя ввести в соответствующий метод чужого класса. Как же тогда синхронизировать объект такого класса? К счастью, этот вопрос разрешается довольно просто: доступ к объекту может быть заблокирован из внешнего кода по отношению к данному объекту, для чего достаточно указать этот объект в операторе lock. В качестве примера ниже приведен другой вариант реализации предыдущей программы. Обратите внимание на то, что код в методе SumIt() уже не является заблокированным, а объект lockOn больше не объявляется. Вместо этого вызовы метода SumIt() блокируются в классе MyThread.

```
// Другой способ блокировки для синхронизации доступа к объекту.

using System;
```

```
using System.Threading;

class SumArray {
  int sum;

  public int SumIt(int[] nums) {
    sum = 0; // установить исходное значение суммы

    for(int i=0; i < nums.Length; i++) {
      sum += nums[i];
      Console.WriteLine("Текущая сумма для потока " +
                        Thread.CurrentThread.Name + " равна " + sum);
      Thread.Sleep(10); // разрешить переключение задач
    }
    return sum;
  }
}

class MyThread {
  public Thread Thrd;
  int[] a;
  int answer;

  /* Создать один объект типа SumArray для всех
     экземпляров класса MyThread. */
  static SumArray sa = new SumArray();

  // Сконструировать новый поток.
  public MyThread(string name, int[] nums) {
    a = nums;
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start(); // начать поток
  }

  // Начать выполнение нового потока.
  void Run() {
    Console.WriteLine(Thrd.Name + " начат.");

    // Заблокировать вызовы метода SumIt().
    lock(sa) answer = sa.SumIt(a);

    Console.WriteLine("Сумма для потока " + Thrd.Name +
                      " равна " + answer);

    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

class Sync {
  static void Main() {
    int[] a = {1, 2, 3, 4, 5};
    MyThread mt1 = new MyThread("Потомок #1", a);
    MyThread mt2 = new MyThread("Потомок #2", a);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

В данной программе блокируется вызов метода `sa.SumIt()`, а не сам метод `SumIt()`. Ниже приведена соответствующая строка кода, в которой осуществляется подобная блокировка.

```
// Заблокировать вызовы метода SumIt().
lock(sa) answer = sa.SumIt(a);
```

Объект `sa` является закрытым, и поэтому он может быть благополучно заблокирован. При таком подходе к синхронизации потоков данная программа дает такой же правильный результат, как и при первоначальном подходе.

Класс `Monitor` и блокировка

Ключевое слово `lock` на самом деле служит в C# быстрым способом доступа к средствам синхронизации, определенным в классе `Monitor`, который находится в пространстве имен `System.Threading`. В этом классе определен, в частности, ряд методов для управления синхронизацией. Например, для получения блокировки объекта вызывается метод `Enter()`, а для снятия блокировки — метод `Exit()`. Ниже приведены общие формы этих методов:

```
public static void Enter(object obj)
public static void Exit(object obj)
```

где *obj* обозначает синхронизируемый объект. Если же объект недоступен, то после вызова метода `Enter()` вызывающий поток ожидает до тех пор, пока объект не станет доступным. Тем не менее методы `Enter()` и `Exit()` применяются редко, поскольку оператор `lock` автоматически предоставляет эквивалентные средства синхронизации потоков. Именно поэтому оператор `lock` оказывается "более предпочтительным" для получения блокировки объекта при программировании на C#.

Впрочем, один метод из класса `Monitor` может все же оказаться полезным. Это метод `TryEnter()`, одна из общих форм которого приведена ниже.

```
public static bool TryEnter(object obj)
```

Этот метод возвращает логическое значение `true`, если вызывающий поток получает блокировку для объекта *obj*, а иначе он возвращает логическое значение `false`. Но в любом случае вызывающему потоку придется ждать своей очереди. С помощью метода `TryEnter()` можно реализовать альтернативный вариант синхронизации потоков, если требуемый объект временно недоступен.

Кроме того, в классе `Monitor` определены методы `Wait()`, `Pulse()` и `PulseAll()`, которые рассматриваются в следующем разделе.

Сообщение между потоками с помощью методов `Wait()`, `Pulse()` и `PulseAll()`

Рассмотрим следующую ситуацию. Поток *T* выполняется в кодовом блоке `lock`, и ему требуется доступ к ресурсу *R*, который временно недоступен. Что же тогда делать потоку *T*? Если поток *T* войдет в организованный в той или иной форме цикл опроса, ожидая освобождения ресурса *R*, то тем самым он свяжет соответствующий объект, блокируя доступ к нему других потоков. Это далеко не самое оптимальное решение, поскольку оно лишает отчасти преимуществ программирования для многопоточной

среды. Более совершенное решение заключается в том, чтобы временно освободить объект и тем самым дать возможность выполняться другим потокам. Такой подход основывается на некоторой форме сообщения между потоками, благодаря которому один поток может уведомлять другой о том, что он заблокирован и что другой поток может возобновить свое выполнение. Сообщение между потоками организуется в C# с помощью методов `Wait()`, `Pulse()` и `PulseAll()`.

Методы `Wait()`, `Pulse()` и `PulseAll()` определены в классе `Monitor` и могут вызываться только из заблокированного фрагмента блока. Они применяются следующим образом. Когда выполнение потока временно заблокировано, он вызывает метод `Wait()`. В итоге поток переходит в состояние ожидания, а блокировка с соответствующего объекта снимается, что дает возможность использовать этот объект в другом потоке. В дальнейшем ожидающий поток активизируется, когда другой поток войдет в аналогичное состояние блокировки, и вызывает метод `Pulse()` или `PulseAll()`. При вызове метода `Pulse()` возобновляется выполнение первого потока, ожидающего своей очереди на получение блокировки. А вызов метода `PulseAll()` сигнализирует о снятии блокировки всем ожидающим потокам.

Ниже приведены две наиболее часто используемые формы метода `Wait()`.

```
public static bool Wait(object obj)
public static bool Wait(object obj, int миллисекунд_простоя)
```

В первой форме ожидание длится вплоть до уведомления об освобождении объекта, а во второй форме — как до уведомления об освобождении объекта, так и до истечения периода времени, на который указывает количество *миллисекунд_простоя*. В обеих формах *obj* обозначает объект, освобождение которого ожидается.

Ниже приведены общие формы методов `Pulse()` и `PulseAll()`:

```
public static void Pulse(object obj)
public static void PulseAll(object obj)
```

где *obj* обозначает освобождаемый объект.

Если методы `Wait()`, `Pulse()` и `PulseAll()` вызываются из кода, находящегося за пределами синхронизированного кода, например из блока `lock`, то генерируется исключение `SynchronizationLockException`.

Пример использования методов `Wait()` и `Pulse()`

Для того чтобы стало понятнее назначение методов `Wait()` и `Pulse()`, рассмотрим пример программы, имитирующей тиканье часов и отображающей этот процесс на экране словами "тик" и "так". Для этой цели в программе создается класс `TickTock`, содержащий два следующих метода: `Tick()` и `Tock()`. Метод `Tick()` выводит на экран слово "тик", а метод `Tock()` — слово "так". Для запуска часов далее в программе создаются два потока: один из них вызывает метод `Tick()`, а другой — метод `Tock()`. Преследуемая в данном случае цель состоит в том, чтобы оба потока выполнялись, поочередно выводя на экран слова "тик" и "так", из которых образуется повторяющийся ряд "тик-так", имитирующий ход часов.

```
// Использовать методы Wait() и Pulse() для имитации тиканья часов.

using System;
using System.Threading;
```

```
class TickTock {
  object lockOn = new object();
  public void Tick(bool running) {
    lock(lockOn) {
      if(!running) { // остановить часы
        Monitor.Pulse(lockOn); // уведомить любые ожидающие потоки
        return;
      }

      Console.Write("тик ");
      Monitor.Pulse(lockOn); // разрешить выполнение метода Tock()
      Monitor.Wait(lockOn);  // ожидать завершения метода Tock()
    }
  }

  public void Tock(bool running) {
    lock(lockOn) {
      if(!running) { // остановить часы
        Monitor.Pulse(lockOn); // уведомить любые ожидающие потоки
        return;
      }

      Console.WriteLine("так");
      Monitor.Pulse(lockOn); // разрешить выполнение метода Tick()
      Monitor.Wait(lockOn);  // ожидать завершения метода Tick()
    }
  }
}

class MyThread {
  public Thread Thrd;
  TickTock ttOb;

  // Сконструировать новый поток.
  public MyThread(string name, TickTock tt) {
    Thrd = new Thread(this.Run);
    ttOb = tt;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Начать выполнение нового потока.
  void Run() {
    if(Thrd.Name == "Tick") {
      for(int i=0; i<5; i++) ttOb.Tick(true);
        ttOb.Tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.Tock(true);
        ttOb.Tock(false);
    }
  }
}
```

```
class TickingClock {
  static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Tick", tt);
    MyThread mt2 = new MyThread("Tock", tt);
    mt1.Thrd.Join();
    mt2.Thrd.Join();

    Console.WriteLine("Часы остановлены");
  }
}
```

Ниже приведен результат выполнения этой программы.

```
тик так
тик так
тик так
тик так
тик так
Часы остановлены
```

Рассмотрим эту программу более подробно. В методе `Main()` создается объект `tt` типа `TickTock`, который используется для запуска двух потоков на выполнение. Если в методе `Run()` из класса `MyThread` обнаруживается имя потока `Tick`, соответствующее ходу часов "тик", то вызывается метод `Tick()`. А если это имя потока `Tock`, соответствующее ходу часов "так", то вызывается метод `Tock()`. Каждый из этих методов вызывается пять раз подряд с передачей логического значения `true` в качестве аргумента. Часы идут до тех пор, пока этим методам передается логическое значение `true`, и останавливаются, как только передается логическое значение `false`.

Самая важная часть рассматриваемой здесь программы находится в методах `Tick()` и `Tock()`. Начнем с метода `Tick()`, код которого для удобства приводится ниже.

```
public void Tick(bool running) {
  lock(lockOn) {
    if(!running) { // остановить часы
      Monitor.Pulse(lockOn); // уведомить любые ожидающие потоки
      return;
    }

    Console.Write("тик ");
    Monitor.Pulse(lockOn); // разрешить выполнение метода Tock()
    Monitor.Wait(lockOn);  // ожидать завершения метода Tock()
  }
}
```

Прежде всего обратите внимание на код метода `Tick()` в блоке `lock`. Напомним, что методы `Wait()` и `Pulse()` могут использоваться только в синхронизированных блоках кода. В начале метода `Tick()` проверяется значение текущего параметра, которое служит явным признаком остановки часов. Если это логическое значение `false`, то часы остановлены. В этом случае вызывается метод `Pulse()`, разрешающий выполнение любого потока, ожидающего своей очереди. Мы еще вернемся к этому моменту в дальнейшем. Если же часы идут при выполнении метода `Tick()`, то на экран выводится слово "тик" с пробелом, затем вызывается метод `Pulse()`, а после него — метод

Wait(). При вызове метода Pulse() разрешается выполнение потока для того же самого объекта, а при вызове метода Wait() выполнение метода Tick() приостанавливается до тех пор, пока метод Pulse() не будет вызван из другого потока. Таким образом, когда вызывается метод Tick(), отображается одно слово "тик" с пробелом, разрешается выполнение другого потока, а затем выполнение данного метода приостанавливается.

Метод Tock() является точной копией метода Tick(), за исключением того, что он выводит на экран слово "так". Таким образом, при входе в метод Tock() на экран выводится слово "так", вызывается метод Pulse(), а затем выполнение метода Tock() приостанавливается. Методы Tick() и Tock() можно рассматривать как поочередно сменяющие друг друга, т.е. они взаимно синхронизированы.

Когда часы остановлены, метод Pulse() вызывается для того, чтобы обеспечить успешный вызов метода Wait(). Напомним, что метод Wait() вызывается в обоих методах, Tick() и Tock(), после вывода соответствующего слова на экран. Но дело в том, что когда часы остановлены, один из этих методов все еще находится в состоянии ожидания. Поэтому завершающий вызов метода Pulse() требуется, чтобы выполнить ожидающий метод до конца. В качестве эксперимента попробуйте удалить этот вызов метода Pulse() и понаблюдайте за тем, что при этом произойдет. Вы сразу же обнаружите, что программа "зависает", и для выхода из нее придется нажать комбинацию клавиш <Ctrl+C>. Дело в том, что когда метод Wait() вызывается в последнем вызове метода Tock(), соответствующий ему метод Pulse() не вызывается, а значит, выполнение метода Tock() оказывается незавершенным, и он ожидает своей очереди до бесконечности.

Прежде чем переходить к чтению следующего раздела, убедитесь сами, если, конечно, сомневаетесь, в том, что следует обязательно вызывать методы Wait() и Pulse(), чтобы имитируемые часы шли правильно. Для этого подставьте приведенный ниже вариант класса TickTock в рассматриваемую здесь программу. В этом варианте все вызовы методов Wait() и Pulse() исключены.

```
// Нерабочий вариант класса TickTock.

class TickTock {

  object lockOn = new object();

  public void Tick(bool running) {
    lock(lockOn) {
      if(!running) { // остановить часы
        return;
      }

      Console.Write("тик ");
    }
  }
  public void Tock(bool running) {
    lock(lockOn) {
      if(!running) { // остановить часы
        return;
      }

      Console.WriteLine("так");
    }
  }
}
```

После этой подстановки результат выполнения данной программы будет выглядеть следующим образом.

```
тик тик тик тик тик так
так
так
так
так
Часы остановлены
```

Очевидно, что методы `Tick()` и `Tock()` больше не синхронизированы!

Взаимоблокировка и состояние гонки

При разработке многопоточных программ следует быть особенно внимательным, чтобы избежать взаимоблокировки и состояний гонок. *Взаимоблокировка*, как подразумевает само название, — это ситуация, в которой один поток ожидает определенных действий от другого потока, а другой поток, в свою очередь, ожидает чего-то от первого потока. В итоге оба потока приостанавливаются, ожидая друг друга, и ни один из них не выполняется. Эта ситуация напоминает двух слишком вежливых людей, каждый из которых настаивает на том, чтобы другой прошел в дверь первым!

На первый взгляд избежать взаимоблокировки нетрудно, но на самом деле не все так просто, ведь взаимоблокировка может возникать окольными путями. В качестве примера рассмотрим класс `TickTock` из предыдущей программы. Как пояснялось выше, в отсутствие завершающего вызова метода `Pulse()` из метода `Tick()` или `Tock()` тот или другой будет ожидать до бесконечности, что приведет к "зависанию" программы вследствие взаимоблокировки. Зачастую причину взаимоблокировки не так-то просто выяснить, анализируя исходный код программы, поскольку параллельно действующие процессы могут взаимодействовать довольно сложным образом во время выполнения. Для исключения взаимоблокировки требуется внимательное программирование и тщательное тестирование. В целом, если многопоточная программа периодически "зависает", то наиболее вероятной причиной этого является взаимоблокировка.

Состояние гонки возникает в том случае, когда два потока или больше пытаются одновременно получить доступ к общему ресурсу без должной синхронизации. Так, в одном потоке может сохраняться значение в переменной, а в другом — инкрементироваться текущее значение этой же переменной. В отсутствие синхронизации конечный результат будет зависеть от того, в каком именно порядке выполняются потоки: инкрементируется ли значение переменной во втором потоке или же оно сохраняется в первом. О подобной ситуации говорят, что потоки "гоняются друг за другом", причем конечный результат зависит от того, какой из потоков завершится первым. Возникающее состояние гонок, как и взаимоблокировку, непросто обнаружить. Поэтому его лучше предотвратить, синхронизируя должным образом доступ к общим ресурсам при программировании.

Применение атрибута `MethodImplAttribute`

Метод может быть полностью синхронизирован с помощью атрибута `MethodImplAttribute`. Такой подход может стать альтернативой оператору `lock` в тех случаях, когда метод требуется заблокировать полностью. Атрибут

MethodImplAttribute определен в пространстве имен System.Runtime.CompilerServices. Ниже приведен конструктор, применяемый для подобной синхронизации:

```
public MethodImplAttribute(MethodImplOptions methodImplOptions)
```

где *methodImplOptions* обозначает атрибут реализации. Для синхронизации метода достаточно указать атрибут MethodImplOptions.Synchronized. Этот атрибут вызывает блокировку всего метода для текущего экземпляра объекта, доступного по ссылке this. Если же метод относится к типу static, то блокируется его тип. Поэтому данный атрибут непригоден для применения в открытых объектах или классах.

Ниже приведена еще одна версия программы, имитирующей тиканье часов, с переделанным вариантом класса TickTock, в котором атрибут MethodImplOptions обеспечивает должную синхронизацию.

```
// Использовать атрибут MethodImplAttribute для синхронизации метода.

using System;
using System.Threading;
using System.Runtime.CompilerServices;

// Вариант класса TickTock, переделанный с целью
// использовать атрибут MethodImplOptions.Synchronized.
class TickTock {

  /* Следующий атрибут полностью синхронизирует метод Tick(). */
  [MethodImplAttribute(MethodImplOptions.Synchronized)]
  public void Tick(bool running) {
    if(!running) { // остановить часы
      Monitor.Pulse(this); // уведомить любые ожидающие потоки
      return;
    }

    Console.Write("тик ");
    Monitor.Pulse(this); // разрешить выполнение метода Tock()
    Monitor.Wait(this);  // ожидать завершения метода Tock()
  }

  /* Следующий атрибут полностью синхронизирует метод Tock(). */
  [MethodImplAttribute(MethodImplOptions.Synchronized)]
  public void Tock(bool running) {
    if(!running) { // остановить часы
      Monitor.Pulse(this); // уведомить любые ожидающие потоки
      return;
    }

    Console.WriteLine("так");
    Monitor.Pulse(this); // разрешить выполнение метода Tick()
    Monitor.Wait(this);  // ожидать завершения метода Tick()
  }
}

class MyThread {
  public Thread Thrd;
  TickTock ttOb;
```

```
  // Сконструировать новый поток.
  public MyThread(string name, TickTock tt) {
    Thrd = new Thread(this.Run);
    ttOb = tt;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Начать выполнение нового потока.
  void Run() {
    if(Thrd.Name == "Tick") {
      for(int i=0; i<5; i++) ttOb.Tick(true);
        ttOb.Tick(false);
    }
    else {
    for(int i=0; i<5; i++) ttOb.Tock(true);
      ttOb.Tock(false);
    }
  }
}

class TickingClock {
  static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Tick", tt);
    MyThread mt2 = new MyThread("Tock", tt);

    mt1.Thrd.Join();
    mt2.Thrd.Join();
    Console.WriteLine("Часы остановлены");
  }
}
```

Эта версия программы дает такой же результат, как и предыдущая.

Синхронизируемый метод не определен в открытом классе и не вызывается для открытого объекта, поэтому применение оператора `lock` или атрибута `MethodImplAttribute` зависит от личных предпочтений. Ведь и тот и другой дает один и тот же результат. Но поскольку ключевое слово `lock` относится непосредственно к языку C#, то в примерах, приведенных в этой книге, предпочтение отдано именно ему.

ПРИМЕЧАНИЕ

Не применяйте атрибут `MethodImplAttribute` в открытых классах или экземплярах открытых объектов. Вместо этого пользуйтесь оператором `lock`, чтобы заблокировать метод для закрытого объекта, как пояснялось ранее.

Применение мьютекса и семафора

В большинстве случаев, когда требуется синхронизация, оказывается достаточно и оператора `lock`. Тем не менее в некоторых случаях, как, например, при ограничении

доступа к общим ресурсам, более удобными оказываются механизмы синхронизации, встроенные в среду .NET Framework. Ниже рассматриваются по порядку два таких механизма: мьютекс и семафор.

Мьютекс

Мьютекс представляет собой взаимно исключающий синхронизирующий объект. Это означает, что он может быть получен потоком только по очереди. Мьютекс предназначен для тех ситуаций, в которых общий ресурс может быть одновременно использован только в одном потоке. Допустим, что системный журнал совместно используется в нескольких процессах, но только в одном из них данные могут записываться в файл этого журнала в любой момент времени. Для синхронизации процессов в данной ситуации идеально подходит мьютекс.

Мьютекс поддерживается в классе `System.Threading.Mutex`. У него имеется несколько конструкторов. Ниже приведены два наиболее употребительных конструктора.

```
public Mutex()
public Mutex(bool initiallyOwned)
```

В первой форме конструктора создается мьютекс, которым первоначально никто не владеет. А во второй форме исходным состоянием мьютекса завладевает вызывающий поток, если параметр *initiallyOwned* имеет логическое значение `true`. В противном случае мьютексом никто не владеет.

Для того чтобы получить мьютекс, в коде программы следует вызвать метод `WaitOne()` для этого мьютекса. Метод `WaitOne()` наследуется классом `Mutex` от класса `Thread.WaitHandle`. Ниже приведена его простейшая форма.

```
public bool WaitOne();
```

Метод `WaitOne()` ожидает до тех пор, пока не будет получен мьютекс, для которого он был вызван. Следовательно, этот метод блокирует выполнение вызывающего потока до тех пор, пока не станет доступным указанный мьютекс. Он всегда возвращает логическое значение `true`.

Когда же в коде больше не требуется владеть мьютексом, он освобождается посредством вызова метода `ReleaseMutex()`, форма которого приведена ниже.

```
public void ReleaseMutex()
```

В этой форме метод `ReleaseMutex()` освобождает мьютекс, для которого он был вызван, что дает возможность другому потоку получить данный мьютекс.

Для применения мьютекса с целью синхронизировать доступ к общему ресурсу упомянутые выше методы `WaitOne()` и `ReleaseMutex()` используются так, как показано в приведенном ниже фрагменте кода.

```
Mutex myMtx = new Mutex();

// ...

myMtx.WaitOne(); // ожидать получения мьютекса

// Получить доступ к общему ресурсу.

myMtx.ReleaseMutex(); // освободить мьютекс
```

При вызове метода `WaitOne()` выполнение соответствующего потока приостанавливается до тех пор, пока не будет получен мьютекс. А при вызове метода `ReleaseMutex()` мьютекс освобождается и затем может быть получен другим потоком. Благодаря такому подходу к синхронизации одновременный доступ к общему ресурсу ограничивается только одним потоком.

В приведенном ниже примере программы описанный выше механизм синхронизации демонстрируется на практике. В этой программе создаются два потока в виде классов `IncThread` и `DecThread`, которым требуется доступ к общему ресурсу: переменной `SharedRes.Count`. В потоке `IncThread` переменная `SharedRes.Count` инкрементируется, а в потоке `DecThread` — декрементируется. Во избежание одновременного доступа обоих потоков к общему ресурсу `SharedRes.Count` этот доступ синхронизируется мьютексом `Mtx`, также являющимся членом класса `SharedRes`.

```
// Применить мьютекс.

using System;
using System.Threading;

// В этом классе содержится общий ресурс(переменная Count),
// а также мьютекс (Mtx), управляющий доступом к ней.
class SharedRes {
  public static int Count = 0;
  public static Mutex Mtx = new Mutex();
}

// В этом потоке переменная SharedRes.Count инкрементируется.
class IncThread {
  int num;
  public Thread Thrd;

  public IncThread(string name, int n) {
    Thrd = new Thread(this.Run);
    num = n;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
    Console.WriteLine(Thrd.Name + " ожидает мьютекс.");

    // Получить мьютекс.
    SharedRes.Mtx.WaitOne();

    Console.WriteLine(Thrd.Name + " получает мьютекс.");

    do {
      Thread.Sleep(500);
      SharedRes.Count++;
      Console.WriteLine("В потоке " + Thrd.Name +
                        ", SharedRes.Count = " + SharedRes.Count);
      num--;
    } while(num > 0);
```

```
    Console.WriteLine(Thrd.Name + " освобождает мьютекс.");

    // Освободить мьютекс.
    SharedRes.Mtx.ReleaseMutex();
  }
}

// В этом потоке переменная SharedRes.Count декрементируется.
class DecThread {
  int num;
  public Thread Thrd;

  public DecThread(string name, int n) {
    Thrd = new Thread(new ThreadStart(this.Run));
    num = n;
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
    Console.WriteLine(Thrd.Name + " ожидает мьютекс.");

    // Получить мьютекс.
    SharedRes.Mtx.WaitOne();

    Console.WriteLine(Thrd.Name + " получает мьютекс.");

    do {
      Thread.Sleep(500);
      SharedRes.Count--;
      Console.WriteLine("В потоке " + Thrd.Name +
                        ", SharedRes.Count = " + SharedRes.Count);
      num--;
    } while(num > 0);

    Console.WriteLine(Thrd.Name + " освобождает мьютекс.");

    // Освободить мьютекс.
    SharedRes.Mtx.ReleaseMutex();
  }
}

class MutexDemo {
  static void Main() {

    // Сконструировать два потока.
    IncThread mt1 = new IncThread("Инкрементирующий Поток", 5);

    Thread.Sleep(1); // разрешить инкрементирующему потоку начаться

    DecThread mt2 = new DecThread("Декрементирующий Поток", 5);

    mt1.Thrd.Join();
```

```
    mt2.Thrd.Join();
  }
}
```

Эта программа дает следующий результат.

```
Инкрементирующий Поток ожидает мьютекс.
Инкрементирующий Поток получает мьютекс.
Декрементирующий Поток ожидает мьютекс.
В потоке Инкрементирующий Поток, SharedRes.Count = 1
В потоке Инкрементирующий Поток, SharedRes.Count = 2
В потоке Инкрементирующий Поток, SharedRes.Count = 3
В потоке Инкрементирующий Поток, SharedRes.Count = 4
В потоке Инкрементирующий Поток, SharedRes.Count = 5
Инкрементирующий Поток освобождает мьютекс.
Декрементирующий Поток получает мьютекс.
В потоке Декрементирующий Поток, SharedRes.Count = 4
В потоке Декрементирующий Поток, SharedRes.Count = 3
В потоке Декрементирующий Поток, SharedRes.Count = 2
В потоке Декрементирующий Поток, SharedRes.Count = 1
В потоке Декрементирующий Поток, SharedRes.Count = 0
Декрементирующий Поток освобождает мьютекс.
```

Как следует из приведенного выше результата, доступ к общему ресурсу (переменной `SharedRes.Count`) синхронизирован, и поэтому значение данной переменной может быть одновременно изменено только в одном потоке.

Для того чтобы убедиться в том, что мьютекс необходим для получения приведенного выше результата, попробуйте закомментировать вызовы методов `WaitOne()` и `ReleaseMutex()` в исходном коде рассматриваемой здесь программы. При ее последующем выполнении вы получите следующий результат, хотя у вас он может оказаться несколько иным.

```
В потоке Инкрементирующий Поток, SharedRes.Count = 1
В потоке Декрементирующий Поток, SharedRes.Count = 0
В потоке Инкрементирующий Поток, SharedRes.Count = 1
В потоке Декрементирующий Поток, SharedRes.Count = 0
В потоке Инкрементирующий Поток, SharedRes.Count = 1
В потоке Декрементирующий Поток, SharedRes.Count = 0
В потоке Инкрементирующий Поток, SharedRes.Count = 1
В потоке Декрементирующий Поток, SharedRes.Count = 0
В потоке Инкрементирующий Поток, SharedRes.Count = 1
```

Как следует из приведенного выше результата, без мьютекса инкрементирование и декрементирование переменной `SharedRes.Count` происходит, скорее, беспорядочно, чем последовательно.

Мьютекс, созданный в предыдущем примере, известен только тому процессу, который его породил. Но мьютекс можно создать и таким образом, чтобы он был известен где-нибудь еще. Для этого он должен быть именованным. Ниже приведены формы конструктора, предназначенные для создания такого мьютекса.

```
public Mutex(bool initiallyOwned, string имя)
public Mutex(bool initiallyOwned, string имя, out bool createdNew)
```

В обеих формах конструктора *имя* обозначает конкретное имя мьютекса. Если в первой форме конструктора параметр *initiallyOwned* имеет логическое значение

true, то владение мьютексом запрашивается. Но поскольку мьютекс может принадлежать другому процессу на системном уровне, то для этого параметра лучше указать логическое значение false. А после возврата из второй формы конструктора параметр *createdNew* будет иметь логическое значение true, если владение мьютексом было запрошено и получено, и логическое значение false, если запрос на владение был отклонен. Существует и третья форма конструктора типа Mutex, в которой допускается указывать управляющий доступом объект типа MutexSecurity. С помощью именованных мьютексов можно синхронизировать взаимодействие процессов.

И последнее замечание: в потоке, получившем мьютекс, допускается делать один или несколько дополнительных вызовов метода WaitOne() перед вызовом метода ReleaseMutex(), причем все эти дополнительные вызовы будут произведены успешно. Это означает, что дополнительные вызовы метода WaitOne() не будут блокировать поток, который уже владеет мьютексом. Но количество вызовов метода WaitOne() должно быть равно количеству вызовов метода ReleaseMutex() перед освобождением мьютекса.

Семафор

Семафор подобен мьютексу, за исключением того, что он предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам. Поэтому семафор пригоден для синхронизации целого ряда ресурсов. Семафор управляет доступом к общему ресурсу, используя для этой цели счетчик. Если значение счетчика больше нуля, то доступ к ресурсу разрешен. А если это значение равно нулю, то доступ к ресурсу запрещен. С помощью счетчика ведется подсчет количества *разрешений*. Следовательно, для доступа к ресурсу поток должен получить разрешение от семафора.

Обычно поток, которому требуется доступ к общему ресурсу, пытается получить разрешение от семафора. Если значение счетчика семафора больше нуля, то поток получает разрешение, а счетчик семафора декрементируется. В противном случае поток блокируется до тех пор, пока не получит разрешение. Когда же потоку больше не требуется доступ к общему ресурсу, он высвобождает разрешение, а счетчик семафора инкрементируется. Если разрешения ожидает другой поток, то он получает его в этот момент. Количество одновременно разрешаемых доступов указывается при создании семафора. Так, если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс.

Семафоры особенно полезны в тех случаях, когда общий ресурс состоит из группы или пула ресурсов. Например, пул ресурсов может состоять из целого ряда сетевых соединений, каждое из которых служит для передачи данных. Поэтому потоку, которому требуется сетевое соединение, все равно, какое именно соединение он получит. В данном случае семафор обеспечивает удобный механизм управления доступом к сетевым соединениям.

Семафор реализуется в классе System.Threading.Semaphore, у которого имеется несколько конструкторов. Ниже приведена простейшая форма конструктора данного класса:

```
public Semaphore(int initialCount, int maximumCount)
```

где *initialCount* — это первоначальное значение для счетчика разрешений семафора, т.е. количество первоначально доступных разрешений; *maximumCount* — максимальное значение данного счетчика, т.е. максимальное количество разрешений, которые может дать семафор.

Семафор применяется таким же образом, как и описанный ранее мьютекс. В целях получения доступа к ресурсу в коде программы вызывается метод `WaitOne()` для семафора. Этот метод наследуется классом `Semaphore` от класса `WaitHandle`. Метод `WaitOne()` ожидает до тех пор, пока не будет получен семафор, для которого он вызывается. Таким образом, он блокирует выполнение вызывающего потока до тех пор, пока указанный семафор не предоставит разрешение на доступ к ресурсу.

Если коду больше не требуется владеть семафором, он освобождает его, вызывая метод `Release()`. Ниже приведены две формы этого метода.

```
public int Release()
public int Release(int releaseCount)
```

В первой форме метод `Release()` высвобождает только одно разрешение, а во второй форме — количество разрешений, определяемых параметром *`releaseCount`*. В обеих формах данный метод возвращает подсчитанное количество разрешений, существовавших до высвобождения.

Метод `WaitOne()` допускается вызывать в потоке несколько раз перед вызовом метода `Release()`. Но количество вызовов метода `WaitOne()` должно быть равно количеству вызовов метода `Release()` перед высвобождением разрешения. С другой стороны, можно воспользоваться формой вызова метода `Release(int num)`, чтобы передать количество высвобождаемых разрешений, равное количеству вызовов метода `WaitOne()`.

Ниже приведен пример программы, в которой демонстрируется применение семафора. В этой программе семафор используется в классе `MyThread` для одновременного выполнения только двух потоков типа `MyThread`. Следовательно, разделяемым ресурсом в данном случае является ЦП.

```
// Использовать семафор.

using System;
using System.Threading;

// Этот поток разрешает одновременное выполнение
// только двух своих экземпляров.
class MyThread {
  public Thread Thrd;

  // Здесь создается семафор, дающий только два
  // разрешения из двух первоначально имеющихся.
  static Semaphore sem = new Semaphore(2, 2);

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {

    Console.WriteLine(Thrd.Name + " ожидает разрешения.");

    sem.WaitOne();
```

```
    Console.WriteLine(Thrd.Name + " получает разрешение.");

    for(char ch='A'; ch < 'D'; ch++) {
      Console.WriteLine(Thrd.Name + " : " + ch + " ");
      Thread.Sleep(500);
    }

    Console.WriteLine(Thrd.Name + " высвобождает разрешение.");

    // Освободить семафор.
    sem.Release();
  }
}

class SemaphoreDemo {
  static void Main() {

    // Сконструировать три потока.
    MyThread mt1 = new MyThread("Поток #1");
    MyThread mt2 = new MyThread("Поток #2");
    MyThread mt3 = new MyThread("Поток #3");

    mt1.Thrd.Join();
    mt2.Thrd.Join();
    mt3.Thrd.Join();
  }
}
```

В классе `MyThread` объявляется семафор `sem`, как показано ниже.

```
static Semaphore sem = new Semaphore(2, 2);
```

При этом создается семафор, способный дать не более двух разрешений на доступ к ресурсу из двух первоначально имеющихся разрешений.

Обратите внимание на то, что выполнение метода `MyThread.Run()` не может быть продолжено до тех пор, пока семафор `sem` не даст соответствующее разрешение. Если разрешение отсутствует, то выполнение потока приостанавливается. Когда же разрешение появляется, выполнение потока возобновляется. В методе `In Main()` создаются три потока. Но выполняться могут только два первых потока, а третий должен ожидать окончания одного из этих двух потоков. Ниже приведен результат выполнения рассматриваемой здесь программы, хотя у вас он может оказаться несколько иным.

```
Поток #1 ожидает разрешения.
Поток #1 получает разрешение.
Поток #1 : A
Поток #2 ожидает разрешения.
Поток #2 получает разрешение.
Поток #2 : A
Поток #3 ожидает разрешения.
Поток #1 : B
Поток #2 : B
Поток #1 : C
Поток #2 : C
Поток #1 высвобождает разрешение.
```

```
Поток #3 получает разрешение.
Поток #3 : A
Поток #2 высвобождает разрешение.
Поток #3 : B
Поток #3 : C
Поток #3 высвобождает разрешение.
```

Семафор, созданный в предыдущем примере, известен только тому процессу, который его породил. Но семафор можно создать и таким образом, чтобы он был известен где-нибудь еще. Для этого он должен быть именованным. Ниже приведены формы конструктора класса `Semaphore`, предназначенные для создания такого семафора.

```
public Semaphore(int initialCount, int maximumCount, string имя)
public Semaphore(int initialCount, int maximumCount, string имя,
                 out bool createdNew)
```

В обеих формах *имя* обозначает конкретное имя, передаваемое конструктору. Если в первой форме семафор, на который указывает *имя*, еще не существует, то он создается с помощью значений, определяемых параметрами `initialCount` и `maximumCount`. А если он уже существует, то значения параметров `initialCount` и `maximumCount` игнорируются. После возврата из второй формы конструктора параметр `createdNew` будет иметь логическое значение `true`, если семафор был создан. В этом случае значения параметров `initialCount` и `maximumCount` используются для создания семафора. Если же параметр `createdNew` будет иметь логическое значение `false`, значит, семафор уже существует и значения параметров `initialCount` и `maximumCount` игнорируются. Существует и третья форма конструктора класса `Semaphore`, в которой допускается указывать управляющий доступом объект типа `SemaphoreSecurity`. С помощью именованных семафоров можно синхронизировать взаимодействие процессов.

Применение событий

Для синхронизации в C# предусмотрен еще один тип объекта: событие. Существуют две разновидности событий: устанавливаемые в исходное состояние вручную и автоматически. Они поддерживаются в классах `ManualResetEvent` и `AutoResetEvent` соответственно. Эти классы являются производными от класса `EventWaitHandle`, находящегося на верхнем уровне иерархии классов, и применяются в тех случаях, когда один поток ожидает появления некоторого события в другом потоке. Как только такое событие появляется, второй поток уведомляет о нем первый поток, позволяя тем самым возобновить его выполнение.

Ниже приведены конструкторы классов `ManualResetEvent` и `AutoResetEvent`.

```
public ManualResetEvent(bool initialState)
public AutoResetEvent(bool initialState)
```

Если в обеих формах параметр `initialState` имеет логическое значение `true`, то о событии первоначально уведомляется. А если он имеет логическое значение `false`, то о событии первоначально не уведомляется.

Применяются события очень просто. Так, для события типа `ManualResetEvent` порядок применения следующий. Поток, ожидающий некоторое событие, вызывает метод `WaitOne()` для событийного объекта, представляющего данное событие. Если событийный объект находится в сигнальном состоянии, то происходит немедленный

возврат из метода WaitOne(). В противном случае выполнение вызывающего потока приостанавливается до тех пор, пока не будет получено уведомление о событии. Как только событие произойдет в другом потоке, этот поток установит событийный объект в сигнальное состояние, вызвав метод Set(). Поэтому метод Set() следует рассматривать как уведомляющий о том, что событие произошло. После установки событийного объекта в сигнальное состояние произойдет немедленный возврат из метода WaitOne(), и первый поток возобновит свое выполнение. А в результате вызова метода Reset() событийный объект возвращается в несигнальное состояние.

Событие типа AutoResetEvent отличается от события типа ManualResetEvent лишь способом установки в исходное состояние. Если для события типа ManualResetEvent событийный объект остается в сигнальном состоянии до тех пор, пока не будет вызван метод Reset(), то для события типа AutoResetEvent событийный объект автоматически переходит в несигнальное состояние, как только поток, ожидающий это событие, получит уведомление о нем и возобновит свое выполнение. Поэтому если применяется событие типа AutoResetEvent, то вызывать метод Reset() необязательно.

В приведенном ниже примере программы демонстрируется применение события типа ManualResetEvent.

```
// Использовать событийный объект, устанавливаемый
// в исходное состояние вручную.

using System;
using System.Threading;

// Этот поток уведомляет о том, что событие передано его конструктору.
class MyThread {
  public Thread Thrd;
  ManualResetEvent mre;

  public MyThread(string name, ManualResetEvent evt) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    mre = evt;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
    Console.WriteLine("Внутри потока " + Thrd.Name);

    for(int i=0; i<5; i++) {
      Console.WriteLine(Thrd.Name);
      Thread.Sleep(500);
    }

    Console.WriteLine(Thrd.Name + " завершен!");

    // Уведомить о событии.
    mre.Set();
  }
}
```

```
class ManualEventDemo {
  static void Main() {
    ManualResetEvent evtObj = new ManualResetEvent(false);

    MyThread mt1 = new MyThread("Событийный Поток 1", evtObj);

    Console.WriteLine("Основной поток ожидает событие.");

    // Ожидать уведомления о событии.
    evtObj.WaitOne();

    Console.WriteLine("Основной поток получил " +
                      "уведомление о событии от первого потока.");

    // Установить событийный объект в исходное состояние.
    evtObj.Reset();

    mt1 = new MyThread("Событийный Поток 2", evtObj);

    // Ожидать уведомления о событии.
    evtObj.WaitOne();

    Console.WriteLine("Основной поток получил " +
                      "уведомление о событии от второго потока.");
  }
}
```

Ниже приведен результат выполнения рассматриваемой здесь программы, хотя у вас он может оказаться несколько иным.

```
В потоке Событийный Поток 1
Событийный Поток 1
Основной поток ожидает событие.
Событийный Поток 1
Событийный Поток 1
Событийный Поток 1
Событийный Поток 1
Событийный Поток 1 завершен!
Основной поток получил уведомление о событии от первого потока.
В потоке Событийный Поток 2
Событийный Поток 2
Событийный Поток 2
Событийный Поток 2
Событийный Поток 2
Событийный Поток 2
Событийный Поток 2 завершен!
Основной поток получил уведомление о событии от второго потока.
```

Прежде всего обратите внимание на то, что событие типа `ManualResetEvent` передается непосредственно конструктору класса `MyThread`. Когда завершается метод `Run()` из класса `MyThread`, он вызывает для событийного объекта метод `Set()`, устанавливающий этот объект в сигнальное состояние. В методе `Main()` формируется событийный объект `evtObj` типа `ManualResetEvent`, первоначально устанавливаемый в исходное, несигнальное состояние. Затем создается экземпляр объекта типа

MyThread, которому передается событийный объект evtObj. После этого основной поток ожидает уведомления о событии. А поскольку событийный объект evtObj первоначально находится в несигнальном состоянии, то основной поток вынужден ожидать до тех пор, пока для экземпляра объекта типа MyThread не будет вызван метод Set(), устанавливающий событийный объект evtObj в сигнальное состояние. Это дает возможность основному потоку возобновить свое выполнение. Затем событийный объект устанавливается в исходное состояние, и весь процесс повторяется, но на этот раз для второго потока. Если бы не событийный объект, то все потоки выполнялись бы одновременно, а результаты их выполнения оказались бы окончательно запутанными. Для того чтобы убедиться в этом, попробуйте закомментировать вызов метода WaitOne() в методе Main().

Если бы в рассматриваемой здесь программе событийный объект типа AutoResetEvent использовался вместо событийного объекта типа ManualResetEvent, то вызывать метод Reset() в методе Main() не пришлось бы. Ведь в этом случае событийный объект автоматически устанавливается в несигнальное состояние, когда поток, ожидающий данное событие, возобновляет свое выполнение. Для опробования этой разновидности события замените в данной программе все ссылки на объект типа ManualResetEvent ссылками на объект типа AutoResetEvent и удалите все вызовы метода Reset(). Видоизмененная версия программы будет работать так же, как и прежде.

Класс Interlocked

Еще одним классом, связанным с синхронизацией, является класс Interlocked. Этот класс служит в качестве альтернативы другим средствам синхронизации, когда требуется только изменить значение общей переменной. Методы, доступные в классе Interlocked, гарантируют, что их действие будет выполняться как единая, непрерываемая операция. Это означает, что никакой синхронизации в данном случае вообще не требуется. В классе Interlocked предоставляются статические методы для сложения двух целых значений, инкрементирования и декрементирования целого значения, сравнения и установки значений объекта, обмена объектами и получения 64-разрядного значения. Все эти операции выполняются без прерывания.

В приведенном ниже примере программы демонстрируется применение двух методов из класса Interlocked: Increment() и Decrement(). При этом используются следующие формы обоих методов:

```
public static int Increment(ref int location)
public static int Decrement(ref int location)
```

где *location* — это переменная, которая подлежит инкрементированию или декрементированию.

```
// Использовать блокируемые операции.

using System;
using System.Threading;

// Общий ресурс.
class SharedRes {
  public static int Count = 0;
}
```

```
// В этом потоке переменная SharedRes.Count инкрементируется.
class IncThread {
  public Thread Thrd;

  public IncThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
    for(int i=0; i<5; i++) {
      Interlocked.Increment(ref SharedRes.Count);
      Console.WriteLine(Thrd.Name + " Count = " + SharedRes.Count);
    }
  }
}

// В этом потоке переменная SharedRes.Count декрементируется.
class DecThread {
  public Thread Thrd;

  public DecThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Точка входа в поток.
  void Run() {
    for(int i=0; i<5; i++) {
      Interlocked.Decrement(ref SharedRes.Count);
      Console.WriteLine(Thrd.Name + " Count = " + SharedRes.Count);
    }
  }
}

class InterlockedDemo {
  static void Main() {

    // Сконструировать два потока.
    IncThread mt1 = new IncThread("Инкрементирующий Поток");
    DecThread mt2 = new DecThread("Декрементирующий Поток");

    mt1.Thrd.Join();
    mt2.Thrd.Join();
  }
}
```

Классы синхронизации, внедренные в версии .NET Framework 4.0

Рассматривавшиеся ранее классы синхронизации, в том числе `Semaphore` и `AutoResetEvent`, были доступны в среде .NET Framework, начиная с версии 1.1.

Таким образом, эти классы образуют основу поддержки синхронизации в среде .NET Framework. Но после выпуска версии .NET Framework 4.0 появился ряд новых альтернатив этим классам синхронизации. Все они перечисляются ниже.

Класс	Назначение
`Barrier`	Вынуждает потоки ожидать появления всех остальных потоков в указанной точке, называемой *барьерной*
`CountdownEvent`	Выдает сигнал, когда обратный отсчет завершается
`ManualResetEventSlim`	Это упрощенный вариант класса `ManualResetEvent`
`SemaphoreSlim`	Это упрощенный вариант класса `Semaphore`

Если вам понятно, как пользоваться основными классами синхронизации, описанными ранее в этой главе, то у вас не должно возникнуть затруднений при использовании их новых альтернатив и дополнений.

Прерывание потока

Иногда поток полезно прервать до его нормального завершения. Например, отладчику может понадобиться прервать вышедший из-под контроля поток. После прерывания поток удаляется из системы и не может быть начат снова.

Для прерывания потока до его нормального завершения служит метод `Thread.Abort()`. Ниже приведена простейшая форма этого метода.

```
public void Abort()
```

Метод `Abort()` создает необходимые условия для генерирования исключения `ThreadAbortException` в том потоке, для которого он был вызван. Это исключение приводит к прерыванию потока и может быть перехвачено и в коде программы, но в этом случае оно автоматически генерируется еще раз, чтобы остановить поток. Метод `Abort()` не всегда способен остановить поток немедленно, поэтому если поток требуется остановить перед тем, как продолжить выполнение программы, то после метода `Abort()` следует сразу же вызвать метод `Join()`. Кроме того, в самых редких случаях методу `Abort()` вообще не удается остановить поток. Это происходит, например, в том случае, если кодовый блок `finally` входит в бесконечный цикл.

В приведенном ниже примере программы демонстрируется применение метода `Abort()` для прерывания потока.

```
// Прервать поток с помощью метода Abort().

using System;
using System.Threading;

class MyThread {
  public Thread Thrd;

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }
```

```
  // Это точка входа в поток.
  void Run() {
    Console.WriteLine(Thrd.Name + " начат.");

    for(int i = 1; i <= 1000; i++) {
      Console.Write(i + " ");
      if((i%10)==0) {
        Console.WriteLine();
        Thread.Sleep(250);
      }
    }

    Console.WriteLine(Thrd.Name + " завершен.");
  }
}

class StopDemo {
  static void Main() {
    MyThread mt1 = new MyThread("Мой Поток");

    Thread.Sleep(1000); // разрешить порожденному потоку начать свое выполнение

    Console.WriteLine("Прерывание потока.");
    mt1.Thrd.Abort();

    mt1.Thrd.Join(); // ожидать прерывания потока

    Console.WriteLine("Основной поток прерван.");
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Мой Поток начат
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Прерывание потока.
Основной поток прерван.
```

ПРИМЕЧАНИЕ

Метод `Abort()` не следует применять в качестве обычного средства прерывания потока, поскольку он предназначен для особых случаев. Обычно поток должен завершаться естественным образом, чтобы произошел возврат из метода, выполняющего роль точки входа в него.

Другая форма метода `Abort()`

В некоторых случаях оказывается полезной другая форма метода `Abort()`, приведенная ниже в общем виде:

```
public void Abort(object stateInfo)
```

где *stateInfo* обозначает любую информацию, которую требуется передать потоку, когда он останавливается. Эта информация доступна посредством свойства ExceptionState из класса исключения ThreadAbortException. Подобным образом потоку можно передать код завершения. В приведенном ниже примере программы демонстрируется применение данной формы метода Abort().

```
// Использовать форму метода Abort(object stateInfo).

using System;
using System.Threading;

class MyThread {
  public Thread Thrd;

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Это точка входа в поток.
  void Run() {
    try {
      Console.WriteLine(Thrd.Name + " начат.");

      for(int i = 1; i <= 1000; i++) {
        Console.Write(i + " ");
        if((i%10)==0) {
          Console.WriteLine();
          Thread.Sleep(250);
        }
      }
      Console.WriteLine(Thrd.Name + " завершен нормально.");
    } catch(ThreadAbortException exc) {
      Console.WriteLine("Поток прерван, код завершения " +
                        exc.ExceptionState);
    }
  }
}

class UseAltAbort {
  static void Main() {
    MyThread mt1 = new MyThread("Мой Поток");

    Thread.Sleep(1000); // разрешить порожденному потоку начать свое выполнение

    Console.WriteLine("Прерывание потока.");

    mt1.Thrd.Abort(100);

    mt1.Thrd.Join(); // ожидать прерывания потока
```

```
    Console.WriteLine("Основной поток прерван.");
  }
}
```

Эта программа дает следующий результат.

```
Мой Поток начат
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Прерывание потока.
Поток прерван, код завершения 100
Основной поток прерван.
```

Как следует из приведенного выше результата, значение 100 передается методу `Abort()` в качестве кода прерывания. Это значение становится затем доступным посредством свойства `ExceptionState` из класса исключения `ThreadAbortException`, которое перехватывается потоком при его прерывании.

Отмена действия метода `Abort()`

Запрос на преждевременное прерывание может быть переопределен в самом потоке. Для этого необходимо сначала перехватить в потоке исключение `ThreadAbortException`, а затем вызвать метод `ResetAbort()`. Благодаря этому исключается повторное генерирование исключения по завершении обработчика исключения, прерывающего данный поток. Ниже приведена форма объявления метода `ResetAbort()`.

```
public static void ResetAbort()
```

Вызов метода `ResetAbort()` может завершиться неудачно, если в потоке отсутствует надлежащий режим надежной отмены преждевременного прерывания потока.

В приведенном ниже примере программы демонстрируется применение метода `ResetAbort()`.

```
// Использовать метод ResetAbort().

using System;
using System.Threading;

class MyThread {
  public Thread Thrd;

  public MyThread(string name) {
    Thrd = new Thread(this.Run);
    Thrd.Name = name;
    Thrd.Start();
  }

  // Это точка входа в поток.
  void Run() {
    Console.WriteLine(Thrd.Name + " начат.");

    for(int i = 1; i <= 1000; i++) {
```

```
      try {
        Console.Write(i + " ");
        if((i%10)==0) {
          Console.WriteLine();
          Thread.Sleep(250);
        }
      } catch(ThreadAbortException exc) {
        if((int)exc.ExceptionState == 0) {
          Console.WriteLine("Прерывание потока отменено! " +
                            "Код завершения " + exc.ExceptionState);
          Thread.ResetAbort();
        }
        else
          Console.WriteLine("Поток прерван, код завершения " +
                            exc.ExceptionState);
      }
    }
    Console.WriteLine(Thrd.Name + " завершен нормально.");
  }
}

class ResetAbort {
  static void Main() {
    MyThread mt1 = new MyThread("Мой Поток");

    Thread.Sleep(1000); // разрешить порожденному потоку начать свое выполнение

    Console.WriteLine("Прерывание потока.");
    mt1.Thrd.Abort(0); // это не остановит поток

    Thread.Sleep(1000); // разрешить порожденному потоку выполняться подольше

    Console.WriteLine("Прерывание потока.");
    mt1.Thrd.Abort(100); // а это остановит поток

    mt1.Thrd.Join(); // ожидать прерывания потока

    Console.WriteLine("Основной поток прерван.");
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Мой Поток начат
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Прерывание потока.
Прерывание потока отменено! Код завершения 0
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Поток прерван, код завершения 100
Основной поток прерван.
```

Если в данном примере программы метод `Abort()` вызывается с нулевым аргументом, то запрос на преждевременное прерывание отменяется потоком, вызывающим метод `ResetAbort()`, и выполнение этого потока продолжается. Любое другое значение аргумента приведет к прерыванию потока.

Приостановка и возобновление потока

В первоначальных версиях среды .NET Framework поток можно было приостановить вызовом метода `Thread.Suspend()` и возобновить вызовом метода `Thread.Resume()`. Но теперь оба эти метода считаются устаревшими и не рекомендуются к применению в новом коде. Объясняется это, в частности, тем, что пользоваться методом `Suspend()` на самом деле небезопасно, так как с его помощью можно приостановить поток, который в настоящий момент удерживает блокировку, что препятствует ее снятию, а следовательно, приводит к взаимоблокировке. Применение обоих методов может стать причиной серьезных осложнений на уровне системы. Поэтому для приостановки и возобновления потока следует использовать другие средства синхронизации, в том числе мьютекс и семафор.

Определение состояния потока

Состояние потока может быть получено из свойства `ThreadState`, доступного в классе `Thread`. Ниже приведена общая форма этого свойства.

```
public ThreadState ThreadState{ get; }
```

Состояние потока возвращается в виде значения, определенного в перечислении `ThreadState`. Ниже приведены значения, определенные в этом перечислении.

`ThreadState.Aborted`	`ThreadState.AbortRequested`
`ThreadState.Background`	`ThreadState.Running`
`ThreadState.Stopped`	`ThreadState.StopRequested`
`ThreadState.Suspended`	`ThreadState.SuspendRequested`
`ThreadState.Unstarted`	`ThreadState.WaitSleepJoin`

Все эти значения не требуют особых пояснений, за исключением одного. Значение `ThreadState.WaitSleepJoin` обозначает состояние, в которое поток переходит во время ожидания в связи с вызовом метода `Wait()`, `Sleep()` или `Join()`.

Применение основного потока

Как пояснялось в самом начале этой главы, у всякой программы на C# имеется хотя бы один поток исполнения, называемый *основным*. Этот поток программа получает автоматически, как только начинает выполняться. С основным потоком можно обращаться таким же образом, как и со всеми остальными потоками.

Для доступа к основному потоку необходимо получить объект типа `Thread`, который ссылается на него. Это делается с помощью свойства `CurrentThread`, являющегося членом класса `Thread`. Ниже приведена общая форма этого свойства.

```
public static Thread CurrentThread{ get; }
```

Данное свойство возвращает ссылку на тот поток, в котором оно используется. Поэтому если свойство `CurrentThread` используется при выполнении кода в основном потоке, то с его помощью можно получить ссылку на основной поток. Имея в своем распоряжении такую ссылку, можно управлять основным потоком так же, как и любым другим потоком.

В приведенном ниже примере программы сначала получается ссылка на основной поток, а затем получаются и устанавливаются имя и приоритет основного потока.

```
// Продемонстрировать управление основным потоком.

using System;
using System.Threading;

class UseMain {
  static void Main() {
    Thread Thrd;

    // Получить основной поток.
    Thrd = Thread.CurrentThread;

    // Отобразить имя основного потока.
    if(Thrd.Name == null)
      Console.WriteLine("У основного потока нет имени.");
    else
      Console.WriteLine("Основной поток называется: " + Thrd.Name);

    // Отобразить приоритет основного потока.
    Console.WriteLine("Приоритет: " + Thrd.Priority);
    Console.WriteLine();

    // Установить имя и приоритет.
    Console.WriteLine("Установка имени и приоритета.\n");
    Thrd.Name = "Основной Поток";
    Thrd.Priority = ThreadPriority.AboveNormal;

    Console.WriteLine("Теперь основной поток называется: " +
                      Thrd.Name);

    Console.WriteLine("Теперь приоритет: " + Thrd.Priority);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
У основного потока нет имени.
Приоритет: Normal

Установка имени и приоритета.
Теперь основной поток называется: Основной Поток
Теперь приоритет: AboveNormal
```

Следует, однако, быть очень внимательным, выполняя операции с основным потоком. Так, если добавить в конце метода `Main()` следующий вызов метода `Join()`:

```
Thrd.Join();
```

программа никогда не завершится, поскольку она будет ожидать окончания основного потока!

Дополнительные средства многопоточной обработки, внедренные в версии .NET Framework 4.0

В версии .NET Framework 4.0 внедрен ряд новых средств многопоточной обработки, которые могут оказаться весьма полезными. Самым важным среди них является новая система отмены. В этой системе поддерживается механизм отмены потока простым, вполне определенным и структурированным способом. В основу этого механизма положено понятие *признака отмены*, с помощью которого указывается состояние отмены потока. Признаки отмены поддерживаются в классе `CancellationTokenSource` и в структуре `CancellationToken`. Система отмены полностью интегрирована в новую библиотеку распараллеливания задач (TPL), и поэтому она подробнее рассматривается вместе с TPL в главе 24.

В класс `System.Threading` добавлена структура `SpinWait`, предоставляющая методы `SpinOnce()` и `SpinUntil()`, которые обеспечивают более полный контроль над ожиданием в состоянии занятости. Вообще говоря, структура `SpinWait` оказывается непригодной для однопроцессорных систем. А для многопроцессорных систем она применяется в цикле. Еще одним элементом, связанным с ожиданием в состоянии занятости, является структура `SpinLock`, которая применяется в цикле ожидания до тех пор, пока не станет доступной блокировка. В класс `Thread` добавлен метод `Yield()`, который просто выдает остаток кванта времени, выделенного потоку. Ниже приведена общая форма объявления этого метода.

```
public static bool Yield()
```

Этот метод возвращает логическое значение `true`, если происходит переключение контекста. В отсутствие другого потока, готового для выполнения, переключение контекста не произойдет.

Рекомендации по многопоточному программированию

Для эффективного многопоточного программирования самое главное — мыслить категориями параллельного, а не последовательного выполнения кода. Так, если в одной программе имеются две подсистемы, которые могут работать параллельно, их следует организовать в отдельные потоки. Но делать это следует очень внимательно и аккуратно, поскольку если создать слишком много потоков, то тем самым можно значительно снизить, а не повысить производительность программы. Следует также иметь в виду дополнительные издержки, связанные с переключением контекста. Так, если создать слишком много потоков, то на смену контекста уйдет больше времени ЦП, чем на выполнение самой программы! И наконец, для написания нового кода, предназначенного для многопоточной обработки, рекомендуется пользоваться библиотекой распараллеливания задач (TPL), о которой речь пойдет в следующей главе.

Запуск отдельной задачи

Многозадачность на основе потоков чаще всего организуется при программировании на C#. Но там, где это уместно, можно организовать и многозадачность на основе процессов. В этом случае вместо запуска другого потока в одной и той же программе

одна программа начинает выполнение другой. При программировании на C# это делается с помощью класса `Process`, определенного в пространстве имен `System.Diagnostics`. В заключение этой главы вкратце будут рассмотрены особенности запуска и управления другим процессом.

Простейший способ запустить другой процесс — воспользоваться методом `Start()`, определенным в классе `Process`. Ниже приведена одна из самых простых форм этого метода:

```
public static Process Start(string имя_файла)
```

где *имя_файла* обозначает конкретное имя файла, который должен исполняться или же связан с исполняемым файлом.

Когда созданный процесс завершается, следует вызвать метод `Close()`, чтобы освободить память, выделенную для этого процесса. Ниже приведена форма объявления метода `Close()`.

```
public void Close()
```

Процесс может быть прерван двумя способами. Если процесс является приложением Windows с графическим пользовательским интерфейсом, то для прерывания такого процесса вызывается метод `CloseMainWindow()`, форма которого приведена ниже.

```
public bool CloseMainWindow()
```

Этот метод посылает процессу сообщение, предписывающее ему остановиться. Он возвращает логическое значение `true`, если сообщение получено, и логическое значение `false`, если приложение не имеет графического пользовательского интерфейса или главного окна. Следует, однако, иметь в виду, что метод `CloseMainWindow()` служит только для запроса остановки процесса. Если приложение проигнорирует такой запрос, то оно не будет прервано как процесс.

Для безусловного прерывания процесса следует вызвать метод `Kill()`, как показано ниже.

```
public void Kill()
```

Но методом `Kill()` следует пользоваться аккуратно, так как он приводит к неконтролируемому прерыванию процесса. Любые несохраненные данные, связанные с прерываемым процессом, будут, скорее всего, потеряны.

Для того чтобы организовать ожидание завершения процесса, можно воспользоваться методом `WaitForExit()`. Ниже приведены две его формы.

```
public void WaitForExit()
public bool WaitForExit(int миллисекунд)
```

В первой форме ожидание продолжается до тех пор, пока процесс не завершится, а во второй форме — только в течение указанного количества *миллисекунд*. В последнем случае метод `WaitForExit()` возвращает логическое значение `true`, если процесс завершился, и логическое значение `false`, если он все еще выполняется.

В приведенном ниже примере программы демонстрируется создание, ожидание и закрытие процесса. В этой программе сначала запускается стандартная сервисная программа Windows: текстовый редактор WordPad.exe, а затем организуется ожидание завершения программы WordPad как процесса.

```
// Продемонстрировать запуск нового процесса.

using System;
using System.Diagnostics;

class StartProcess {
  static void Main() {
    Process newProc = Process.Start("wordpad.exe");

    Console.WriteLine("Новый процесс запущен.");

    newProc.WaitForExit();

    newProc.Close(); // освободить выделенные ресурсы

    Console.WriteLine("Новый процесс завершен.");
  }
}
```

При выполнении этой программы запускается стандартное приложение WordPad, и на экране появляется сообщение `"Новый процесс запущен."`. Затем программа ожидает закрытия WordPad. По окончании работы WordPad на экране появляется заключительное сообщение `"Новый процесс завершен."`.

24
ГЛАВА

Многопоточное программирование. Часть вторая: библиотека TPL

Вероятно, самым главным среди новых средств, внедренных в версию 4.0 среды .NET Framework, является библиотека распараллеливания задач (TPL). Эта библиотека усовершенствует многопоточное программирование двумя основными способами. Во-первых, она упрощает создание и применение многих потоков. И во-вторых, она позволяет автоматически использовать несколько процессоров. Иными словами, TPL открывает возможности для автоматического масштабирования приложений с целью эффективного использования ряда доступных процессоров. Благодаря этим двух особенностям библиотеки TPL она рекомендуется в большинстве случаев к применению для организации многопоточной обработки.

Еще одним средством параллельного программирования, внедренным в версию 4.0 среды .NET Framework, является параллельный язык интегрированных запросов (PLINQ). Язык PLINQ дает возможность составлять запросы, для обработки которых автоматически используется несколько процессоров, а также принцип параллелизма, когда это уместно. Как станет ясно из дальнейшего, запросить параллельную обработку запроса очень просто. Следовательно, с помощью PLINQ можно без особого труда внедрить параллелизм в запрос.

Главной причиной появления таких важных новшеств, как TPL и PLINQ, служит возросшее значение параллелизма в современном программировании. В настоящее время многоядерные процессоры уже стали обычным явлением. Кроме того, постоянно растет потребность в повышении производительности программ. Все это, в свою очередь, вызвало растущую потребность в механизме, который

позволял бы с выгодой использовать несколько процессов для повышения производительности программного обеспечения. Но дело в том, что в прошлом это было не так-то просто сделать ясным и допускающим масштабирование способом. Изменить это положение, собственно, и призваны TPL и PLINQ. Ведь они дают возможность легче (и безопаснее) использовать системные ресурсы.

Библиотека TPL определена в пространстве имен `System.Threading.Tasks`. Но для работы с ней обычно требуется также включать в программу класс `System.Threading`, поскольку он поддерживает синхронизацию и другие средства многопоточной обработки, в том числе и те, что входят в класс `Interlocked`.

В этой главе рассматривается и TPL, и PLINQ. Следует, однако, иметь в виду, что и та и другая тема довольно обширны. Поэтому в этой главе даются самые основы и рассматриваются некоторые простейшие способы применения TPL и PLINQ. Таким образом, материал этой главы послужит вам в качестве удобной отправной точки для дальнейшего изучения TPL и PLINQ. Если параллельное программирование входит в сферу ваших интересов, то именно эти средства .NET Framework вам придется изучить более основательно.

ПРИМЕЧАНИЕ

Несмотря на то что применение TPL и PLINQ рекомендуется теперь для разработки большинства многопоточных приложений, организация многопоточной обработки на основе класса `Thread`, представленного в главе 23, по-прежнему находит широкое распространение. Кроме того, многое из того, что пояснялось в главе 23, применимо и к TPL. Поэтому усвоение материала главы 23 все еще необходимо для полного овладения особенностями организации многопоточной обработки на C#.

Два подхода к параллельному программированию

Применяя TPL, параллелизм в программу можно ввести двумя основными способами. Первый из них называется *параллелизмом данных*. При таком подходе одна операция над совокупностью данных разбивается на два параллельно выполняемых потока или больше, в каждом из которых обрабатывается часть данных. Так, если изменяется каждый элемент массива, то, применяя параллелизм данных, можно организовать параллельную обработку разных областей массива в двух или больше потоках. Нетрудно догадаться, что такие параллельно выполняющиеся действия могут привести к значительному ускорению обработки данных по сравнению с последовательным подходом. Несмотря на то что параллелизм данных был всегда возможен и с помощью класса `Thread`, построение масштабируемых решений средствами этого класса требовало немало усилий и времени. Это положение изменилось с появлением библиотеки TPL, с помощью которой масштабируемый параллелизм данных без особого труда вводится в программу.

Второй способ ввода параллелизм называется *параллелизмом задач*. При таком подходе две операции или больше выполняются параллельно. Следовательно, параллелизм задач представляет собой разновидность параллелизма, который достигался в прошлом средствами класса `Thread`. А к преимуществам, которые сулит применение TPL, относится простота применения и возможность автоматически масштабировать исполнение кода на несколько процессоров.

Класс Task

В основу TPL положен класс Task. Элементарная единица исполнения инкапсулируется в TPL средствами класса Task, а не Thread. Класс Task отличается от класса Thread тем, что он является абстракцией, представляющей асинхронную операцию. А в классе Thread инкапсулируется поток исполнения. Разумеется, на системном уровне поток по-прежнему остается элементарной единицей исполнения, которую можно планировать средствами операционной системы. Но соответствие экземпляра объекта класса Task и потока исполнения не обязательно оказывается взаимно-однозначным. Кроме того, исполнением задач управляет планировщик задач, который работает с пулом потоков. Это, например, означает, что несколько задач могут разделять один и тот же поток. Класс Task (и вся остальная библиотека TPL) определены в пространстве имен System.Threading.Tasks.

Создание задачи

Создать новую задачу в виде объекта класса Task и начать ее исполнение можно самыми разными способами. Для начала создадим объект типа Task с помощью конструктора и запустим его, вызвав метод Start(). Для этой цели в классе Task определено несколько конструкторов. Ниже приведен тот конструктор, которым мы собираемся воспользоваться:

```
public Task(Action действие)
```

где *действие* обозначает точку входа в код, представляющий задачу, тогда как Action — делегат, определенный в пространстве имен System. Форма делегата Action, которой мы собираемся воспользоваться, выглядит следующим образом.

```
public delegate void Action()
```

Таким образом, точкой входа должен служить метод, не принимающий никаких параметров и не возвращающий никаких значений. (Как будет показано далее, делегату Action можно также передать аргумент.)

Как только задача будет создана, ее можно запустить на исполнение, вызвав метод Start(). Ниже приведена одна из его форм.

```
public void Start()
```

После вызова метода Start() планировщик задач запланирует исполнение задачи.

В приведенной ниже программе все изложенное выше демонстрируется на практике. В этой программе отдельная задача создается на основе метода MyTask(). После того как начнет выполняться метод Main(), задача фактически создается и запускается на исполнение. Оба метода MyTask() и Main() выполняются параллельно.

```
// Создать и запустить задачу на исполнение.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoTask {

  // Метод выполняемый в качестве задачи.
```

```
  static void MyTask() {
    Console.WriteLine("MyTask() запущен");

    for(int count = 0; count < 10; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В методе MyTask(), подсчет равен " + count);
    }

    Console.WriteLine("MyTask завершен");
  }

  static void Main() {
    Console.WriteLine("Основной поток запущен.");

    // Сконструировать объект задачи.
    Task tsk = new Task(MyTask);

    // Запустить задачу на исполнение.
    tsk.Start();

    // метод Main() активным до завершения метода MyTask().
    for(int i = 0; i < 60; i++) {
      Console.Write(".");
      Thread.Sleep(100);
    }

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен результат выполнения этой программы. (У вас он может несколько отличаться в зависимости от загрузки задач, операционной системы и прочих факторов.)

```
Основной поток запущен.
.MyTask() запущен
.....В методе MyTask(), подсчет равен 0
.....В методе MyTask(), подсчет равен 1
.....В методе MyTask(), подсчет равен 2
.....В методе MyTask(), подсчет равен 3
.....В методе MyTask(), подсчет равен 4
.....В методе MyTask(), подсчет равен 5
.....В методе MyTask(), подсчет равен 6
.....В методе MyTask(), подсчет равен 7
.....В методе MyTask(), подсчет равен 8
.....В методе MyTask(), подсчет равен 9
MyTask завершен
.........Основной поток завершен.
```

Следует иметь в виду, что по умолчанию задача исполняется в фоновом потоке. Следовательно, при завершении создающего потока завершается и сама задача. Именно поэтому в рассматриваемой здесь программе метод `Thread.Sleep()` использован для сохранения активным основного потока до тех пор, пока не завершится выполнение метода `MyTask()`. Как и следовало ожидать, организовать ожидание завершения задачи можно и более совершенными способами, что и будет показано далее.

В приведенном выше примере программы задача, предназначавшаяся для параллельного исполнения, обозначалась в виде статического метода. Но такое требование к задаче не является обязательным. Например, в приведенной ниже программе, которая является переработанным вариантом предыдущей, метод `MyTask()`, выполняющий роль задачи, инкапсулирован внутри класса.

```
// Использовать метод экземпляра в качестве задачи.

using System;
using System.Threading;
using System.Threading.Tasks;

class MyClass {
  // Метод выполняемый в качестве задачи.
  public void MyTask() {
    Console.WriteLine("MyTask() запущен");

    for(int count = 0; count < 10; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В методе MyTask(), подсчет равен " + count);
    }

    Console.WriteLine("MyTask завершен ");
  }
}

class DemoTask {

static void Main() {
    Console.WriteLine("Основной поток запущен.");

    // Сконструировать объект типа MyClass.
    MyClass mc = new MyClass();

    // Сконструировать объект задачи для метода mc.MyTask().
    Task tsk = new Task(mc.MyTask);

    // Запустить задачу на исполнение.
    tsk.Start();

    // Сохранить метод Main() активным до завершения метода MyTask().
    for(int i = 0; i < 60; i++) {
      Console.Write(".");
      Thread.Sleep(100);
    }

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Результат выполнения этой программы получается таким же, как и прежде. Единственное отличие состоит в том, что метод `MyTask()` вызывается теперь для экземпляра объекта класса `MyClass`.

В отношении задач необходимо также иметь в виду следующее: после того, как задача завершена, она не может быть перезапущена. Следовательно, иного способа повторного запуска задачи на исполнение, кроме создания ее снова, не существует.

Применение идентификатора задачи

В отличие от класса `Thread`, в классе `Task` отсутствует свойство `Name` для хранения имени задачи. Но вместо этого в нем имеется свойство `Id` для хранения идентификатора задачи, по которому можно распознавать задачи. Свойство `Id` доступно только для чтения и относится к типу `int`. Оно объявляется следующим образом.

```
public int Id { get; }
```

Каждая задача получает идентификатор, когда она создается. Значения идентификаторов уникальны, но не упорядочены. Поэтому один идентификатор задачи может появиться перед другим, хотя он может и не иметь меньшее значение.

Идентификатор исполняемой в настоящий момент задачи можно выявить с помощью свойства `CurrentId`. Это свойство доступно только для чтения, относится к типу `static` и объявляется следующим образом.

```
public static Nullable<int> CurrentID { get; }
```

Оно возвращает исполняемую в настоящий момент задачу или же пустое значение, если вызывающий код не является задачей.

В приведенном ниже примере программы создаются две задачи и показывается, какая из них исполняется.

```
// Продемонстрировать применение свойств Id и CurrentId.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoTask {

  // Метод, исполняемый как задача.
  static void MyTask() {
    Console.WriteLine("MyTask() №" + Task.CurrentId + " запущен");

    for(int count = 0; count < 10; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В методе MyTask() #" + Task.CurrentId +
                        ", подсчет равен " + count );
    }

    Console.WriteLine("MyTask №" + Task.CurrentId + " завершен");
  }

  static void Main() {

   Console.WriteLine("Основной поток запущен.");

    // Сконструировать объекты двух задач.
    Task tsk = new Task(MyTask);
    Task tsk2 = new Task(MyTask);
```

```
    // Запустить задачи на исполнение.
    tsk.Start();
    tsk2.Start();

    Console.WriteLine("Идентификатор задачи tsk: " + tsk.Id);
    Console.WriteLine("Идентификатор задачи tsk2: " + tsk2.Id);

    // Сохранить метод Main() активным до завершения остальных задач.
    for(int i = 0; i < 60; i++) {
      Console.Write(".");
      Thread.Sleep(100);
    }

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Основной поток запущен
Идентификатор задачи tsk: 1
Идентификатор задачи tsk2: 2
.MyTask() №1 запущен
MyTask() №2 запущен
.....В методе MyTask() №1, подсчет равен 0
В методе MyTask() №2, подсчет равен 0
.....В методе MyTask() №2, подсчет равен 1
В методе MyTask() №1, подсчет равен 1
.....В методе MyTask() №1, подсчет равен 2
В методе MyTask() №2, подсчет равен 2
.....В методе MyTask() №2, подсчет равен 3
В методе MyTask() №1, подсчет равен 3
.....В методе MyTask() №1, подсчет равен 4
В методе MyTask() №2, подсчет равен 4
.....В методе MyTask() №1, подсчет равен 5
В методе MyTask() №2, подсчет равен 5
.....В методе MyTask() №2, подсчет равен 6
В методе MyTask() №1, подсчет равен 6
.....В методе MyTask() №2, подсчет равен 7
В методе MyTask() №1, подсчет равен 7
.....В методе MyTask() №1, подсчет равен 8
В методе MyTask() №2, подсчет равен 8
.....В методе MyTask() №1, подсчет равен 9
MyTask №1 завершен
В методе MyTask() №2, подсчет равен 9
MyTask №2 завершен
.........Основной поток завершен.
```

Применение методов ожидания

В приведенных выше примерах основной поток исполнения, а по существу, метод `Main()`, завершался потому, что такой результат гарантировали вызовы метода `Thread.Sleep()`. Но подобный подход нельзя считать удовлетворительным.

Организовать ожидание завершения задач можно и более совершенным способом, применяя методы ожидания, специально предоставляемые в классе `Task`. Самым простым из них считается метод `Wait()`, приостанавливающий исполнение вызывающего потока до тех пор, пока не завершится вызываемая задача. Ниже приведена простейшая форма объявления этого метода.

```
public void Wait()
```

При выполнении этого метода могут быть сгенерированы два исключения. Первым из них является исключение `ObjectDisposedException`. Оно генерируется в том случае, если задача освобождена посредством вызова метода `Dispose()`. А второе исключение, `AggregateException`, генерируется в том случае, если задача сама генерирует исключение или же отменяется. Как правило, отслеживается и обрабатывается именно это исключение. В связи с тем что задача может сгенерировать не одно исключение, если, например, у нее имеются порожденные задачи, все подобные исключения собираются в единое исключение типа `AggregateException`. Для того чтобы выяснить, что же произошло на самом деле, достаточно проанализировать внутренние исключения, связанные с этим совокупным исключением. А до тех пор в приведенных далее примерах любые исключения, генерируемые задачами, будут обрабатываться во время выполнения.

Ниже приведен вариант предыдущей программы, измененный с целью продемонстрировать применение метода `Wait()` на практике. Этот метод используется внутри метода `Main()`, чтобы приостановить его выполнение до тех пор, пока не завершатся обе задачи `tsk` и `tsk2`.

```
// Применить метод Wait().

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoTask {

  // Метод, исполняемый как задача.
  static void MyTask() {

    Console.WriteLine("MyTask() №" + Task.CurrentId + " запущен");

      for(int count = 0; count < 10; count++) {
        Thread.Sleep(500);
        Console.WriteLine("В методе MyTask() #" + Task.CurrentId +
                          ", подсчет равен " + count );
    }

    Console.WriteLine("MyTask №" + Task.CurrentId + " завершен");
  }

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Сконструировать объекты двух задач.
    Task tsk = new Task(MyTask);
    Task tsk2 = new Task(MyTask);
```

```
    // Запустить задачи на исполнение.
    tsk.Start();
    tsk2.Start();

    Console.WriteLine("Идентификатор задачи tsk: " + tsk.Id);
    Console.WriteLine("Идентификатор задачи tsk2: " + tsk2.Id);

    // Приостановить выполнение метода Main() до тех пор,
    // пока не завершатся обе задачи tsk и tsk2
    tsk.Wait();
    tsk2.Wait();

    Console.WriteLine("Основной поток завершен.");
  }
}
```

При выполнении этой программы получается следующий результат.

```
Основной поток запущен
Идентификатор задачи tsk: 1
Идентификатор задачи tsk2: 2
MyTask() №1 запущен
MyTask() №2 запущен
В методе MyTask() №1, подсчет равен 0
В методе MyTask() №2, подсчет равен 0
В методе MyTask() №1, подсчет равен 1
В методе MyTask() №2, подсчет равен 1
В методе MyTask() №1, подсчет равен 2
В методе MyTask() №2, подсчет равен 2
В методе MyTask() №1, подсчет равен 3
В методе MyTask() №2, подсчет равен 3
В методе MyTask() №1, подсчет равен 4
В методе MyTask() №2, подсчет равен 4
В методе MyTask() №1, подсчет равен 5
В методе MyTask() №2, подсчет равен 5
В методе MyTask() №1, подсчет равен 6
В методе MyTask() №2, подсчет равен 6
В методе MyTask() №1, подсчет равен 7
В методе MyTask() №2, подсчет равен 7
В методе MyTask() №1, подсчет равен 8
В методе MyTask() №2, подсчет равен 8
В методе MyTask() №1, подсчет равен 9
MyTask №1 завершен
В методе MyTask() №2, подсчет равен 9
MyTask №2 завершен
Основной поток завершен.
```

Как следует из приведенного выше результата, выполнение метода `Main()` приостанавливается до тех пор, пока не завершатся обе задачи `tsk` и `tsk2`. Следует, однако, иметь в виду, что в рассматриваемой здесь программе последовательность завершения задач `tsk` и `tsk2` не имеет особого значения для вызовов метода `Wait()`. Так, если первой завершается задача `tsk2`, то в вызове метода `tsk.Wait()` будет по-прежнему ожидаться завершение задачи `tsk`. В таком случае вызов метода `tsk2.Wait()` приведет к выполнению и немедленному возврату из него, поскольку задача `tsk2` уже завершена.

В данном случае оказывается достаточно двух вызовов метода `Wait()`, но того же результата можно добиться и более простым способом, воспользовавшись методом `WaitAll()`. Этот метод организует ожидание завершения группы задач. Возврата из него не произойдет до тех пор, пока не завершатся все задачи. Ниже приведена простейшая форма объявления этого метода.

```
public static void WaitAll(params Task[] tasks)
```

Задачи, завершения которых требуется ожидать, передаются с помощью параметра в виде массива *tasks*. А поскольку этот параметр относится к типу `params`, то данному методу можно отдельно передать массив объектов типа `Task` или список задач. При этом могут быть сгенерированы различные исключения, включая и `AggregateException`.

Для того чтобы посмотреть, как метод `WaitAll()` действует на практике, замените в приведенной выше программе следующую последовательность вызовов.

```
tsk.Wait();
tsk2.Wait();
```

на

```
Task.WaitAll(tsk, tsk2);
```

Программа будет работать точно так же, но логика ее выполнения станет более понятной.

Организуя ожидание завершения нескольких задач, следует быть особенно внимательным, чтобы избежать взаимоблокировок. Так, если две задачи ожидают завершения друг друга, то вызов метода `WaitAll()` вообще не приведет к возврату из него. Разумеется, условия для взаимоблокировок возникают в результате ошибок программирования, которых следует избегать. Следовательно, если вызов метода `WaitAll()` не приводит к возврату из него, то следует внимательно проанализировать, могут ли две задачи или больше взаимно блокироваться. (Вызов метода `Wait()`, который не приводит к возврату из него, также может стать причиной взаимоблокировок.)

Иногда требуется организовать ожидание до тех пор, пока не завершится любая из группы задач. Для этой цели служит метод `WaitAny()`. Ниже приведена простейшая форма его объявления.

```
public static int WaitAny(params Task[] tasks)
```

Задачи, завершения которых требуется ожидать, передаются с помощью параметра в виде массива *tasks* объектов типа `Task` или отдельного списка аргументов типа `Task`. Этот метод возвращает индекс задачи, которая завершается первой. При этом могут быть сгенерированы различные исключения.

Попробуйте применить метод `WaitAny()` на практике, подставив в предыдущей программе следующий вызов.

```
Task.WaitAny(tsk, tsk2);
```

Теперь, выполнение метода `Main()` возобновится, а программа завершится, как только завершится одна из двух задач.

Помимо рассматривавшихся здесь форм методов `Wait()`, `WaitAll()` и `WaitAny()`, имеются и другие их варианты, в которых можно указывать период простоя или отслеживать признак отмены. (Подробнее об отмене задач речь пойдет далее в этой главе.)

Вызов метода Dispose()

В классе Task реализуется интерфейс IDisposable, в котором определяется метод Dispose(). Ниже приведена форма его объявления.

```
public void Dispose()
```

Метод Dispose() реализуется в классе Task, освобождая ресурсы, используемые этим классом. Как правило, ресурсы, связанные с классом Task, освобождаются автоматически во время "сборки мусора" (или по завершении программы). Но если эти ресурсы требуется освободить еще раньше, то для этой цели служит метод Dispose(). Это особенно важно в тех программах, где создается большое число задач, оставляемых на произвол судьбы.

Следует, однако, иметь в виду, что метод Dispose() можно вызывать для отдельной задачи только после ее завершения. Следовательно, для выяснения факта завершения отдельной задачи, прежде чем вызывать метод Dispose(), потребуется некоторый механизм, например, вызов метода Wait(). Именно поэтому так важно было рассмотреть метод Wait(), перед тем как обсуждать метод Dispose(). Ели же попытаться вызвать Dispose() для все еще активной задачи, то будет сгенерировано исключение InvalidOperationException.

Во всех примерах, приведенных в этой главе, создаются довольно короткие задачи, которые сразу же завершаются, и поэтому применение метода Dispose() в этих примерах не дает никаких преимуществ. (Именно по этой причине вызывать метод Dispose() в приведенных выше программах не было никакой необходимости. Ведь все они завершались, как только завершалась задача, что в конечном итоге приводило к освобождению от остальных задач.) Но в целях демонстрации возможностей данного метода и во избежание каких-либо недоразумений метод Dispose() будет вызываться явным образом при непосредственном обращении с экземплярами объектов типа Task во всех последующих примерах программ. Если вы обнаружите отсутствие вызовов метода Dispose() в исходном коде, полученном из других источников, то не удивляйтесь этому. Опять же, если программа завершается, как только завершится задача, то вызывать метод Dispose() нет никакого смысла — разве что в целях демонстрации его применения.

Применение класса TaskFactory для запуска задачи

Приведенные выше примеры программы были составлены не так эффективно, как следовало бы, поскольку задачу можно создать и сразу же начать ее исполнение, вызвав метод StartNew(), определенный в классе TaskFactory. В классе TaskFactory предоставляются различные методы, упрощающие создание задач и управление ими. По умолчанию объект класса TaskFactory может быть получен из свойства Factory, доступного только для чтения в классе Task. Используя это свойство, можно вызвать любые методы класса TaskFactory. Метод StartNew() существует во множестве форм. Ниже приведена самая простая форма его объявления:

```
public Task StartNew(Action action)
```

где *action* — точка входа в исполняемую задачу. Сначала в методе StartNew() автоматически создается экземпляр объекта типа Task для действия, определяемого параметром *action*, а затем планируется запуск задачи на исполнение. Следовательно, необходимость в вызове метода Start() теперь отпадает.

Например, следующий вызов метода StartNew() в рассматривавшихся ранее программах приведет к созданию и запуску задачи tsk одним действием.

```
Task tsk = Task.Factory.StartNew(MyTask);
```

После этого оператора сразу же начнет выполняться метод MyTask().

Метод StartNew() оказывается более эффективным в тех случаях, когда задача создается и сразу же запускается на исполнение. Поэтому именно такой подход и применяется в последующих примерах программ.

Применение лямбда-выражения в качестве задачи

Кроме использования обычного метода в качестве задачи, существует и другой, более рациональный подход: указать лямбда-выражение как отдельно решаемую задачу. Напомним, что лямбда-выражения являются особой формой анонимных функций. Поэтому они могут исполняться как отдельные задачи. Лямбда-выражения оказываются особенно полезными в тех случаях, когда единственным назначением метода является решение одноразовой задачи. Лямбда-выражения могут составлять отдельную задачу или же вызывать другие методы. Так или иначе, применение лямбда-выражения в качестве задачи может стать привлекательной альтернативой именованному методу.

В приведенном ниже примере программы демонстрируется применение лямбда-выражения в качестве задачи. В этой программе код метода MyTask() из предыдущих примеров программ преобразуется в лямбда-выражение.

```
// Применить лямбда-выражение в качестве задачи.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoLambdaTask {
  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Далее лямбда-выражение используется для определения задачи.
    Task tsk = Task.Factory.StartNew( () => {
      Console.WriteLine("Задача запущена");

      for(int count = 0; count < 10; count++) {
        Thread.Sleep(500);
        Console.WriteLine("Подсчет в задаче равен " + count );
      }

      Console.WriteLine("Задача завершена");
    } );

    // Ожидать завершения задачи tsk.
    tsk.Wait();

    // Освободить задачу tsk.
    tsk.Dispose();
```

```
    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Основной поток запущен.
Задача запущена
Подсчет в задаче равен 0
Подсчет в задаче равен 1
Подсчет в задаче равен 2
Подсчет в задаче равен 3
Подсчет в задаче равен 4
Подсчет в задаче равен 5
Подсчет в задаче равен 6
Подсчет в задаче равен 7
Подсчет в задаче равен 8
Подсчет в задаче равен 9
Задача завершена
Основной поток завершен.
```

Помимо применения лямбда-выражения для описания задачи, обратите также внимание в данной программе на то, что вызов метода `tsk.Dispose()` не делается до тех пор, пока не произойдет возврат из метода `tsk.Wait()`. Как пояснялось в предыдущем разделе, метод `Dispose()` можно вызывать только по завершении задачи. Для того чтобы убедиться в этом, попробуйте поставить вызов метода `tsk.Dispose()` в рассматриваемой здесь программе перед вызовом метода `tsk.Wait()`. Вы сразу же заметите, что это приведет к исключительной ситуации.

Создание продолжения задачи

Одной из новаторских и очень удобных особенностей библиотеки TPL является возможность создавать продолжение задачи. *Продолжение* — это одна задача, которая автоматически начинается после завершения другой задачи. Создать продолжение можно, в частности, с помощью метода `ContinueWith()`, определенного в классе `Task`. Ниже приведена простейшая форма его объявления:

```
public Task ContinueWith(Action<Task> действие_продолжения)
```

где *действие_продолжения* обозначает задачу, которая будет запущена на исполнение по завершении вызывающей задачи. У делегата `Action` имеется единственный параметр типа `Task`. Следовательно, вариант делегата `Action`, применяемого в данном методе, выглядит следующим образом.

```
public delegate void Action<in T>(T obj)
```

В данном случае обобщенный параметр `T` обозначает класс `Task`.

Продолжение задачи демонстрируется на примере следующей программы.

```
// Продемонстрировать продолжение задачи.

using System;
using System.Threading;
```

```
using System.Threading.Tasks;

class ContinuationDemo {

  // Метод, исполняемый как задача.
  static void MyTask() {
    Console.WriteLine("MyTask() запущен");

    for(int count = 0; count < 5; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В методе MyTask() подсчет равен " + count );
    }

    Console.WriteLine("MyTask завершен");
  }

  // Метод, исполняемый как продолжение задачи.
  static void ContTask(Task t) {
    Console.WriteLine("Продолжение запущено");

    for(int count = 0; count < 5; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В продолжении подсчет равен " + count );
    }
    Console.WriteLine("Продолжение завершено");
  }

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Сконструировать объект первой задачи.
    Task tsk = new Task(MyTask);

    // А теперь создать продолжение задачи.
    Task taskCont = tsk.ContinueWith(ContTask);

    // Начать последовательность задач.
    tsk.Start();

    // Ожидать завершения продолжения.
    taskCont.Wait();

    tsk.Dispose();
    taskCont.Dispose();

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен результата выполнения данной программы.

```
Основной поток запущен.
MyTask() запущен
В методе MyTask() подсчет равен 0
```

```
В методе MyTask() подсчет равен 1
В методе MyTask() подсчет равен 2
В методе MyTask() подсчет равен 3
В методе MyTask() подсчет равен 4
MyTask() завершен
Продолжение запущено
В продолжении подсчет равен 0
В продолжении подсчет равен 1
В продолжении подсчет равен 2
В продолжении подсчет равен 3
В продолжении подсчет равен 4
Продолжение завершено
Основной поток завершен.
```

Как следует из приведенного выше результата, вторая задача не начинается до тех пор, пока не завершится первая. Обратите также внимание на то, что в методе `Main()` пришлось ожидать окончания только продолжения задачи. Дело в том, что метод `MyTask()` как задача завершается еще до начала метода `ContTask` как продолжения задачи. Следовательно, ожидать завершения метода `MyTask()` нет никакой надобности, хотя если и организовать такое ожидание, то в этом будет ничего плохого.

Любопытно, что в качестве продолжения задачи нередко применяется лямбда-выражение. Для примера ниже приведен еще один способ организации продолжения задачи из предыдущего примера программы.

```
// В данном случае в качестве продолжения задачи применяется лямбда-выражение.
Task taskCont = tsk.ContinueWith((first) =>
        {
          Console.WriteLine("Продолжение запущено");
          for(int count = 0; count < 5; count++) {
          Thread.Sleep(500);
          Console.WriteLine("В продолжении подсчет равен " + count );
        }
        Console.WriteLine("Продолжение завершено");
  }
);
```

В этом фрагменте кода параметр `first` принимает предыдущую задачу (в данном случае — `tsk`).

Помимо метода `ContinueWith()`, в классе `Task` предоставляются и другие методы, поддерживающие продолжение задачи, обеспечиваемое классом `TaskFactory`. К их числу относятся различные формы методов `ContinueWhenAny()` и `ContinueWhenAll()`, которые продолжают задачу, если завершится любая или все указанные задачи соответственно.

Возврат значения из задачи

Задача может возвращать значение. Это очень удобно по двум причинам. Во-первых, это означает, что с помощью задачи можно вычислить некоторый результат. Подобным образом поддерживаются параллельные вычисления. И во-вторых, вызывающий процесс окажется блокированным до тех пор, пока не будет получен результат. Это означает, что для организации ожидания результата не требуется никакой особой синхронизации.

Для того чтобы возвратить результат из задачи, достаточно создать эту задачу, используя обобщенную форму `Task<TResult>` класса `Task`. Ниже приведены два конструктора этой формы класса `Task`:

```
public Task(Func<TResult> функция)
public Task(Func<Object, TResult> функция, Object состояние)
```

где *функция* обозначает выполняемый делегат. Обратите внимание на то, что он должен быть типа `Func`, а не `Action`. Тип `Func` используется именно в тех случаях, когда задача возвращает результат. В первом конструкторе создается задача без аргументов, а во втором конструкторе — задача, принимающая аргумент типа `Object`, передаваемый как *состояние*. Имеются также другие конструкторы данного класса.

Как и следовало ожидать, имеются также другие варианты метода `StartNew()`, доступные в обобщенной форме класса `TaskFactory<TResult>` и поддерживающие возврат результата из задачи. Ниже приведены те варианты данного метода, которые применяются параллельно с только что рассмотренными конструкторами класса `Task`.

```
public Task<TResult> StartNew(Func<TResult> функция)
public Task<TResult> StartNew(Func<Object,TResult> функция, Object состояние)
```

В любом случае значение, возвращаемое задачей, получается из свойства `Result` в классе `Task`, которое определяется следующим образом.

```
public TResult Result { get; internal set; }
```

Аксессор `set` является внутренним для данного свойства, и поэтому оно оказывается доступным во внешнем коде, по существу, только для чтения. Следовательно, задача получения результата блокирует вызывающий код до тех пор, пока результат не будет вычислен.

В приведенном ниже примере программы демонстрируется возврат задачей значений. В этой программе создаются два метода. Первый из них, `MyTask()`, не принимает параметров, а просто возвращает логическое значение `true` типа `bool`. Второй метод, `SumIt()`, принимает единственный параметр, который приводится к типу `int`, и возвращает сумму из значения, передаваемого в качестве этого параметра.

```
// Возвратить значение из задачи.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoTask {
  // Простейший метод, возвращающий результат и не принимающий аргументов.
  static bool MyTask() {
    return true;
  }

  // Этот метод возвращает сумму из положительного целого значения,
  // которое ему передается в качестве единственного параметра
  static int SumIt(object v) {
    int x = (int) v;
    int sum = 0;
```

```
    for(; x > 0; x--)
      sum += x;

    return sum;
  }

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Сконструировать объект первой задачи.
    Task<bool> tsk = Task<bool>.Factory.StartNew(MyTask);

    Console.WriteLine("Результат после выполнения задачи MyTask: " +
                      tsk.Result);

    // Сконструировать объект второй задачи.
    Task<int> tsk2 = Task<int>.Factory.StartNew(SumIt, 3);

    Console.WriteLine("Результат после выполнения задачи SumIt: " +
                      tsk2.Result);

    tsk.Dispose();
    tsk2.Dispose();

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
Основной поток запущен.
Результат после выполнения задачи MyTask: True
Результат после выполнения SumIt: 6
Основной поток завершен.
```

Помимо упомянутых выше форм класса `Task<TResult>` и метода `StartNew<TResult>`, имеются также другие формы. Они позволяют указывать другие дополнительные параметры.

Отмена задачи и обработка исключения `AggregateException`

В версии 4.0 среды .NET Framework внедрена новая подсистема, обеспечивающая структурированный, хотя и очень удобный способ отмены задачи. Эта новая подсистема основывается на понятии *признака отмены*. Признаки отмены поддерживаются в классе `Task`, среди прочего, с помощью фабричного метода `StartNew()`.

ПРИМЕЧАНИЕ

Новую подсистему отмены можно применять и для отмены потоков, рассматривавшихся в предыдущей главе, но она полностью интегрирована в TPL и PLINQ. Именно поэтому эта подсистема рассматривается в этой главе.

Отмена задачи, как правило, выполняется следующим образом. Сначала получается признак отмены из источника признаков отмены. Затем этот признак передается задаче, после чего она должна контролировать его на предмет получения запроса на отмену. (Этот запрос может поступить только из источника признаков отмены.) Если получен запрос на отмену, задача должна завершиться. В одних случаях этого оказывается достаточно для простого прекращения задачи без каких-либо дополнительных действий, а в других — из задачи должен быть вызван метод `ThrowIfCancellationRequested()` для признака отмены. Благодаря этому в отменяющем коде становится известно, что задача отменена. А теперь рассмотрим процесс отмены задачи более подробно.

Признак отмены является экземпляром объекта типа `CancellationToken`, т.е. структуры, определенной в пространстве имен `System.Threading`. В структуре `CancellationToken` определено несколько свойств и методов, но мы воспользуемся двумя из них. Во-первых, это доступное только для чтения свойство `IsCancellationRequested`, которое объявляется следующим образом.

```
public bool IsCancellationRequested { get; }
```

Оно возвращает логическое значение `true`, если отмена задачи была запрошена для вызывающего признака, а иначе — логическое значение `false`. И во-вторых, это метод `ThrowIfCancellationRequested()`, который объявляется следующим образом.

```
public void ThrowIfCancellationRequested()
```

Если признак отмены, для которого вызывается этот метод, получил запрос на отмену, то в данном методе генерируется исключение `OperationCanceledException`. В противном случае никаких действий не выполняется. В отменяющем коде можно организовать отслеживание упомянутого исключения с целью убедиться в том, что отмена задачи действительно произошла. Как правило, с этой целью сначала перехватывается исключение `AggregateException`, а затем его внутреннее исключение анализируется с помощью свойства `InnerException` или `InnerExceptions`. (Свойство `InnerExceptions` представляет собой коллекцию исключений. Подробнее о коллекциях речь пойдет в главе 25.)

Признак отмены получается из источника признаков отмены, который представляет собой объект класса `CancellationTokenSource`, определенного в пространстве имен `System.Threading`. Для того чтобы получить данный признак, нужно создать сначала экземпляр объекта типа `CancellationTokenSource`. (С этой целью можно воспользоваться вызываемым по умолчанию конструктором класса `CancellationTokenSource`.) Признак отмены, связанный с данным источником, оказывается доступным через используемое только для чтения свойство `Token`, которое объявляется следующим образом.

```
public CancellationToken Token { get; }
```

Это и есть тот признак, который должен быть передан отменяемой задаче.

Для отмены в задаче должна быть получена копия признака отмены и организован контроль этого признака с целью отслеживать саму отмену. Такое отслеживание можно организовать тремя способами: опросом, методом обратного вызова и с помощью дескриптора ожидания. Проще всего организовать опрос, и поэтому здесь будет рассмотрен именно этот способ. С целью опроса в задаче проверяется упомянутое выше свойство `IsCancellationRequested` признака отмены. Если это свойство содержит логическое значение `true`, значит, отмена была запрошена, и задача долж-

на быть завершена. Опрос может оказаться весьма эффективным, если организовать его правильно. Так, если задача содержит вложенные циклы, то проверка свойства `IsCancellationRequested` во внешнем цикле зачастую дает лучший результат, чем его проверка на каждом шаге внутреннего цикла.

Для создания задачи, из которой вызывается метод `ThrowIfCancellationRequested()`, когда она отменяется, обычно требуется передать признак отмены как самой задаче, так и конструктору класса `Task`, будь то непосредственно или же косвенно через метод `StartNew()`. Передача признака отмены самой задаче позволяет изменить состояние отменяемой задачи в запросе на отмену из внешнего кода. Далее будет использована следующая форма метода `StartNew()`.

```
public Task StartNew(Action<Object> action, Object состояние,
                     CancellationToken признак_отмены)
```

В этой форме признак отмены передается через параметры, обозначаемые как *состояние* и *признак_отмены*. Это означает, что признак отмены будет передан как делегату, реализующему задачу, так и самому экземпляру объекта типа `Task`. Ниже приведена форма, поддерживающая делегат `Action`.

```
public delegate void Action<in T>(T obj)
```

В данном случае обобщенный параметр `T` обозначает тип `Object`. В силу этого объект *obj* должен быть приведен внутри задачи к типу `CancellationToken`.

И еще одно замечание: по завершении работы с источником признаков отмены следует освободить его ресурсы, вызвав метод `Dispose()`.

Факт отмены задачи может быть проверен самыми разными способами. Здесь применяется следующий подход: проверка значения свойства `IsCanceled` для экземпляра объекта типа `Task`. Если это логическое значение `true`, то задача была отменена.

В приведенной ниже программе демонстрируется отмена задачи. В ней применяется опрос для контроля состояния признака отмены. Обратите внимание на то, что метод `ThrowIfCancellationRequested()` вызывается после входа в метод `MyTask()`. Это дает возможность завершить задачу, если она была отмена еще до ее запуска. Внутри цикла проверяется свойство `IsCancellationRequested`. Если это свойство содержит логическое значение `true`, а оно устанавливается после вызова метода `Cancel()` для экземпляра источника признаков отмены, то на экран выводится сообщение об отмене и далее вызывается метод `ThrowIfCancellationRequested()` для отмены задачи.

```
// Простой пример отмены задачи с использованием опроса.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoCancelTask {

  // Метод, исполняемый как задача.
  static void MyTask(Object ct) {
    CancellationToken cancelTok = (CancellationToken) ct;

    // Проверить, отменена ли задача, прежде чем запускать ее.
    cancelTok.ThrowIfCancellationRequested();
```

```
    Console.WriteLine("MyTask() запущен");

    for(int count = 0; count < 10; count++) {
      // В данном примере для отслеживания отмены задачи применяется опрос.
      if(cancelTok.IsCancellationRequested) {
        Console.WriteLine("Получен запрос на отмену задачи.");
        cancelTok.ThrowIfCancellationRequested();
      }

      Thread.Sleep(500);
      Console.WriteLine("В методе MyTask() подсчет равен " + count );
    }
      Console.WriteLine("MyTask завершен");
  }

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Создать объект источника признаков отмены.
    CancellationTokenSource cancelTokSrc = new CancellationTokenSource();

    // Запустить задачу, передав признак отмены ей самой и делегату.
    Task tsk = Task.Factory.StartNew(MyTask, cancelTokSrc.Token,
                                     cancelTokSrc.Token);

    // Дать задаче возможность исполняться вплоть до ее отмены.
    Thread.Sleep(2000);
    try {
      // Отменить задачу.
      cancelTokSrc.Cancel();

      // Приостановить выполнение метода Main() до тех пор,
      // пока не завершится задача tsk.
      tsk.Wait();
    } catch (AggregateException exc) {
      if(tsk.IsCanceled)
        Console.WriteLine("\nЗадача tsk отменена\n");

    // Для просмотра исключения снять комментарии со следующей строки кода:
    // Console.WriteLine(exc);
    } finally {
      tsk.Dispose();
      cancelTokSrc.Dispose();
    }

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Ниже приведен результат выполнения этой программы. Обратите внимание на то что задача отменяется через 2 секунды.

```
Основной поток запущен.
MyTask() запущен
```

```
В методе MyTask() подсчет равен 0
В методе MyTask() подсчет равен 1
В методе MyTask() подсчет равен 2
В методе MyTask() подсчет равен 3
Получен запрос на отмену задачи.

Задача tsk отменена

Основной поток завершен.
```

Как следует из приведенного выше результата, выполнение метода `MyTask()` отменяется в методе `Main()` лишь две секунды спустя. Следовательно, в методе `MyTask()` выполняются четыре шага цикла. Когда же перехватывается исключение `AggregateException`, проверяется состояние задачи. Если задача `tsk` отменена, что и должно произойти в данном примере, то об этом выводится соответствующее сообщение. Следует, однако, иметь в виду, что когда сообщение `AggregateException` генерируется в ответ на отмену задачи, то это еще не свидетельствует об ошибке, а просто означает, что задача была отменена.

Выше были изложены лишь самые основные принципы, положенные в основу отмены задачи и генерирования исключения `AggregateException`. Тем не менее эта тема намного обширнее и требует от вас самостоятельного и углубленного изучения, если вы действительно хотите создавать высокопроизводительные, масштабируемые приложения.

Другие средства организации задач

В предыдущих разделах был описан ряд понятий и основных способов организации и исполнения задач. Но имеются и другие полезные средства. В частности, задачи можно делать вложенными, когда одни задачи способны создавать другие, или же порожденными, когда вложенные задачи оказываются тесно связанными с создающей их задачей.

В предыдущем разделе было дано краткое описание исключения `AggregateException`, но у него имеются также другие особенности, которые могут оказаться весьма полезными. К их числу относится метод `Flatten()`, применяемый для преобразования любых внутренних исключений типа `AggregateException` в единственное исключение `AggregateException`. Другой метод, `Handle()`, служит для обработки исключения, составляющего совокупное исключение `AggregateException`.

При создании задачи имеется возможность указать различные дополнительные параметры, оказывающие влияние на особенности ее исполнения. Для этой цели указывается экземпляр объекта типа `TaskCreationOptions` в конструкторе класса `Task` или же в фабричном методе `StartNew()`. Кроме того, в классе `TaskFactory` доступно целое семейство методов `FromAsync()`, поддерживающих модель асинхронного программирования (APM — Asynchronous Programming Model).

Как упоминалось ранее в этой главе, задачи планируются на исполнение экземпляром объекта класса `TaskScheduler`. Как правило, для этой цели предоставляется планировщик, используемый по умолчанию в среде .NET Framework. Но этот планировщик может быть настроен под конкретные потребности разработчика. Кроме того, допускается применение специализированных планировщиков задач.

Класс Parallel

В примерах, приведенных до сих пор в этой главе, демонстрировались ситуации, в которых библиотека TPL использовалась таким же образом, как и класс Thread. Но это было лишь самое элементарное ее применение, поскольку в TPL имеются и другие средства. К их числу относится класс Parallel, который упрощает параллельное исполнение кода и предоставляет методы, рационализирующие оба вида параллелизма: данных и задач.

Класс Parallel является статическим, и в нем определены методы For(), ForEach() и Invoke(). У каждого из этих методов имеются различные формы. В частности, метод For() выполняет распараллеливаемый цикл for, а метод ForEach() — распараллеливаемый цикл foreach, и оба метода поддерживают параллелизм данных. А метод Invoke() поддерживает параллельное выполнение двух методов или больше. Как станет ясно дальше, эти методы дают преимущество реализации на практике распространенных методик параллельного программирования, не прибегая к управлению задачами или потоками явным образом. В последующих разделах каждый из этих методов будет рассмотрен более подробно.

Распараллеливание задач методом Invoke()

Метод Invoke(), определенный в классе Parallel, позволяет выполнять один или несколько методов, указываемых в виде его аргументов. Он также масштабирует исполнение кода, используя доступные процессоры, если имеется такая возможность. Ниже приведена простейшая форма его объявления.

```
public static void Invoke(params Action[] actions)
```

Выполняемые методы должны быть совместимы с описанным ранее делегатом Action. Напомним, что делегат Action объявляется следующим образом.

```
public delegate void Action()
```

Следовательно, каждый метод, передаваемый методу Invoke() в качестве аргумента, не должен ни принимать параметров, ни возвращать значение. Благодаря тому что параметр *actions* данного метода относится к типу params, выполняемые методы могут быть указаны в виде переменного списка аргументов. Для этой цели можно также воспользоваться массивом объектов типа Action, но зачастую оказывается проще указать список аргументов.

Метод Invoke() сначала инициирует выполнение, а затем ожидает завершения всех передаваемых ему методов. Это, в частности, избавляет от необходимости (да и не позволяет) вызывать метод Wait(). Все функции параллельного выполнения метод Wait() берет на себя. И хотя это не гарантирует, что методы будут действительно выполняться параллельно, тем не менее, именно такое их выполнение предполагается, если система поддерживает несколько процессоров. Кроме того, отсутствует возможность указать порядок выполнения методов от первого и до последнего, и этот порядок не может быть таким же, как и в списке аргументов.

В приведенном ниже примере программы демонстрируется применение метода Invoke() на практике. В этой программе два метода MyMeth() и MyMeth2() выполняются параллельно посредством вызова метода Invoke(). Обратите внимание на простоту организации данного процесса.

```
// Применить метод Parallel.Invoke() для параллельного выполнения двух методов.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoParallel {

  // Метод, исполняемый как задача.
  static void MyMeth() {
    Console.WriteLine("MyMeth запущен");

    for(int count = 0; count < 5; count++) {
      Thread.Sleep(500);
       Console.WriteLine("В методе MyMeth подсчет равен " + count );
    }

    Console.WriteLine("MyMeth завершен");
  }

  // Метод, исполняемый как задача.
  static void MyMeth2() {
    Console.WriteLine("MyMeth2 запущен");

    for(int count = 0; count < 5; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В методе MyMeth2, подсчет равен " + count );
    }

    Console.WriteLine("MyMeth2 завершен");
  }

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Выполнить параллельно два именованных метода.
    Parallel.Invoke(MyMeth, MyMeth2);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Выполнение этой программы может привести к следующему результату.

```
Основной поток запущен.
MyMeth() запущен
MyMeth2() запущен
В методе MyMeth() подсчет равен 0
В методе MyMeth2() подсчет равен 0
В методе MyMeth() подсчет равен 1
В методе MyMeth2() подсчет равен 1
В методе MyMeth() подсчет равен 2
В методе MyMeth2() подсчет равен 2
В методе MyMeth() подсчет равен 3
```

```
В методе MyMeth2() подсчет равен 3
В методе MyMeth() подсчет равен 4
MyMeth() завершен
В методе MyMeth2() подсчет равен 4
MyMeth2() завершен
Основной поток завершен.
```

В данном примере особое внимание обращает на себя следующее обстоятельство: выполнение метода `Main()` приостанавливается до тех пор, пока не произойдет возврат из метода `Invoke()`. Следовательно, метод `Main()`, в отличие от методов `MyMeth()` и `MyMeth2()`, не выполняется параллельно. Поэтому применять метод `Invoke()` показанным здесь способом нельзя в том случае, если требуется, чтобы исполнение вызывающего потока продолжалось.

В приведенном выше примере использовались именованные методы, но для вызова метода `Invoke()` это условие не является обязательным. Ниже приведен переделанный вариант той же самой программы, где в качестве аргументов в вызове метода `Invoke()` применяются лямбда-выражения.

```
// Применить метод Parallel.Invoke() для параллельного выполнения двух методов.
// В этой версии программы применяются лямбда-выражения.

using System;
using System.Threading;
using System.Threading.Tasks;

class DemoParallel {

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Выполнить два анонимных метода, указываемых в лямбда-выражениях.
    Parallel.Invoke( () => {
        Console.WriteLine("Выражение #1 запущено");

        for(int count = 0; count < 5; count++) {
          Thread.Sleep(500);
          Console.WriteLine("В выражении #1 подсчет равен " + count );
        }

        Console.WriteLine("Выражение #1 завершено");
      },

        () => {
          Console.WriteLine("Выражение #2 запущено");

          for(int count = 0; count < 5; count++) {
            Thread.Sleep(500);
            Console.WriteLine("В выражении #2 подсчет равен " + count );
          }

          Console.WriteLine("Выражение #2 завершено");
        }
    );
```

```
    Console.WriteLine("Основной поток завершен.");
  }
}
```

Эта программа дает результат, похожий на результат выполнения предыдущей программы.

Применение метода `For()`

В TPL параллелизм данных поддерживается, в частности, с помощью метода `For()`, определенного в классе `Parallel`. Этот метод существует в нескольких формах. Его рассмотрение мы начнем с самой простой формы, приведенной ниже:

```
public static ParallelLoopResult
              For(int fromInclusive, int toExclusive, Action<int> body)
```

где *fromInclusive* обозначает начальное значение того, что соответствует переменной управления циклом; оно называется также итерационным, или индексным, значением; а *toExclusive* — значение, на единицу больше конечного. На каждом шаге цикла переменная управления циклом увеличивается на единицу. Следовательно, цикл постепенно продвигается от начального значения *fromInclusive* к конечному значению *toExclusive* минус единица. Циклически выполняемый код указывается методом, передаваемым через параметр *body*. Этот метод должен быть совместим с делегатом `Action<int>`, объявляемым следующим образом.

```
public delegate void Action<in T>(T obj)
```

Для метода `For()` обобщенный параметр `T` должен быть, конечно, типа `int`. Значение, передаваемое через параметр *obj*, будет следующим значением переменной управления циклом. А метод, передаваемый через параметр *body*, может быть именованным или анонимным. Метод `For()` возвращает экземпляр объекта типа `ParallelLoopResult`, описывающий состояние завершения цикла. Для простых циклов этим значением можно пренебречь. (Более подробно это значение будет рассмотрено несколько ниже.)

Главная особенность метода `For()` состоит в том, что он позволяет, когда такая возможность имеется, распараллелить исполнение кода в цикле. А это, в свою очередь, может привести к повышению производительности. Например, процесс преобразования массива в цикле может быть разделен на части таким образом, чтобы разные части массива преобразовывались одновременно. Следует, однако, иметь в виду, что повышение производительности не гарантируется из-за отличий в количестве доступных процессоров в разных средах выполнения, а также из-за того, что распараллеливание мелких циклов может составить издержки, которые превышают сэкономленное время.

В приведенном ниже примере программы демонстрируется применение метода `For()` на практике. В начале этой программы создается массив `data`, состоящий из 1000000000 целых значений. Затем вызывается метод `For()`, которому в качестве "тела" цикла передается метод `MyTransform()`. Этот метод состоит из ряда операторов, выполняющих произвольные преобразования в массиве `data`. Его назначение — сымитировать конкретную операцию. Как будет подробнее пояснено несколько ниже, выполняемая операция должна быть нетривиальной, чтобы параллелизм данных принес какой-то положительный эффект. В противном случае последовательное выполнение цикла может завершиться быстрее.

```
// Применить метод Parallel.For() для организации параллельно
// выполняемого цикла обработки данных.

using System;
using System.Threading.Tasks;

class DemoParallelFor {
  static int[] data;

  // Метод, служащий в качестве тела параллельно выполняемого цикла.
  // Операторы этого цикла просто расходуют время ЦП для целей демонстрации.
  static void MyTransform(int i) {
    data[i] = data[i] / 10;

    if(data[i] < 10000) data[i] = 0;
    if(data[i] > 10000 & data[i] < 20000) data[i] = 100;
    if(data[i] > 20000 & data[i] < 30000) data[i] = 200;
    if(data[i] > 30000) data[i] = 300;
  }

static void Main() {
    Console.WriteLine("Основной поток запущен.");

    data = new int[100000000];

    // Инициализировать данные в обычном цикле for.
    for(int i=0; i < data.Length; i++) data[i] = i;

    //  Распараллелить цикл методом For().
    Parallel.For(0, data.Length, MyTransform);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Эта программа состоит из двух циклов. В первом, стандартном, цикле `for` инициализируется массив `data`. А во втором цикле, выполняемом параллельно методом `For()`, над каждым элементом массива `data` производится преобразование. Как упоминалось выше, это преобразование носит произвольный характер и выбрано лишь для целей демонстрации. Метод `For()` автоматически разбивает вызовы метода `MyTransform()` на части для параллельной обработки отдельных порций данных, хранящихся в массиве. Следовательно, если запустить данную программу на компьютере с двумя доступными процессорами или больше, то цикл преобразования данных в массиве может быть выполнен методом `For()` параллельно.

Следует, однако, иметь в виду, что далеко не все циклы могут выполняться эффективно, когда они распараллеливаются. Как правило, мелкие циклы, а также циклы, состоящие из очень простых операций, выполняются быстрее последовательным способом, чем параллельным. Именно поэтому цикл `for` инициализации массива данных не распараллеливается методом `For()` в рассматриваемой здесь программе. Распараллеливание мелких и очень простых циклов может оказаться неэффективным потому, что время, требующееся для организации параллельных задач, а также время, расходуемое на переключение контекста, превышает время, экономящееся благодаря параллелизму. В подтверждение этого факта в приведенном ниже примере программы

создаются последовательный и параллельный варианты цикла for, а для сравнения на экран выводится время выполнения каждого из них.

```
// Продемонстрировать отличия во времени последовательного
// и параллельного выполнения цикла for.

using System;
using System.Threading.Tasks;
using System.Diagnostics;

class DemoParallelFor {
  static int[] data;

  // Метод, служащий в качестве тела параллельно выполняемого цикла.
  // Операторы этого цикла просто расходуют время ЦП для целей демонстрации.
  static void MyTransform(int i) {
    data[i] = data[i] / 10;

    if(data[i] < 1000) data[i] = 0;
    if(data[i] > 1000 & data[i] < 2000) data[i] = 100;
    if(data[i] > 2000 & data[i] < 3000) data[i] = 200;
    if(data[i] > 3000) data[i] = 300;
  }

  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    // Create экземпляр объекта типа Stopwatch
    // для хранения времени выполнения цикла.
    Stopwatch sw = new Stopwatch();

    data = new int[100000000];

    // Инициализировать данные.
    sw.Start();

    // Параллельный вариант инициализации массива в цикле.
    Parallel.For(0, data.Length, (i) => data[i] = i );

    sw.Stop();

    Console.WriteLine("Параллельно выполняемый цикл инициализации: " +
                      "{0} секунд", sw.Elapsed.TotalSeconds);

    sw.Reset();

    sw.Start();

    // Последовательный вариант инициализации массива в цикле.
    for(int i=0; i < data.Length; i++) data[i] = i;

    sw.Stop();

    Console.WriteLine("Последовательно выполняемый цикл инициализации: " +
                      "{0} секунд", sw.Elapsed.TotalSeconds);

    Console.WriteLine();
```

```
    // Выполнить преобразования.
    sw.Start();

    // Параллельный вариант преобразования данных в цикле.
    Parallel.For(0, data.Length, MyTransform);

    sw.Stop();

    Console.WriteLine("Параллельно выполняемый цикл преобразования: " +
                      "{0} секунд", sw.Elapsed.TotalSeconds);

    sw.Reset();

    sw.Start();

    // Последовательный вариант преобразования данных в цикле.
    for(int i=0; i < data.Length; i++) MyTransform(i);

    sw.Stop();

    Console.WriteLine("Последовательно выполняемый цикл преобразования: " +
                      "{0} секунд", sw.Elapsed.TotalSeconds);

    Console.WriteLine("Основной поток завершен.");
  }
}
```

При выполнении этой программы на двухъядерном компьютере получается следующий результат.

```
Основной поток запущен.
Параллельно выполняемый цикл инициализации: 1.0537757 секунд
Последовательно выполняемый цикл инициализации: 0.3457628 секунд

Параллельно выполняемый цикл преобразования: 4.2246675 секунд
Последовательно выполняемый цикл преобразования: 5.3849959 секунд
Основной поток завершен.
```

Прежде всего, обратите внимание на то, что параллельный вариант цикла инициализации массива данных выполняется приблизительно в три раза медленнее, чем последовательный. Дело в том, что в данном случае на операцию присваивания расходуется так мало времени, что издержки на дополнительно организуемое распараллеливание превышают экономию, которую оно дает. Обратите далее внимание на то, что параллельный вариант цикла преобразования данных выполняется быстрее, чем последовательный. В данном случае экономия от распараллеливания с лихвой возмещает издержки на его дополнительную организацию.

ПРИМЕЧАНИЕ

Как правило, в отношении преимуществ, которые дает распараллеливание различных видов циклов, следует руководствоваться текущими рекомендациями корпорации Microsoft. Кроме того, необходимо убедиться в том, что распараллеливание цикла действительно приводит к повышению производительности, прежде чем использовать такой цикл в окончательно выпускаемом прикладном коде.

Что касается приведенной выше программы, то необходимо упомянуть о двух других ее особенностях. Во-первых, обратите внимание на то, что в параллельно выполняемом цикле для инициализации данных применяется лямбда-выражение, как показано ниже.

```
Parallel.For(0, data.Length, (i) => data[i] = i );
```

Здесь "тело" цикла указывается в лямбда-выражении. (Напомним, что в лямбда-выражении создается анонимный метод.) Следовательно, для параллельного выполнения методом `For()` совсем не обязательно указывать именованный метод.

И во-вторых, обратите внимание на применение класса `Stopwatch` для вычисления времени выполнения цикла. Этот класс находится в пространстве имен `System.Diagnostics`. Для того чтобы воспользоваться им, достаточно создать экземпляр его объекта, а затем вызвать метод `Start()`, начинающий отчет времени, и далее — метод `Stop()`, завершающий отсчет времени. А с помощью метода `Reset()` отсчет времени сбрасывается в исходное состояние. Продолжительность выполнения можно получить различными способами. В рассматриваемой здесь программе для этой цели использовано свойство `Elapsed`, возвращающее объект типа `TimeSpan`. С помощью этого объекта и свойства `TotalSeconds` время отображается в секундах, включая и доли секунды. Как показывает пример рассматриваемой здесь программы, класс `Stopwatch` оказывается весьма полезным при разработке параллельно исполняемого кода.

Как упоминалось выше, метод `For()` возвращает экземпляр объекта типа `ParallelLoopResult`. Это структура, в которой определяются два следующих свойства.

```
public bool IsCompleted { get; }
public Nullable<long> LowestBreakIteration { get; }
```

Свойство `IsCompleted` будет иметь логическое значение `true`, если выполнены все шаги цикла. Иными словами, при нормальном завершении цикла это свойство будет содержать логическое значение `true`. Если же выполнение цикла прервется раньше времени, то данное свойство будет содержать логическое значение `false`. Свойство `LowestBreakIteration` будет содержать наименьшее значение переменной управления циклом, если цикл прервется раньше времени вызовом метода `ParallelLoopState.Break()`.

Для доступа к объекту типа `ParallelLoopState` следует использовать форму метода `For()`, делегат которого принимает в качестве второго параметра текущее состояние цикла. Ниже эта форма метода `For()` приведена в простейшем виде.

```
public static ParallelLoopResult For(int fromInclusive, int toExclusive,
                                      Action<int, ParallelLoopState> body)
```

В данной форме делегат `Action`, описывающий тело цикла, определяется следующим образом.

```
public delegate void Action<in T1, in T2>(T arg1, T2 arg2)
```

Для метода `For()` обобщенный параметр `T1` должен быть типа `int`, а обобщенный параметр `T2` — типа `ParallelLoopState`. Всякий раз, когда делегат `Action` вызывается, текущее состояние цикла передается в качестве аргумента *arg2*.

Для преждевременного завершения цикла следует воспользоваться методом `Break()`, вызываемым для экземпляра объекта типа `ParallelLoopState` внутри тела цикла, определяемого параметром *body*. Метод `Break()` объявляется следующим образом.

```
public void Break()
```

Вызов метода Break() формирует запрос на как можно более раннее прекращение параллельно выполняемого цикла, что может произойти через несколько шагов цикла после вызова метода Break(). Но все шаги цикла до вызова метода Break() все же выполняются. Следует, также иметь в виду, что отдельные части цикла могут и не выполняться параллельно. Так, если выполнено 10 шагов цикла, то это еще не означает, что все эти 10 шагов представляют 10 первых значений переменной управления циклом.

Прерывание цикла, параллельно выполняемого методом For(), нередко оказывается полезным при поиске данных. Так, если искомое значение найдено, то продолжать выполнение цикла нет никакой надобности. Прерывание цикла может оказаться полезным и в том случае, если во время очередной операции встретились недостоверные данные.

В приведенном ниже примере программы демонстрируется применение метода Break() для прерывания цикла, параллельно выполняемого методом For(). Это вариант предыдущего примера, переработанный таким образом, чтобы метод MyTransform() принимал теперь объект типа ParallelLoopState в качестве своего параметра, а метод Break() вызывался при обнаружении отрицательного значения в массиве данных. Отрицательное значение, по которому прерывается выполнение цикла, вводится в массив data внутри метода Main(). Далее проверяется состояние завершения цикла преобразования данных. Свойство IsCompleted будет содержать логическое значение false, поскольку в массиве data обнаруживается отрицательное значение. При этом на экран выводится номер шага, на котором цикл был прерван. (В этой программе исключены все избыточные циклы, применявшиеся в ее предыдущей версии, а оставлены только самые эффективные из них: последовательно выполняемый цикл инициализации и параллельно выполняемый цикл преобразования.)

```
// Использовать объекты типа ParallelLoopResult и ParallelLoopState, а также
// метод Break() вместе с методом For() для параллельного выполнения цикла.

using System;
using System.Threading.Tasks;

class DemoParallelForWithLoopResult {
  static int[] data;

  // Метод, служащий в качестве тела параллельно выполняемого цикла.
  // Операторы этого цикла просто расходуют время ЦП для целей демонстрации.
  static void MyTransform(int i, ParallelLoopState pls) {

    // Прервать цикл при обнаружении отрицательного значения.
    if(data[i] < 0) pls.Break();

    data[i] = data[i] / 10;

    if(data[i] < 1000) data[i] = 0;
    if(data[i] > 1000 & data[i] < 2000) data[i] = 100;
    if(data[i] > 2000 & data[i] < 3000) data[i] = 200;
    if(data[i] > 3000) data[i] = 300;
  }

  static void Main() {
```

```
    Console.WriteLine("Основной поток запущен.");
    data = new int[100000000];

    // Инициализировать данные.
    for(int i=0; i < data.Length; i++) data[i] = i;

    // Поместить отрицательное значение в массив data.
    data[1000] = -10;

    // Параллельный вариант инициализации массива в цикле.
    ParallelLoopResult loopResult = Parallel.For(0, data.Length, MyTransform);

    // Проверить, завершился ли цикл.
    if(!loopResult.IsCompleted)
      Console.WriteLine("\nЦикл завершился преждевременно из-за того, " +
                        "что обнаружено отрицательное значение\n" +
                         "на шаге цикла номер " +
                         loopResult.LowestBreakIteration + ".\n");

    Console.WriteLine("Основной поток завершен.");
  }
}
```

Выполнение этой программы может привести, например, к следующему результату.

```
Основной поток запущен.

Цикл завершился преждевременно из-за того, что обнаружено отрицательное значение
на шаге цикла номер 1000

Основной поток завершен.
```

Как следует из приведенного выше результата, цикл преобразования данных преждевременно завершается после 1000 шагов. Дело в том, что метод `Break()` вызывается внутри метода `MyTransform()` при обнаружении в массиве данных отрицательного значения.

Помимо двух описанных выше форм метода `For()` существует и ряд других его форм. В одних из этих форм допускается указывать различные дополнительные параметры, а в других — использовать параметры типа `long` вместо `int` для пошагового выполнения цикла. Имеются также формы метода `For()`, предоставляющие такие дополнительные преимущества, как, например, возможность указывать метод, вызываемый по завершении потока каждого цикла.

И еще одно, последнее замечание: если требуется остановить цикл, параллельно выполняемый методом `For()`, не обращая особого внимания на любые шаги цикла, которые еще могут быть в нем выполнены, то для этой цели лучше воспользоваться методом `Stop()`, чем методом `Break()`.

Применение метода `ForEach()`

Используя метод `ForEach()`, можно создать распараллеливаемый вариант цикла `foreach`. Существует несколько форм метода `ForEach()`. Ниже приведена простейшая форма его объявления:

```
public static ParallelLoopResult
              ForEach<TSource>(IEnumerable<TSource> source,
                               Action<TSource> body)
```

где *source* обозначает коллекцию данных, обрабатываемых в цикле, а *body* — метод, который будет выполняться на каждом шаге цикла. Как пояснялось ранее в этой книге, во всех массивах, коллекциях (описываемых в главе 25) и других источниках данных поддерживается интерфейс `IEnumerable<T>`. Метод, передаваемый через параметр *body*, принимает в качестве своего аргумента значение или ссылку на каждый обрабатываемый в цикле элемент массива, но не его индекс. А в итоге возвращаются сведения о состоянии цикла.

Аналогично методу `For()`, параллельное выполнение цикла методом `ForEach()` можно остановить, вызвав метод `Break()` для экземпляра объекта типа `ParallelLoopState`, передаваемого через параметр `body`, при условии, что используется приведенная ниже форма метода `ForEach()`.

```
public static ParallelLoopResult
              ForEach<TSource>(IEnumerable<TSource> source,
                               Action<TSource, ParallelLoopState> body)
```

В приведенном ниже примере программы демонстрируется применение метода `ForEach()` на практике. Как и прежде, в данном примере создается крупный массив целых значений. А отличается данный пример от предыдущих тем, что метод, выполняющийся на каждом шаге цикла, просто выводит на консоль значения из массива. Как правило, метод `WriteLine()` в распараллеливаемом цикле не применяется, потому что ввод-вывод на консоль осуществляется настолько медленно, что цикл оказывается полностью привязанным к вводу-выводу. Но в данном примере метод `WriteLine()` применяется исключительно в целях демонстрации возможностей метода `ForEach()`. При обнаружении отрицательного значения выполнение цикла прерывается вызовом метода `Break()`. Несмотря на то что метод `Break()` вызывается в одной задаче, другая задача может по-прежнему выполняться в течение нескольких шагов цикла, прежде чем он будет прерван, хотя это зависит от конкретных условий работы среды выполнения.

```
// Использовать объекты типа ParallelLoopResult и ParallelLoopState, а также
// метод Break() вместе с методом ForEach() для параллельного выполнения цикла.

using System;
using System.Threading.Tasks;

class DemoParallelForWithLoopResult {
  static int[] data;

  // Метод, служащий в качестве тела параллельно выполняемого цикла.
  // В данном примере переменной v передается значение элемента массива
  // данных, а не индекс этого элемента.
  static void DisplayData(int v, ParallelLoopState pls) {

  // Прервать цикл при обнаружении отрицательного значения.
  if(v < 0) pls.Break();

    Console.WriteLine("Значение: " + v);
  }
```

```
  static void Main() {

    Console.WriteLine("Основной поток запущен.");

    data = new int[100000000];

    // Инициализировать данные.
    for(int i=0; i < data.Length; i++) data[i] = i;

    // Поместить отрицательное значение в массив data.
    data[100000] = -10;

    // Использовать цикл, параллельно выполняемый методом ForEach(),
    // для отображения данных на экране.
    ParallelLoopResult loopResult = Parallel.ForEach(data, DisplayData);

    // Проверить, завершился ли цикл.
    if(!loopResult.IsCompleted)
      Console.WriteLine("\nЦикл завершился преждевременно из-за того, " +
                        "что обнаружено отрицательное значение\n" +
                         "на шаге цикла номер " +
                         loopResult.LowestBreakIteration + ".\n");

    Console.WriteLine("Основной поток завершен.");
  }
}
```

В приведенной выше программе именованный метод применяется в качестве делегата, представляющего "тело" цикла. Но иногда удобнее применять анонимный метод. В качестве примера ниже приведено реализуемое в виде лямбда-выражения "тело" цикла, параллельно выполняемого методом `ForEach()`.

```
// Использовать цикл, параллельно выполняемый методом ForEach(),
// для отображения данных на экране.
ParallelLoopResult loopResult =
            Parallel.ForEach(data, (v, pls) => {
              Console.WriteLine("Значение: " + v);
              if(v < 0) pls.Break();
            });
```

Исследование возможностей PLINQ

PLINQ представляет собой параллельный вариант языка интегрированных запросов LINQ и тесно связан с библиотекой TPL. PLINQ применяется, главным образом, для достижения параллелизма данных внутри запроса. Как станет ясно из дальнейшего, сделать это совсем не трудно. Как и TPL, тема PLINQ довольно обширна и многогранна, поэтому в этой главе представлены лишь самые основные понятия данного языка.

Класс `ParallelEnumerable`

Основу PLINQ составляет класс `ParallelEnumerable`, определенный в пространстве имен `System.Linq`. Это статический класс, в котором определены многие

методы расширения, поддерживающие параллельное выполнение операций. По существу, он представляет собой параллельный вариант стандартного для LINQ класса `Enumerable`. Многие его методы являются расширением класса `ParallelQuery`, а некоторые из них возвращают объект типа `ParallelQuery`. В классе `ParallelQuery` инкапсулируется последовательность операций, поддерживающая параллельное выполнение. Имеются как обобщенный, так и необобщенный варианты данного класса. Мы не будем обращаться к классу `ParallelQuery` непосредственно, а воспользуемся несколькими методами класса `ParallelEnumerable`. Самый главный из них, метод `AsParallel()`, описывается в следующем разделе.

Распараллеливание запроса методом `AsParallel()`

Едва ли не самым удобным средством PLINQ является возможность просто создавать параллельный запрос. Нужно лишь вызвать метод `AsParallel()` для источника данных. Метод `AsParallel()` определен в классе `ParallelEnumerable` и возвращает источник данных, инкапсулированный в экземпляре объекта типа `ParallelQuery`. Это дает возможность поддерживать методы расширения параллельных запросов. После вызова данного метода запрос разделяет источник данных на части и оперирует с каждой из них таким образом, чтобы извлечь максимальную выгоду из распараллеливания. (Если распараллеливание оказывается невозможным или неприемлемым, то запрос, как обычно, выполняется последовательно.) Таким образом, добавления в исходный код единственного вызова метода `AsParallel()` оказывается достаточно для того, чтобы превратить последовательный запрос LINQ в параллельный запрос LINQ. Для простых запросов это единственное необходимое условие.

Существуют как обобщенные, так и необобщенные формы метода `AsParallel()`. Ниже приведена простейшая обобщенная его форма:

```
public static ParallelQuery AsParallel(this IEnumerable source)
public static ParallelQuery<TSource>
                AsParallel<TSource>(this IEnumerable<TSource> source)
```

где `TSource` обозначает тип элементов в последовательном источнике данных *source*.

Ниже приведен пример, демонстрирующий простой запрос PLINQ.

```
// Простой запрос PLINQ.

using System;
using System.Linq;

class PLINQDemo {
  static void Main() {
    int[] data = new int[10000000];

    // Инициализировать массив данных положительными значениями.
    for(int i=0; i < data.Length; i++) data[i] = i;

    // А теперь ввести в массив данных ряд отрицательных значений.
    data[1000] = -1;
    data[14000] = -2;
    data[15000] = -3;
    data[676000] = -4;
```

```
    data[8024540] = -5;
    data[9908000] = -6;

    // Использовать запрос PLINQ для поиска отрицательных значений.
    var negatives = from val in data.AsParallel()
                    where val < 0
                    select val;

    foreach(var v in negatives)
      Console.Write(v + " ");

    Console.WriteLine();
  }
}
```

Эта программа начинается с создания крупного массива `data`, инициализируемого целыми положительными значениями. Затем в него вводится ряд отрицательных значений. А далее формируется запрос на возврат последовательности отрицательных значений. Ниже приведен этот запрос.

```
var negatives = from val in data.AsParallel()
                where val < 0
                select val;
```

В этом запросе метод `AsParallel()` вызывается для источника данных, в качестве которого служит массив `data`. Благодаря этому разрешается параллельное выполнение операций над массивом `data`, а именно: поиск отрицательных значений параллельно в нескольких потоках. По мере обнаружения отрицательных значений они добавляются в последовательность вывода. Это означает, что порядок формирования последовательности вывода может и не отражать порядок расположения отрицательных значений в массиве `data`. В качестве примера ниже приведен результат выполнения приведенного выше кода в двухъядерной системе.

```
-5 -6 -1 -2 -3 -4
```

Как видите, в том потоке, где поиск выполнялся в верхней части массива, отрицательные значения –5 и –6 были обнаружены раньше, чем значение -1 в том потоке, где поиск происходил в нижней части массива. Следует, однако, иметь в виду, что из-за отличий в степени загрузки задачами, количества доступных процессоров и прочих факторов системного характера могут быть получены разные результаты. А самое главное, что результирующая последовательность совсем не обязательно будет отражать порядок формирования исходной последовательности.

Применение метода `AsOrdered()`

Как отмечалось в предыдущем разделе, по умолчанию порядок формирования результирующей последовательности в параллельном запросе совсем не обязательно должен отражать порядок формирования исходной последовательности. Более того, результирующую последовательность следует рассматривать как практически неупорядоченную. Если же результат должен отражать порядок организации источника данных, то его нужно запросить специально с помощью метода `AsOrdered()`, определенного в классе `ParallelEnumerable`. Ниже приведены обобщенная и необобщенная формы этого метода:

```
public static ParallelQuery AsOrdered(this ParallelQuery source)
public static ParallelQuery<TSource>
                AsOrdered<TSource>(this ParallelQuery<TSource> source)
```

где `TSource` обозначает тип элементов в источнике данных *source*. Метод `AsOrdered()` можно вызывать только для объекта типа `ParallelQuery`, поскольку он является методом расширения класса `ParallelQuery`.

Для того чтобы посмотреть, к какому результату может привести применение метода `AsOrdered()`, подставьте его вызов в приведенный ниже запрос из предыдущего примера программы.

```
// Использовать метод AsOrdered() для сохранения порядка
// в результирующей последовательности.
var negatives = from val in data.AsParallel().AsOrdered()
                where val < 0
                select val;
```

После выполнения программы порядок следования элементов в результирующей последовательности будет отражать порядок их расположения в исходной последовательности.

Отмена параллельного запроса

Параллельный запрос отменяется таким же образом, как и задача. И в том и в другом случае отмена опирается на структуру `CancellationToken`, получаемую из класса `CancellationTokenSource`. Получаемый в итоге признак отмены передается запросу с помощью метода `WithCancellation()`. Отмена параллельного запроса производится методом `Cancel()`, который вызывается для источника признаков отмены. Главное отличие отмены параллельного запроса от отмены задачи состоит в следующем: когда параллельный запрос отменяется, он генерирует исключение `OperationCanceledException`, а не `AggregateException`. Но в тех случаях, когда запрос способен сгенерировать несколько исключений, исключение `OperationCanceledException` может быть объединено в совокупное исключение `AggregateException`. Поэтому отслеживать лучше оба вида исключений.

Ниже приведена форма объявления метода `WithCancellation()`:

```
public static ParallelQuery<TSource>
                WithCancellation<TSource> (
                    this ParallelQuery<TSource> source,
                    CancellationToken cancellationToken)
```

где *source* обозначает вызывающий запрос, а *cancellationToken* — признак отмены. Этот метод возвращает запрос, поддерживающий указанный признак отмены.

В приведенном ниже примере программы демонстрируется порядок отмены параллельного запроса, сформированного в программе из предыдущего примера. В данной программе организуется отдельная задача, которая ожидает в течение 100 миллисекунд, а затем отменяет запрос. Отдельная задача требуется потому, что цикл `foreach`, в котором выполняется запрос, блокирует выполнение метода `Main()` до завершения цикла.

```
// Отменить параллельный запрос.

using System;
```

```
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class PLINQCancelDemo {

  static void Main() {
    CancellationTokenSource cancelTokSrc = new CancellationTokenSource();
    int[] data = new int[10000000];

    // Инициализировать массив данных положительными значениями.
    for(int i=0; i < data.Length; i++) data[i] = i;

    // А теперь ввести в массив данных ряд отрицательных значений.
    data[1000] = -1;
    data[14000] = -2;
    data[15000] = -3;
    data[676000] = -4;
    data[8024540] = -5;
    data[9908000] = -6;

    // Использовать запрос PLINQ для поиска отрицательных значений.
    var negatives = from val in data.AsParallel().
                                WithCancellation(cancelTokSrc.Token)
                    where val < 0
                    select val;

    // Создать задачу для отмены запроса по истечении 100 миллисекунд.
    Task cancelTsk = Task.Factory.StartNew( () => {
                       Thread.Sleep(100);
                       cancelTokSrc.Cancel();
                     });

    try {
      foreach(var v in negatives)
        Console.Write(v + " ");
    } catch(OperationCanceledException exc) {
      Console.WriteLine(exc.Message);
    } catch(AggregateException exc) {
      Console.WriteLine(exc);
    } finally {
      cancelTsk.Wait();
      cancelTokSrc.Dispose();
      cancelTsk.Dispose();
    }

    Console.WriteLine();
  }
}
```

Ниже приведен результат выполнения этой программы. Если запрос отменяется до его завершения, то на экран выводится только сообщение об исключительной ситуации.

```
Запрос отменен с помощью маркера, переданного в метод WithCancellation.
```

Другие средства PLINQ

Как упоминалось ранее, PLINQ представляет собой довольно крупную подсистему. Это объясняется отчасти той гибкостью, которой обладает PLINQ. В PLINQ доступны и многие другие средства, помогающие подстраивать параллельные запросы под конкретную ситуацию. Так, при вызове метода `WithDegreeOfParallelism()` можно указать максимальное количество процессоров, выделяемых для обработки запроса, а при вызове метода `AsSequential()` — запросить последовательное выполнение части параллельного запроса. Если вызывающий поток, ожидающий результатов от цикла `foreach`, не требуется блокировать, то для этой цели можно воспользоваться методом `ForAll()`. Все эти методы определены в классе `ParallelEnumerable`. А в тех случаях, когда PLINQ должен по умолчанию поддерживать последовательное выполнение, можно воспользоваться методом `WithExecutionMode()`, передав ему в качестве параметра признак `ParallelExecutionMode.ForceParallelism`.

Вопросы эффективности PLINQ

Далеко не все запросы выполняются быстрее только потому, что они распараллелены. Как пояснялось ранее в отношении TPL, издержки, связанные с созданием параллельных потоков и управлением их исполнением, могут "перекрыть" все преимущества, которые дает распараллеливание. Вообще говоря, если источник данных оказывается довольно мелким, а требующаяся обработка данных — очень короткой, то внедрение параллелизма может и не привести к ускорению обработки запроса. Поэтому за рекомендациями по данному вопросу следует обращаться к информации корпорации Microsoft.

25
ГЛАВА

Коллекции, перечислители и итераторы

В этой главе речь пойдет об одной из самых важных составляющих среды .NET Framework: коллекциях. В C# *коллекция* представляет собой совокупность объектов. В среде .NET Framework имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций. Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных. Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц. Коллекции являются современным технологическим средством, заслуживающим пристального внимания всех, кто программирует на C#.

Первоначально существовали только классы необобщенных коллекций. Но с внедрением обобщений в версии C# 2.0 среда .NET Framework была дополнена многими новыми обобщенными классами и интерфейсами. Благодаря введению обобщенных коллекций общее количество классов и интерфейсов удвоилось. Вместе с библиотекой распараллеливания задач (TPL) в версии 4.0 среды .NET Framework появился ряд новых классов коллекций, предназначенных для применения в тех случаях, когда доступ к коллекции осуществляется из нескольких потоков. Нетрудно догадаться, что прикладной интерфейс Collections API составляет значительную часть среды .NET Framework.

Кроме того, в настоящей главе рассматриваются два средства, непосредственно связанные с коллекциями: перечислители и итераторы. И те и другие позволяют поочередно обращаться к содержимому класса коллекции в цикле `foreach`.

Краткий обзор коллекций

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе `ArrayList`, `Hashtable`, `Stack` и `Queue`, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные. Необобщенные коллекции реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также *словари*, в которых можно хранить пары "ключ–значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа `object`.

Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Очевидно, что такие коллекции не типизированы, поскольку в них хранятся ссылки на данные типа `object`. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен `System.Collections`.

Специальные коллекции оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен `System.Collections.Specialized`.

В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это `BitArray`. Коллекция типа `BitArray` поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И или исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа `BitArray` объявляется в пространстве имен `System.Collections`.

Обобщенные коллекции обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен `System.Collections.Generic`.

Параллельные коллекции поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен `System.Collections.Concurrent`.

В пространстве имен `System.Collections.ObjectModel` находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций.

Основополагающим для всех коллекций является понятие *перечислителя*, который поддерживается в необобщенных интерфейсах `IEnumerator` и `IEnumerable`, а также в обобщенных интерфейсах `IEnumerator<T>` и `IEnumerable<T>`. Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции. Следовательно, он *перечисляет* содержимое коллекции. В каждой коллекции должна быть

реализована обобщенная или необобщенная форма интерфейса `IEnumerable`, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в интерфейсе `IEnumerator` или `IEnumerator<T>`. Это означает, что, внеся минимальные изменения в код циклического обращения к коллекции одного типа, его можно использовать для аналогичного обращения к коллекции другого типа. Любопытно, что для поочередного обращения к содержимому коллекции в цикле `foreach` используется перечислитель.

Основополагающим для всех коллекций является понятие *перечислителя*, который поддерживается в необобщенных интерфейсах `IEnumerator` и `IEnumerable`, а также в обобщенных интерфейсах `IEnumerator<T>` и `IEnumerable<T>`. Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции. Следовательно, он *перечисляет* содержимое коллекции. В каждой коллекции должна быть реализована обобщенная или необобщенная форма интерфейса `IEnumerable`, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в интерфейсе `IEnumerator` или `IEnumerator<T>`. Это означает, что, внеся минимальные изменения в код циклического обращения к коллекции одного типа, его можно использовать для аналогичного обращения к коллекции другого типа. Любопытно, что для поочередного обращения к содержимому коллекции в цикле `foreach` используется перечислитель.

С перечислителем непосредственно связано другое средство, называемое *итератором*. Это средство упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле `foreach`. Итераторы также рассматриваются в этой главе.

И последнее замечание: если у вас имеется некоторый опыт программирования на C++, то вам, вероятно, будет полезно знать, что классы коллекций по своей сути подобны классам стандартной библиотеки шаблонов (Standard Template Library — STL), определенной в C++. То, что в программировании на C++ называется *контейнером*, в программировании на C# называется *коллекцией*. Это же относится и к Java. Если вы знакомы с библиотекой Collections Framework для Java, то научиться пользоваться коллекциями в C# не составит для вас большого труда.

В силу характерных отличий каждый из пяти типов коллекций (необобщенных, обобщенных, специальных, с поразрядной организацией и параллельных) будет рассмотрен далее в этой главе отдельно.

Необобщенные коллекции

Необобщенные коллекции вошли в состав среды .NET Framework еще в версии 1.0. Они определяются в пространстве имен `System.Collections`. Необобщенные коллекции представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Таким образом, они позволяют манипулировать объектом любого типа, хотя и не типизированным способом. В этом состоит их преимущество и в то же время недостаток. Благодаря тому что необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные. Это удобно в тех случаях, когда требуется манипулировать совокупностью разнотипных объектов или же когда типы хранящихся в коллекции объектов заранее неизвестны. Но если коллекция предназначается для хранения объекта конкретного типа, то необобщенные коллекции не обеспечивают типовую безопасность, которую можно обнаружить в обобщенных коллекциях.

Необобщенные коллекции определены в ряде интерфейсов и классов, реализующих эти интерфейсы. Все они рассматриваются далее по порядку.

Интерфейсы необобщенных коллекций

В пространстве имен `System.Collections` определен целый ряд интерфейсов необобщенных коллекций. Начинать рассмотрение необобщенных коллекций следует именно с интерфейсов, поскольку они определяют функциональные возможности, которые являются общими для всех классов необобщенных коллекций. Интерфейсы, служащие опорой для необобщенных коллекций, сведены в табл. 25.1. Каждый из этих интерфейсов подробно описывается далее.

Таблица 25.1. Интерфейсы необобщенных коллекций

Интерфейс	Описание
`ICollection`	Определяет элементы, которые должны иметь все необобщенные коллекции
`IComparer`	Определяет метод `Compare()` для сравнения объектов, хранящихся в коллекции
`IDictionary`	Определяет коллекцию, состоящую из пар "ключ-значение"
`IDictionaryEnumerator`	Определяет перечислитель для коллекции, реализующей интерфейс `IDictionary`
`IEnumerable`	Определяет метод `GetEnumerator()`, предоставляющий перечислитель для любого класса коллекции
`IEnumerator`	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
`IEqualityComparer`	Сравнивает два объекта на предмет равенства
`IHashCodeProvider`	Считается устаревшим. Вместо него следует использовать интерфейс `IEqualityComparer`
`IList`	Определяет коллекцию, доступ к которой можно получить с помощью индексатора
`IStructuralComparable`	Определяет метод `CompareTo()`, применяемый для структурного сравнения
`IStructuralEquatable`	Определяет метод `Equals()`, применяемый для выяснения структурного, а не ссылочного равенства. Кроме того, определяет метод `GetHashCode()`

Интерфейс `ICollection`

Интерфейс `ICollection` служит основанием, на котором построены все необобщенные коллекции. В нем объявляются основные методы и свойства для всех необобщенных коллекций. Он также наследует от интерфейса `IEnumerable`.

В интерфейсе `ICollection` определяются перечисленные ниже свойства.

Свойство `Count` используется чаще всего, поскольку оно содержит количество элементов, хранящихся в коллекции на данный момент. Если значение свойства `Count` равно нулю, то коллекция считается пустой.

В интерфейсе `ICollection` определяется следующий метод.

```
void CopyTo(Array target, int startIdx)
```

Свойство	Назначение
`int Count { get; }`	Содержит количество элементов в коллекции на данный момент
`bool IsSynchronized { get; }`	Принимает логическое значение `true`, если коллекция синхронизирована, а иначе — логическое значение `false`. По умолчанию коллекции не синхронизированы. Но для большинства коллекций можно получить синхронизированный вариант
`object SyncRoot { get; }`	Содержит объект, для которого коллекция может быть синхронизирована

Метод `CopyTo()` копирует содержимое коллекции в массив *target*, начиная с элемента, указываемого по индексу *startIdx*. Следовательно, метод `CopyTo()` обеспечивает в C# переход от коллекции к стандартному массиву.

Благодаря тому что интерфейс `ICollection` наследует от интерфейса `IEnumerable`, в его состав входит также единственный метод, определенный в интерфейсе `IEnumerable`. Это метод `GetEnumerator()`, объявляемый следующим образом.

```
IEnumerator GetEnumerator()
```

Он возвращает перечислитель для коллекции.

Вследствие того же наследования от интерфейса `IEnumerable` в интерфейсе `ICollection` определяются также четыре следующих метода расширения: `AsParallel()`, `AsQueryable()`, `Cast()` и `OfType()`. В частности, метод `AsParallel()` объявляется в классе `System.Linq.ParallelEnumerable`, метод `AsQueryable()` — в классе `System.Linq.Queryable`, а методы `Cast()` и `OfType()` — в классе `System.Linq.Enumerable`. Эти методы предназначены главным образом для поддержки LINQ, хотя их можно применять и в других целях.

Интерфейс `IList`

В интерфейсе `IList` объявляется такое поведение необобщенной коллекции, которое позволяет осуществлять доступ к ее элементам по индексу с отсчетом от нуля. Этот интерфейс наследует от интерфейсов `ICollection` и `IEnumerable`. Помимо методов, определенных в этих интерфейсах, в интерфейсе `IList` определяется ряд собственных методов. Все эти методы сведены в табл. 25.2. В некоторых из них предусматривается модификация коллекции. Если же коллекция доступна только для чтения или имеет фиксированный размер, то в этих методах генерируется исключение `NotSupportedException`.

Таблица 25.2. Методы, определенные в интерфейсе `IList`

Метод	Описание
`int Add(object value)`	Добавляет объект *value* в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохраняется
`void Clear()`	Удаляет все элементы из вызывающей коллекции
`bool Contains(object value)`	Возвращает логическое значение `true`, если вызывающая коллекция содержит объект *value*, а иначе — логическое значение `false`

Окончание табл. 25.2

Метод	Описание
`int IndexOf(object value)`	Возвращает индекс объекта *value*, если этот объект содержится в вызывающей коллекции. Если же объект *value* не обнаружен, то метод возвращает значение -1
`void Insert(int index, object value)`	Вставляет в вызывающую коллекцию объект *value* по индексу *index*. Элементы, находившиеся до этого по индексу *index* и дальше, смещаются вперед, чтобы освободить место для вставляемого объекта *value*
`void Remove(object value)`	Удаляет первое вхождение объекта *value* в вызывающей коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы устранить образовавшийся "пробел"
`void RemoveAt(int index)`	Удаляет из вызывающей коллекции объект, расположенный по указанному индексу *index*. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы устранить образовавшийся "пробел"

Объекты добавляются в коллекцию типа `IList` вызовом метода `Add()`. Обратите внимание на то, что метод `Add()` принимает аргумент типа `object`. А поскольку `object` является базовым классом для всех типов, то в необобщенной коллекции может быть сохранен объект любого типа, включая и типы значений, в силу автоматической упаковки и распаковки.

Для удаления элемента из коллекции служат методы `Remove()` и `RemoveAt()`. В частности, метод `Remove()` удаляет указанный объект, а метод `RemoveAt()` удаляет объект по указанному индексу. И для опорожнения коллекции вызывается метод `Clear()`.

Для того чтобы выяснить, содержится ли в коллекции конкретный объект, вызывается метод `Contains()`. Для получения индекса объекта вызывается метод `IndexOf()`, а для вставки элемента в коллекцию по указанному индексу — метод `Insert()`.

В интерфейсе `IList` определяются следующие свойства.

```
bool IsFixedSize { get; }
bool IsReadOnly { get; }
```

Если коллекция имеет фиксированный размер, то свойство `IsFixedSize` содержит логическое значение `true`. Это означает, что в такую коллекцию нельзя ни вставлять элементы, ни удалять их из нее. Если же коллекция доступна только для чтения, то свойство `IsReadOnly` содержит логическое значение `true`. Это означает, что содержимое такой коллекции не подлежит изменению.

Кроме того, в интерфейсе `IList` определяется следующий индексатор.

```
object this[int index] { get; set; }
```

Этот индексатор служит для получения и установки значения элемента коллекции. Но его нельзя использовать для добавления в коллекцию нового элемента. С этой целью обычно вызывается метод `Add()`. Как только элемент будет добавлен в коллекцию, он станет доступным посредством индексатора.

Интерфейс IDictionary

В интерфейсе IDictionary определяется такое поведение необобщенной коллекции, которое позволяет преобразовать уникальные ключи в соответствующие значения. Ключ представляет собой объект, с помощью которого значение извлекается впоследствии. Следовательно, в коллекции, реализующей интерфейс IDictionary, хранятся пары "ключ–значение". Как только подобная пара будет сохранена, ее можно извлечь с помощью ключа. Интерфейс IDictionary наследует от интерфейсов ICollection и IEnumerable. Методы, объявленные в интерфейсе IDictionary, сведены в табл. 25.3. Некоторые из них генерируют исключение ArgumentNullException при попытке указать пустой ключ, поскольку пустые ключи не допускаются.

Таблица 25.3. Методы, определенные в интерфейсе IDictionary

Метод	Описание
`void Add(object key, object value)`	Добавляет в вызывающую коллекцию пару "ключ-значение", определяемую параметрами *key* и *value*
`void Clear()`	Удаляет все пары "ключ-значение" из вызывающей коллекции
`bool Contains(object key)`	Возвращает логическое значение true, если вызывающая коллекция содержит объект *key* в качестве ключа, в противном случае — логическое значение false
`IDictionaryEnumerator GetEnumerator()`	Возвращает перечислитель для вызывающей коллекции
`void Remove(object key)`	Удаляет из коллекции элемент, ключ которого равен значению параметра *key*

Для добавления пары "ключ–значение" в коллекцию типа IDictionary служит метод Add(). Обратите внимание на то, что ключ и его значение указываются отдельно. А для удаления элемента из коллекции следует указать ключ этого объекта при вызове метода Remove(). И для опорожнения коллекции вызывается метод Clear().

Для того чтобы выяснить, содержит ли коллекция конкретный объект, вызывается метод Contains() с указанным ключом искомого элемента. С помощью метода GetEnumerator() получается перечислитель, совместимый с коллекцией типа IDictionary. Этот перечислитель оперирует парами "ключ–значение".

В интерфейсе IDictionary определяются перечисленные ниже свойства.

Свойство	Назначение
`bool IsFixedSize { get; }`	Принимает логическое значение true, если словарь имеет фиксированный размер
`bool IsReadOnly { get; }`	Принимает логическое значение true, если словарь доступен только для чтения
`ICollection Keys { get; }`	Получает коллекцию ключей
`ICollection Values { get; }`	Получает коллекцию значений

Следует иметь в виду, что ключи и значения, содержащиеся в коллекции, доступны в отдельных списках с помощью свойств Keys и Values.

Кроме того, в интерфейсе IDictionary определяется следующий индексатор.

```
object this[object key] { get; set; }
```

Этот индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. Но в качестве индекса в данном случае служит ключ элемента, а не собственно индекс.

Интерфейсы IEnumerable, IEnumerator и IDictionaryEnumerator

Интерфейс `IEnumerable` является необобщенным, и поэтому он должен быть реализован в классе для поддержки перечислителей. Как пояснялось выше, интерфейс `IEnumerable` реализуется во всех классах необобщенных коллекций, поскольку он наследуется интерфейсом `ICollection`. Ниже приведен единственный метод `GetEnumerator()`, определяемый в интерфейсе `IEnumerable`.

```
IEnumerator GetEnumerator()
```

Он возвращает коллекцию. Благодаря реализации интерфейса `IEnumerable` можно также получать содержимое коллекции в цикле `foreach`.

В интерфейсе `IEnumerator` определяются функции перечислителя. С помощью методов этого интерфейса можно циклически обращаться к содержимому коллекции. Если в коллекции содержатся пары "ключ–значение" (словари), то метод `GetEnumerator()` возвращает объект типа `IDictionaryEnumerator`, а не типа `IEnumerator`. Интерфейс `IDictionaryEnumerator` наследует от интерфейса `IEnumerator` и вводит дополнительные функции, упрощающие перечисление словарей.

В интерфейсе `IEnumerator` определяются также методы `MoveNext()` и `Reset()` и свойство `Current`. Способы их применения подробнее описываются далее в этой главе. А до тех пор следует отметить, что свойство `Current` содержит элемент, получаемый в текущий момент. Метод `MoveNext()` осуществляет переход к следующему элементу коллекции, а метод `Reset()` возобновляет перечисление с самого начала.

Интерфейсы IComparer и IEqualityComparer

В интерфейсе `IComparer` определяется метод `Compare()` для сравнения двух объектов.

```
int Compare(object x, object y)
```

Он возвращает положительное значение, если значение объекта *x* больше, чем у объекта *y*; отрицательное — если значение объекта *x* меньше, чем у объекта *y*; и нулевое — если сравниваемые значения равны. Данный интерфейс можно использовать для указания способа сортировки элементов коллекции.

В интерфейсе `IEqualityComparer` определяются два метода.

```
bool Equals(object x, object y)
int GetHashCode(object obj)
```

Метод `Equals()` возвращает логическое значение `true`, если значения объектов *x* и *y* равны. А метод `GetHashCode()` возвращает хеш-код для объекта *obj*.

Интерфейсы IStructuralComparable и IStructuralEquatable

Оба интерфейса `IStructuralComparable` и `IStructuralEquatable` добавлены в версию 4.0 среды .NET Framework. В интерфейсе `IStructuralComparable` определяется метод `CompareTo()`, который задает способ структурного сравнения двух объектов для целей сортировки. (Иными словами, метод `CompareTo()` сравнивает содержимое объектов, а не ссылки на них.) Ниже приведена форма объявления данного метода.

```
int CompareTo(object other, IComparer comparer)
```

Он должен возвращать –1, если вызывающий объект предшествует другому объекту *other*; 1, если вызывающий объект следует после объекта *other*; и наконец, 0, если значения обоих объектов одинаковы для целей сортировки. А само сравнение обеспечивает объект, передаваемый через параметр *comparer*.

Интерфейс `IStructuralEquatable` служит для выяснения структурного равенства путем сравнения содержимого двух объектов. В этом интерфейсе определены следующие методы.

```
bool Equals(object other, IEqualityComparer comparer)
int GetHashCode(IEqualityComparer comparer)
```

Метод `Equals()` должен возвращать логическое значение `true`, если вызывающий объект и другой объект *other* равны. А метод `GetHashCode()` должен возвращать хеш-код для вызывающего объекта. Результаты, возвращаемые обоими методами, должны быть совместимы. Само сравнение обеспечивает объект, передаваемый через параметр *comparer*.

Структура `DictionaryEntry`

В пространстве имен `System.Collections` определена структура `DictionaryEntry`. Необобщенные коллекции пар "ключ–значение" сохраняют эти пары в объекте типа `DictionaryEntry`. В данной структуре определяются два следующих свойства.

```
public object Key { get; set; }
public object Value { get; set; }
```

Эти свойства служат для доступа к ключу или значению, связанному с элементом коллекции. Объект типа `DictionaryEntry` может быть сконструирован с помощью конструктора:

```
public DictionaryEntry(object key, object value)
```

где *key* обозначает ключ, а *value* — значение.

Классы необобщенных коллекций

А теперь, когда представлены интерфейсы необобщенных коллекций, можно перейти к рассмотрению стандартных классов, в которых они реализуются. Ниже приведены классы необобщенных коллекций, за исключением коллекции типа `BitArray`, рассматриваемой далее в этой главе.

Класс	Описание
`ArrayList`	Определяет динамический массив, т.е. такой массив, который может при необходимости увеличивать свой размер
`Hashtable`	Определяет хеш-таблицу для пар "ключ-значение"
`Queue`	Определяет очередь, или список, действующий по принципу "первым пришел — первым обслужен"
`SortedList`	Определяет отсортированный список пар "ключ-значение"
`Stack`	Определяет стек, или список, действующий по принципу "первым пришел — последним обслужен"

Каждый из этих классов коллекций подробно рассматривается и демонстрируется далее на конкретных примерах.

Класс ArrayList

В классе ArrayList поддерживаются динамические массивы, расширяющиеся и сокращающиеся по мере необходимости. В языке C# стандартные массивы имеют фиксированную длину, которая не может изменяться во время выполнения программы. Это означает, что количество элементов в массиве нужно знать заранее. Но иногда требуемая конкретная длина массива остается неизвестной до самого момента выполнения программы. Именно для таких ситуаций и предназначен класс ArrayList. В классе ArrayList определяется массив переменной длины, который состоит из ссылок на объекты и может динамически увеличивать и уменьшать свой размер. Массив типа ArrayList создается с первоначальным размером. Если этот размер превышается, то массив автоматически расширяется. А при удалении объектов из такого массива он автоматически сокращается. Коллекции класса ArrayList широко применяются в практике программирования на C#. Именно поэтому они рассматриваются здесь подробно. Но многие способы применения коллекций класса ArrayList распространяются и на другие коллекции, в том числе и на обобщенные.

В классе ArrayList реализуются интерфейсы ICollection, IList, IEnumerable и ICloneable. Ниже приведены конструкторы класса ArrayList.

```
public ArrayList()
public ArrayList(ICollection c)
public ArrayList(int capacity)
```

Первый конструктор создает пустую коллекцию класса ArrayList с нулевой первоначальной емкостью. Второй конструктор создает коллекцию типа ArrayList с количеством инициализируемых элементов, которое определяется параметром *c* и равно первоначальной емкости массива. Третий конструктор создает коллекцию, имеющую указанную первоначальную емкость, определяемую параметром *capacity*. В данном случае емкость обозначает размер базового массива, используемого для хранения элементов коллекции. Емкость коллекции типа ArrayList может увеличиваться автоматически по мере добавления в нее элементов.

В классе ArrayList определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов класса ArrayList перечислены в табл. 25.4. Коллекцию класса ArrayList можно отсортировать, вызвав метод Sort(). В этом случае поиск в отсортированной коллекции с помощью метода BinarySearch() становится еще более эффективным. Содержимое коллекции типа ArrayList можно также обратить, вызвав метод Reverse().

Таблица 25.4. Наиболее часто используемые методы, определенные в классе ArrayList

Метод	Описание
public virtual void AddRange(Icollection *c*)	Добавляет элементы из коллекции *c* в конец вызывающей коллекции типа ArrayList
public virtual int BinarySearch(object *value*)	Выполняет поиск в вызывающей коллекции значения *value*. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован

Продолжение табл. 25.4

Метод	Описание
`public virtual int BinarySearch(object value, Icomparer comparer)`	Выполняет поиск в вызывающей коллекции значения *value*, используя для сравнения способ, определяемый параметром *comparer*. Возвращает индекс совпавшего элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual int BinarySearch(int index, int count, object value, IComparer comparer)`	Выполняет поиск в вызывающей коллекции значения *value*, используя для сравнения способ, определяемый параметром *comparer*. Поиск начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Метод возвращает индекс совпавшего элемента. Если искомое значение не найдено, метод возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual void CopyTo(Array array)`	Копирует содержимое вызывающей коллекции в массив *array*, который должен быть одномерным и совместимым по типу с элементами коллекции
`public virtual void CopyTo(Array array, int arrayIndex)`	Копирует содержимое вызывающей коллекции в массив *array*, начиная с элемента, указываемого по индексу *arrayIndex*. Целевой массив должен быть одномерным и совместимым по типу с элементами коллекции
`public virtual void CopyTo(int index, Array array, int arrayIndex, int count)`	Копирует часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемых параметром *count*, в массив *array*, начиная с элемента, указываемого по индексу *arrayIndex*. Целевой массив должен быть одномерным и совместимым по типу с элементами коллекции
`public static ArrayList FixedSize(ArrayList list)`	Заключает коллекцию *list* в оболочку типа `ArrayList` с фиксированным размером и возвращает результат
`public virtual ArrayList GetRange(int index, int count)`	Возвращает часть вызывающей коллекции типа `ArrayList`. Часть возвращаемой коллекции начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемое параметром *count*. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект
`public virtual int IndexOf(object value)`	Возвращает индекс первого вхождения объекта *value* в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1
`public virtual void InsertRange(int index, ICollection c)`	Вставляет элементы коллекции *c* в вызывающую коллекцию, начиная с элемента, указываемого по индексу *index*
`public virtual int LastIndexOf(object value)`	Возвращает индекс последнего вхождения объекта *value* в вызывающей коллекции. Если искомый объект не обнаружен, метод возвращает значение -1

Окончание табл. 25.4

Метод	Описание
`public static ArrayList ReadOnly(ArrayList list)`	Заключает коллекцию *list* в оболочку типа ArrayList, доступную только для чтения, и возвращает результат
`public virtual void RemoveRange(int index, int count)`	Удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемое параметром *count*
`public virtual void Reverse()`	Располагает элементы вызывающей коллекции в обратном порядке
`public virtual void Reverse(int index, int count)`	Располагает в обратном порядке часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемое параметром *count*
`public virtual void SetRange(int index, ICollection c)`	Заменяет часть вызывающей коллекции, начиная с элемента, указываемого по индексу index, элементами коллекции c
`public virtual void Sort()`	Сортирует вызывающую коллекцию по нарастающей
`public virtual void Sort(Icomparer comparer)`	Сортирует вызывающую коллекцию, используя для сравнения способ, определяемый параметром *comparer*. Если параметр *comparer* имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию
`public virtual void Sort(int index, int count, Icomparer comparer)`	Сортирует вызывающую коллекцию, используя для сравнения способ, определяемый параметром *comparer*. Сортировка начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Если параметр comparer имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию
`public static ArrayList Synchronized(ArrayList list)`	Возвращает синхронизированный вариант коллекции типа ArrayList, передаваемой в качестве параметра *list*
`public virtual object[] ToArray()`	Возвращает массив, содержащий копии элементов вызывающего объекта
`public virtual Array ToArray(Type type)`	Возвращает массив, содержащий копии элементов вызывающего объекта. Тип элементов этого массива определяется параметром *type*
`public virtual void TrimToSize()`	Устанавливает значение свойства Capacity равным значению свойства Count

В классе ArrayList поддерживается также ряд методов, оперирующих элементами коллекции в заданных пределах. Так, в одну коллекцию типа ArrayList можно вставить другую коллекцию, вызвав метод InsertRange(). Для удаления из коллекции элементов в заданных пределах достаточно вызвать метод RemoveRange(). А для

перезаписи элементов коллекции типа ArrayList в заданных пределах элементами из другой коллекции служит метод SetRange(). И наконец, элементы коллекции можно сортировать или искать в заданных пределах, а не во всей коллекции.

По умолчанию коллекция типа ArrayList не синхронизирована. Для получения синхронизированной оболочки, в которую заключается коллекция, вызывается метод Synchronized().

В классе ArrayList имеется также приведенное ниже свойство Capacity, помимо свойств, определенных в интерфейсах, которые в нем реализуются.

```
public virtual int Capacity { get; set; }
```

Свойство Capacity позволяет получать и устанавливать емкость вызывающей коллекции типа ArrayList. Емкость обозначает количество элементов, которые может содержать коллекция типа ArrayList до ее вынужденного расширения. Как упоминалось выше, коллекция типа ArrayList расширяется автоматически, и поэтому задавать ее емкость вручную необязательно. Но из соображений эффективности это иногда можно сделать, если количество элементов коллекции известно заранее. Благодаря этому исключаются издержки на выделение дополнительной памяти.

С другой стороны, если требуется сократить размер базового массива коллекции типа ArrayList, то для этой цели достаточно установить меньшее значение свойства Capacity. Но это значение не должно быть меньше значения свойства Count. Напомним, что свойство Count определено в интерфейсе ICollection и содержит количество объектов, хранящихся в коллекции на данный момент. Всякая попытка установить значение свойства Capacity меньше значения свойства Count приводит к генерированию исключения ArgumentOutOfRangeException. Поэтому для получения такого количества элементов коллекции типа ArrayList, которое содержится в ней на данный момент, следует установить значение свойства Capacity равным значению свойства Count. Для этой цели можно также вызвать метод TrimToSize().

В приведенном ниже примере программы демонстрируется применение класса ArrayList. В ней сначала создается коллекция типа ArrayList, а затем в эту коллекцию вводятся символы, после чего содержимое коллекции отображается. Некоторые элементы затем удаляются из коллекции, и ее содержимое отображается вновь. После этого в коллекцию вводятся дополнительные элементы, что вынуждает увеличить ее емкость. И наконец, содержимое элементов коллекции изменяется.

```
// Продемонстрировать применение класса ArrayList.

using System;
using System.Collections;

class ArrayListDemo {
  static void Main() {
    // Создать коллекцию в виде динамического массива.
    ArrayList al = new ArrayList();

    Console.WriteLine("Исходное количество элементов: " + al.Count);

    Console.WriteLine();

    Console.WriteLine("Добавить 6 элементов");
    // Добавить элементы в динамический массив.
    al.Add('C');
```

```
    al.Add('A');
    al.Add('E');
    al.Add('B');
    al.Add('D');
    al.Add('F');

    Console.WriteLine("Количество элементов: " + al.Count);

    // Отобразить содержимое динамического массива,
    // используя индексирование массива.
    Console.Write("Текущее содержимое: ");
    for(int i=0; i < al.Count; i++)
      Console.Write(al[i] + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Удалить 2 элемента");
    // Удалить элементы из динамического массива.
    al.Remove('F');
    al.Remove('A');

    Console.WriteLine("Количество элементов: " + al.Count);

    // Отобразить содержимое динамического массива, используя цикл foreach.
    Console.Write("Содержимое: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Добавить еще 20 элементов");
    // Добавить количество элементов, достаточное для
    // принудительного расширения массива.
    for(int i=0; i < 20; i++)
      al.Add((char)('a' + i));
    Console.WriteLine("Текущая емкость: " + al.Capacity);
    Console.WriteLine("Количество элементов после добавления 20 новых: " +
                      al.Count);
    Console.Write("Содержимое: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine("\n");

    // Изменить содержимое динамического массива,
    // используя индексирование массива.
    Console.WriteLine("Изменить три первых элемента");
    al[0] = 'X';
    al[1] = 'Y';
    al[2] = 'Z';
    Console.Write("Содержимое: ");
    foreach(char c in al)
      Console.Write(c + " ");
    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Исходное количество элементов: 0

Добавить 6 элементов
Количество элементов: 6
Текущее содержимое: C A E B D F

Удалить 2 элемента
Количество элементов: 4
Содержимое: C E B D

Добавить еще 20 элементов
Текущая емкость: 32
Количество элементов после добавления 20 новых: 24
Содержимое: C E B D a b c d e f g h i j k l m n o p q r s t

Изменить три первых элемента
Содержимое: X Y Z D a b c d e f g h i j k l m n o p q r s t
```

Сортировка и поиск в коллекции типа `ArrayList`

Коллекцию типа `ArrayList` можно отсортировать с помощью метода `Sort()`. В этом случае поиск в отсортированной коллекции с помощью метода `BinarySearch()` становится еще более эффективным. Применение обоих методов демонстрируется в приведенном ниже примере программы.

```
// Отсортировать коллекцию типа ArrayList и осуществить в ней поиск.

using System;
using System.Collections;

class SortSearchDemo {
  static void Main() {
    // Создать коллекцию в виде динамического массива.
    ArrayList al = new ArrayList();

    // Добавить элементы в динамический массив.
    al.Add(55);
    al.Add(43);
    al.Add(-4);
    al.Add(88);
    al.Add(3);
    al.Add(19);

    Console.Write("Исходное содержимое: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");

    // Отсортировать динамический массив.
    al.Sort();

    // Отобразить содержимое динамического массива, используя цикл foreach.
```

```
    Console.Write("Содержимое после сортировки: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine("\n");
    Console.WriteLine("Индекс элемента 43: " +
    al.BinarySearch(43));
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Исходное содержимое: 55 43 -4 88 3 19

Содержимое после сортировки: -4 3 19 43 55 88

Индекс элемента 43: 3
```

В одной и той же коллекции типа `ArrayList` могут храниться объекты любого типа. Тем не менее во время сортировки и поиска в ней эти объекты приходится сравнивать. Так, если бы список объектов в приведенном выше примере программы содержал символьную строку, то их сравнение привело бы к исключительной ситуации. Впрочем, для сравнения символьных строк и целых чисел можно создать специальные методы. О таких методах сравнения речь пойдет далее в этой главе.

Получение массива из коллекции типа `ArrayList`

В работе с коллекцией типа `ArrayList` иногда требуется получить из ее содержимого обычный массив. Этой цели служит метод `ToArray()`. Для преобразования коллекции в массив имеется несколько причин. Две из них таковы: потребность в ускорении обработки при выполнении некоторых операций и необходимость передавать массив методу, который не перегружается, чтобы принять коллекцию. Но независимо от конкретной причины коллекция типа `ArrayList` преобразуется в обычный массив довольно просто, как показано в приведенном ниже примере программы.

```
// Преобразовать коллекцию типа ArrayList в обычный массив.

using System;
using System.Collections;

class ArrayListToArray {
  static void Main() {
    ArrayList al = new ArrayList();

    // Добавить элементы в динамический массив.
    al.Add(1);
    al.Add(2);
    al.Add(3);
    al.Add(4);

    Console.Write("Содержимое: ");
    foreach(int i in al)
      Console.Write(i + " ");
    Console.WriteLine();

    // Получить массив.
```

```
    int[] ia = (int[]) al.ToArray(typeof(int));
    int sum = 0;

    // Просуммировать элементы массива.
    for(int i=0; i<ia.Length; i++)
      sum += ia[i];
    Console.WriteLine("Сумма равна: " + sum);
  }
}
```

Эта программа дает следующий результат.

```
Содержимое: 1 2 3 4
Сумма равна: 10
```

В начале этой программы создается коллекция целых чисел. Затем в ней вызывается метод `ToArray()` с указанием типа `int` получаемого массива. В итоге создается целочисленный массив. Но поскольку `Array` является типом, возвращаемым методом `ToArray()`, то содержимое получаемого в итоге массива должно быть приведено к типу `int[]`. (Напомним, что `Array` является базовым типом для всех массивов в C#.) И наконец, значения всех элементов массива суммируются.

Класс `Hashtable`

Класс `Hashtable` предназначен для создания коллекции, в которой для хранения ее элементов служит хеш-таблица. Как должно быть известно большинству читателей, информация сохраняется в *хеш-таблице* с помощью механизма, называемого *хешированием*. При хешировании для определения уникального значения, называемого *хеш-кодом*, используется информационное содержимое специального ключа. Полученный в итоге хеш-код служит в качестве индекса, по которому в таблице хранятся искомые данные, соответствующие заданному ключу. Преобразование ключа в хеш-код выполняется автоматически, и поэтому сам хеш-код вообще недоступен пользователю. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения операций поиска, извлечения и установки значений независимо от величины массивов данных. В классе `Hashtable` реализуются интерфейсы `IDictionary`, `ICollection`, `IEnumerable`, `ISerializable`, `IDeserializationCallback` и `ICloneable`.

В классе `Hashtable` определено немало конструкторов. Ниже приведены наиболее часто используемые конструкторы этого класса.

```
public Hashtable()
public Hashtable(IDictionary d)
public Hashtable(int capacity)
public Hashtable(int capacity,
                 float loadFactor)
```

В первой форме создается создаваемый по умолчанию объект класса `Hashtable`. Во второй форме создаваемый объект типа `Hashtable` инициализируется элементами из коллекции *d*. В третьей форме создаваемый объект типа `Hashtable` инициализируется, учитывая емкость коллекции, задаваемую параметром *capacity*. И в четвертой форме создаваемый объект типа `Hashtable` инициализируется, учитывая заданную емкость *capacity* и коэффициент заполнения *loadFactor*. Коэффициент заполнения, иногда еще называемый *коэффициентом загрузки*, должен находиться в пределах

от 0,1 до 1,0. Он определяет степень заполнения хеш-таблицы до увеличения ее размера. В частности, таблица расширяется, если количество элементов оказывается больше емкости таблицы, умноженной на коэффициент заполнения. В тех конструкторах, которые не принимают коэффициент заполнения в качестве параметра, этот коэффициент по умолчанию выбирается равным 1,0.

В классе `Hashtable` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов этого класса приведены в табл. 25.5. В частности, для того чтобы определить, содержится ли ключ в коллекции типа `Hashtable`, вызывается метод `ContainsKey()`. А для того чтобы выяснить, хранится ли в такой коллекции конкретное значение, вызывается метод `ContainsValue()`. Для перечисления содержимого коллекции типа `Hashtable` служит метод `GetEnumerator()`, возвращающий объект типа `IDictionaryEnumerator`. Напомним, что `IDictionaryEnumerator` — это перечислитель, используемый для перечисления содержимого коллекции, в которой хранятся пары "ключ–значение".

Таблица 25.5. Наиболее часто используемые методы, определенные в классе `Hashtable`

Метод	Описание
`public virtual bool ContainsKey(object key)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `Hashtable` содержится ключ *`key`*, а иначе — логическое значение `false`
`public virtual bool ContainsValue(object value)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `Hashtable` содержится значение *`value`*, а иначе — логическое значение `false`
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей коллекции типа `Hashtable` перечислитель типа `IDictionaryEnumerator`
`public static Hashtable Synchronized(Hashtable table)`	Возвращает синхронизированный вариант коллекции типа `Hashtable`, передаваемой в качестве параметра *`table`*

В классе `Hashtable` доступны также открытые свойства, определенные в тех интерфейсах, которые в нем реализуются. Особая роль принадлежит двум свойствам, `Keys` и `Values`, поскольку с их помощью можно получить ключи или значения из коллекции типа `Hashtable`. Эти свойства определяются в интерфейсе `IDictionary` следующим образом.

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

В классе `Hashtable` не поддерживаются упорядоченные коллекции, и поэтому ключи или значения получаются из коллекции в произвольном порядке. Кроме того, в классе `Hashtable` имеется защищенное свойство `EqualityComparer`. А два других свойства, `hcp` и `comparer`, считаются устаревшими.

Пары "ключ–значение" сохраняются в коллекции типа `Hashtable` в форме структуры типа `DictionaryEntry`, но чаще всего это делается без прямого вмешательства

со стороны пользователя, поскольку свойства и методы оперируют ключами и значениями по отдельности. Если, например, в коллекцию типа `Hashtable` добавляется элемент, то для этой цели вызывается метод `Add()`, принимающий два аргумента: ключ и значение.

Нужно, однако, иметь в виду, что сохранение порядка следования элементов в коллекции типа `Hashtable` не гарантируется. Дело в том, что процесс хеширования оказывается, как правило, непригодным для создания отсортированных таблиц.

Ниже приведен пример программы, в которой демонстрируется применение класса `Hashtable`.

```
// Продемонстрировать применение класса Hashtable.

using System;
using System.Collections;

class HashtableDemo {
  static void Main() {
    // Создать хеш-таблицу.
    Hashtable ht = new Hashtable();

    // Добавить элементы в таблицу.
    ht.Add("здание", "жилое помещение");
    ht.Add("автомашина", "транспортное средство");
    ht.Add("книга", "набор печатных слов");
    ht.Add("яблоко", "съедобный плод");

    // Добавить элементы с помощью индексатора.
    ht["трактор"] = "сельскохозяйственная машина";

    // Получить коллекцию ключей.
    ICollection c = ht.Keys;

    // Использовать ключи для получения значений.
    foreach(string str in c)
      Console.WriteLine(str + ": " + ht[str]);
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
здание: жилое помещение
книга: набор печатных слов
трактор: сельскохозяйственная машина
автомашина: транспортное средство
яблоко: съедобный плод
```

Как следует из приведенного выше результата, пары "ключ–значение" сохраняются в произвольном порядке. Обратите внимание на то, как получено и отображено содержимое хеш-таблицы `ht`. Сначала была получена коллекция ключей с помощью свойства `Keys`. Затем каждый ключ был использован для индексирования хеш-таблицы `ht` с целью извлечь из нее значение, соответствующее заданному ключу. Напомним, что в качестве индекса в данном случае использовался индексатор, определенный в интерфейсе `IDictionary` и реализованный в классе `Hashtable`.

Класс SortedList

Класс SortedList предназначен для создания коллекции, в которой пары "ключ–значение" хранятся в порядке, отсортированном по значению ключей. В классе SortedList реализуются интерфейсы IDictionary, ICollection, IEnumerable и ICloneable.

В классе SortedList определено несколько конструкторов, включая следующие.

```
public SortedList()
public SortedList(IDictionary d)
public SortedList(int initialCapacity)
public SortedList(IComparer comparer)
```

В первом конструкторе создается пустая коллекция, первоначальная емкость которой равна нулю. Во втором конструкторе создается пустая коллекция типа SortedList, которая инициализируется элементами из коллекции *d*. Ее первоначальная емкость равна количеству указанных элементов. В третьем конструкторе создается пустая коллекция типа SortedList, первоначальный размер которой определяет емкость, задаваемая параметром *initialCapacity*. Эта емкость соответствует размеру базового массива, используемого для хранения элементов коллекции. И в четвертой форме конструктора с помощью параметра *comparer* указывается способ, используемый для сравнения объектов по списку. В этой форме создается пустая коллекция, первоначальная емкость которой равна нулю.

При добавлении новых элементов в список емкость коллекции типа SortedList увеличивается автоматически по мере надобности. Так, если текущая емкость коллекции превышается, то она соответственно увеличивается. Преимущество указания емкости коллекции типа SortedList при ее создании заключается в снижении или полном исключении издержек на изменение размера коллекции. Разумеется, указывать емкость коллекции целесообразно лишь в том случае, если заранее известно, сколько элементов требуется хранить в ней.

В классе SortedList определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов этого класса перечислены в табл. 25.6. Так, если требуется определить, содержится ли ключ в коллекции типа SortedList, вызывается метод ContainsKey(). А если требуется выяснить, хранится ли конкретное значение в коллекции типа SortedList, вызывается метод ContainsValue(). Для перечисления содержимого коллекции типа SortedList служит метод GetEnumerator(), возвращающий объект типа IDictionaryEnumerator. Напомним, что IDictionaryEnumerator — это перечислитель, используемый для перечисления содержимого коллекции, в которой хранятся пары "ключ–значение". И наконец, для получения синхронизированной оболочки, в которую заключается коллекция типа SortedList, вызывается метод Synchronized().

Таблица 25.6. Наиболее часто используемые методы, определенные в классе SortedList

Метод	Описание
public virtual bool ContainsKey(object *key*)	Возвращает логическое значение true, если в вызывающей коллекции типа SortedList содержится ключ *key*, а иначе — логическое значение false

Окончание табл. 25.6

Метод	Описание
`public virtual bool ContainsValue(object value)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `SortedList` содержится значение *value*, а иначе — логическое значение `false`
`public virtual object GetByIndex(int index)`	Возвращает значение, указываемое по индексу *index*
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей коллекции типа `SortedList` перечислитель типа `IDictionaryEnumerator`
`public virtual object GetKey(int index)`	Возвращает значение ключа, указываемое по индексу *index*
`public virtual IList GetKeyList()`	Возвращает коллекцию типа `SortedList` с ключами, хранящимися в вызывающей коллекции типа `SortedList`
`public virtual IList GetValueList()`	Возвращает коллекцию типа `SortedList` со значениями, хранящимися в вызывающей коллекции типа `SortedList`
`public virtual int IndexOfKey(object key)`	Возвращает индекс ключа *key*. Если искомый ключ не обнаружен, возвращается значение `-1`
`public virtual int IndexOfValue(object value)`	Возвращает индекс первого вхождения значения *value* в вызывающей коллекции. Если искомое значение не обнаружено, возвращается значение `-1`
`public virtual void SetByIndex(int index, object value)`	Устанавливает значение по индексу *index* равным значению *value*
`public static SortedList Synchronized(SortedList list)`	Возвращает синхронизированный вариант коллекции типа `SortedList`, передаваемой в качестве параметра *list*
`public virtual void TrimToSize()`	Устанавливает значение свойства `Capacity` равным значению свойства `Count`

Ключ или значение можно получить разными способами. В частности, для получения значения по указанному индексу служит метод `GetByIndex()`, а для установки значения по указанному индексу — метод `SetByIndex()`. Для извлечения ключа по указанному индексу вызывается метод `GetKey()`, а для получения списка ключей по указанному индексу — метод `GetKeyList()`. Кроме того, для получения списка всех значений из коллекции служит метод `GetValueList()`. Для получения индекса ключа вызывается метод `IndexOfKey()`, а для получения индекса значения — метод `IndexOfValue()`. Безусловно, в классе `SortedList` также поддерживается индексатор, определяемый в интерфейсе `IDictionary` и позволяющий устанавливать и получать значение по заданному ключу.

В классе `SortedList` доступны также открытые свойства, определенные в тех интерфейсах, которые в нем реализуются. Как и в классе `Hashtable`, в данном классе особая роль принадлежит двум свойствам, `Keys` и `Values`, поскольку с их помощью можно получить доступную только для чтения коллекцию ключей или значений из

коллекции типа SortedList. Эти свойства определяются в интерфейсе IDictionary следующим образом.

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

Порядок следования ключей и значений отражает порядок их расположения в коллекции типа SortedList.

Аналогично коллекции типа Hashtable, пары "ключ–значение" сохраняются в коллекции типа SortedList в форме структуры типа DictionaryEntry, но, как правило, доступ к ключам и значениям осуществляется по отдельности с помощью методов и свойств, определенных в классе SortedList.

В приведенном ниже примере программы демонстрируется применение класса SortedList. Это переработанный и расширенный вариант предыдущего примера, демонстрировавшего применение класса Hashtable, вместо которого теперь используется класс SortedList. Глядя на результат выполнения этой программы, вы можете сами убедиться, что теперь список полученных значений оказывается отсортированным по заданному ключу.

```
// Продемонстрировать применение класса SortedList.

using System;
using System.Collections;

class SLDemo {
static void Main() {
    // Создать отсортированный список.
    SortedList sl = new SortedList();

    // Добавить элементы в список.
    sl.Add("здание", "жилое помещение");
    sl.Add("автомашина", "транспортное средство");
    sl.Add("книга", "набор печатных слов");
    sl.Add("яблоко", "съедобный плод");

    // Добавить элементы с помощью индексатора.
    sl["трактор"] = "сельскохозяйственная машина";

    // Получить коллекцию ключей.
    ICollection c = sl.Keys;

    // Использовать ключи для получения значений.
    Console.WriteLine("Содержимое списка по индексатору.");
    foreach(string str in c)
      Console.WriteLine(str + ": " + sl[str]);
    Console.WriteLine();

    // Отобразить список, используя целочисленные индексы.
    Console.WriteLine("Содержимое списка по целочисленным индексам.");
    for(int i=0; i < sl.Count; i++)
      Console.WriteLine(sl.GetByIndex(i));
    Console.WriteLine();

    // Показать целочисленные индексы элементов списка.
```

```
    Console.WriteLine("Целочисленные индексы элементов списка.");
    foreach(string str in c)
      Console.WriteLine(str + ": " + sl.IndexOfKey(str));
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Содержимое списка по индексатору.
автомашина: транспортное средство
здание: жилое помещение
книга: набор печатных слов
трактор: сельскохозяйственная машина
яблоко: съедобный плод

Содержимое списка по целочисленным индексам.
транспортное средство
жилое помещение
набор печатных слов
сельскохозяйственная машина
съедобный плод

Целочисленные индексы элементов списка.
автомашина: 0
здание: 1
книга: 2
трактор: 3
яблоко: 4
```

Класс Stack

Как должно быть известно большинству читателей, *стек* представляет собой список, действующий по принципу "первым пришел — последним обслужен". Этот принцип действия стека можно наглядно представить на примере горки тарелок, стоящих на столе. Первая тарелка, поставленная в эту горку, извлекается из нее последней. Стек относится к одним из самых важных структур данных в вычислительной технике. Он нередко применяется, среди прочего, в системном программном обеспечении, компиляторах, а также в программах отслеживания в обратном порядке на основе искусственного интеллекта

Класс коллекции, поддерживающий стек, носит название `Stack`. В нем реализуются интерфейсы `ICollection`, `IEnumerable` и `ICloneable`. Этот класс создает динамическую коллекцию, которая расширяется по мере потребности хранить в ней вводимые элементы. Всякий раз, когда требуется расширить такую коллекцию, ее емкость увеличивается вдвое.

В классе `Stack` определяются следующие конструкторы.

```
public Stack()
public Stack(int initialCapacity)
public Stack(ICollection col)
```

В первой форме конструктора создается пустой стек, во второй форме — пустой стек, первоначальный размер которого определяет первоначальная емкость, задаваемая параметром *`initialCapacity`*, и в третьей форме — стек, содержащий элементы указываемой коллекции *`col`*. Его первоначальная емкость равна количеству указанных элементов.

В классе `Stack` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов этого класса приведены в табл. 25.7. Эти методы обычно применяются следующим образом. Для того чтобы поместить объект на вершине стека, вызывается метод `Push()`. А для того чтобы извлечь и удалить объект из вершины стека, вызывается метод `Pop()`. Если же объект требуется только извлечь, но не удалить из вершины стека, то вызывается метод `Peek()`. А если вызвать метод `Pop()` или `Peek()`, когда вызывающий стек пуст, то сгенерируется исключение `InvalidOperationException`.

Таблица 25.7. Наиболее часто используемые методы, определенные в классе `Stack`

Метод	Описание
`public virtual void Clear()`	Устанавливает свойство `Count` равным нулю, очищая, по существу, стек
`public virtual bool Contains(object obj)`	Возвращает логическое значение `true`, если объект *obj* содержится в вызывающем стеке, а иначе — логическое значение `false`
`public virtual object Peek()`	Возвращает элемент, находящийся на вершине стека, но не удаляет его
`public virtual object Pop()`	Возвращает элемент, находящийся на вершине стека, удаляя его по ходу дела
`public virtual void Push(object obj)`	Помещает объект *obj* в стек
`public static Stack Synchronized(Stack stack)`	Возвращает синхронизированный вариант коллекции типа `Stack`, передаваемой в качестве параметра *stack*
`public virtual object[] ToArray()`	Возвращает массив, содержащий копии элементов вызывающего стека

В приведенном ниже примере программы создается стек, в который помещается несколько целых значений, а затем они извлекаются обратно из стека.

```
// Продемонстрировать применение класса Stack.

using System;
using System.Collections;

class StackDemo {
  static void ShowPush(Stack st, int a) {
    st.Push(a);
    Console.WriteLine("Поместить в стек: Push(" + a + ")");
     Console.Write("Содержимое стека: ");
     foreach(int i in st)
       Console.Write(i + " ");

     Console.WriteLine();
  }

  static void ShowPop(Stack st) {
    Console.Write("Извлечь из стека: Pop -> ");
    int a = (int) st.Pop();
```

```
    Console.WriteLine(a);

    Console.Write("Содержимое стека: ");
    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void Main() {
    Stack st = new Stack();

    foreach(int i in st)
      Console.Write(i + " ");

    Console.WriteLine();

    ShowPush(st, 22);
    ShowPush(st, 65);
    ShowPush(st, 91);
    ShowPop(st);
    ShowPop(st);
    ShowPop(st);

    try {
      ShowPop(st);
    } catch (InvalidOperationException) {
      Console.WriteLine("Стек пуст.");
    }
  }
}
```

Ниже приведен результат выполнения этой программы. Обратите внимание на то, как обрабатывается исключение `InvalidOperationException`, генерируемое при попытке извлечь элемент из пустого стека.

```
Поместить в стек: Push(22)
Содержимое стека: 22
Поместить в стек: Push(65)
Содержимое стека: 65 22
Поместить в стек: Push(91)
Содержимое стека: 91 65 22
Извлечь из стека: Pop -> 91
Содержимое стека: 65 22
Извлечь из стека: Pop -> 65
Содержимое стека: 22
Извлечь из стека: Pop -> 22
Содержимое стека:
Извлечь из стека: Pop -> Стек пуст.
```

Класс Queue

Еще одной распространенной структурой данных является *очередь*, действующая по принципу: первым пришел — первым обслужен. Это означает, что первым из очереди извлекается элемент, помещенный в нее первым. Очереди часто встречаются в

реальной жизни. Многим из нас нередко приходилось стоять в очередях к кассе в банке, магазине или столовой. В программировании очереди применяются для хранения таких элементов, как процессы, выполняющиеся в данный момент в системе, списки приостановленных транзакций в базе данных или пакеты данных, полученные по Интернету. Кроме того, очереди нередко применяются в области имитационного моделирования.

Класс коллекции, поддерживающий очередь, носит название `Queue`. В нем реализуются интерфейсы `ICollection`, `IEnumerable` и `ICloneable`. Этот класс создает динамическую коллекцию, которая расширяется, если в ней необходимо хранить вводимые элементы. Так, если в очереди требуется свободное место, ее размер увеличивается на коэффициент роста, который по умолчанию равен 2,0.

В классе `Queue` определяются приведенные ниже конструкторы.

```
public Queue()
public Queue (int capacity)
public Queue (int capacity, float growFactor)
public Queue (ICollection col)
```

В первой форме конструктора создается пустая очередь с выбираемыми по умолчанию емкостью и коэффициентом роста 2,0. Во второй форме создается пустая очередь, первоначальный размер которой определяет емкость, задаваемая параметром *capacity*, а коэффициент роста по умолчанию выбирается для нее равным 2,0. В третьей форме допускается указывать не только емкость (в качестве параметра *capacity*), но и коэффициент роста создаваемой очереди (в качестве параметра *growFactor* в пределах от 1,0 до 10,0). И в четвертой форме создается очередь, состоящая из элементов указываемой коллекции *col*. Ее первоначальная емкость равна количеству указанных элементов, а коэффициент роста по умолчанию выбирается для нее равным 2,0.

В классе `Queue` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов этого класса перечислены в табл. 25.8. Эти методы обычно применяются следующим образом. Для того чтобы поместить объект в очередь, вызывается метод `Enqueue()`. Если требуется извлечь и удалить первый объект из начала очереди, то вызывается метод `Dequeue()`. Если же требуется извлечь, но не удалять следующий объект из очереди, то вызывается метод `Peek()`. А если методы `Dequeue()` и `Peek()` вызываются, когда очередь пуста, то генерируется исключение `InvalidOperationException`.

Таблица 25.8. Наиболее часто используемые методы, определенные в классе Queue

Метод	Описание
`public virtual void Clear()`	Устанавливает свойство `Count` равным нулю, очищая, по существу, очередь
`public virtual bool Contains(object obj)`	Возвращает логическое значение `true`, если объект `obj` содержится в вызывающей очереди, а иначе — логическое значение `false`
`public virtual object Dequeue()`	Возвращает объект из начала вызывающей очереди. Возвращаемый объект удаляется из очереди
`public virtual void Enqueue(object obj)`	Добавляет объект `obj` в конец очереди

Окончание табл. 25.8

Метод	Описание
`public virtual object Peek()`	Возвращает объект из начала вызывающей очереди, но не удаляет его
`public static Queue Synchronized(Queue queue)`	Возвращает синхронизированный вариант коллекции типа `Queue`, передаваемой в качестве параметра *queue*
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов из вызывающей очереди
`public virtual void TrimToSize()`	Устанавливает значение свойства `Capacity` равным значению свойства `Count`

В приведенном ниже примере программы демонстрируется применение класса `Queue`.

```
// Продемонстрировать применение класса Queue.

using System;
using System.Collections;

class QueueDemo {
  static void ShowEnq(Queue q, int a) {
    q.Enqueue(a);
    Console.WriteLine("Поместить в очередь: Enqueue(" + a + ")");

    Console.Write("Содержимое очереди: ");
    foreach(int i in q)
     Console.Write(i + " ");

    Console.WriteLine();
  }

  static void ShowDeq(Queue q) {
    Console.Write("Извлечь из очереди: Dequeue -> ");
    int a = (int) q.Dequeue();
    Console.WriteLine(a);

    Console.Write("Содержимое очереди: ");
    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();
  }

  static void Main() {
    Queue q = new Queue();

    foreach(int i in q)
      Console.Write(i + " ");

    Console.WriteLine();
```

```
    ShowEnq(q, 22);
    ShowEnq(q, 65);
    ShowEnq(q, 91);
    ShowDeq(q);
    ShowDeq(q);
    ShowDeq(q);

    try {
      ShowDeq(q);
    } catch (InvalidOperationException) {
      Console.WriteLine("Очередь пуста.");
    }
  }
}
```

Эта программа дает следующий результат.

```
Поместить в очередь: Enqueue(22)
Содержимое очереди: 22
Поместить в очередь: Enqueue(65)
Содержимое очереди: 22 65
Поместить в очередь: Enqueue(91)
Содержимое очереди: 22 65 91
Извлечь из очереди: Dequeue -> 22
Содержимое очереди: 65 91
Извлечь из очереди: Dequeue -> 65
Содержимое очереди: 91
Извлечь из очереди: Dequeue -> 91
Содержимое очереди:
Извлечь из очереди: Dequeue -> Очередь пуста.
```

Хранение отдельных битов в классе коллекции `BitArray`

Класс `BitArray` служит для хранения отдельных битов в коллекции. А поскольку в коллекции этого класса хранятся биты, а не объекты, то своими возможностями он отличается от классов других коллекций. Тем не менее в классе `BitArray` реализуются интерфейсы `ICollection` и `IEnumerable` как основополагающие элементы поддержки всех типов коллекций. Кроме того, в классе `BitArray` реализуется интерфейс `ICloneable`.

В классе `BitArray` определено несколько конструкторов. Так, с помощью приведенного ниже конструктора можно сконструировать объект типа `BitArray` из массива логических значений.

```
public BitArray(bool[] values)
```

В данном случае каждый элемент массива *`values`* становится отдельным битом в коллекции. Это означает, что каждому элементу массива `values` соответствует отдельный бит в коллекции. Более того, порядок расположения элементов в массиве `values` сохраняется и в коллекции соответствующих им битов.

Коллекцию типа `BitArray` можно также составить из массива байтов, используя следующий конструктор.

```
public BitArray(byte[] bytes)
```

Здесь битами в коллекции становится уже целый их набор из массива *bytes*, причем элемент *bytes*[0] обозначает первые 8 битов, элемент *bytes*[1] — вторые 8 битов и т.д. Аналогично, коллекцию типа BitArray можно составить из массива целочисленных значений, используя приведенный ниже конструктор.

```
public BitArray(int[ ] values)
```

В данном случае элемент *values*[0] обозначает первые 32 бита, элемент *values*[1] — вторые 32 бита и т.д.

С помощью следующего конструктора можно составить коллекцию типа BitArray, указав ее конкретный размер:

```
public BitArray(int length)
```

где *length* обозначает количество битов в коллекции, которые инициализируются логическим значением false. В приведенном ниже конструкторе можно указать не только размер коллекции, но и первоначальное значение составляющих ее битов.

```
public BitArray(int length, bool defaultValue)
```

В данном случае все биты в коллекции инициализируются значением *defaultValue*, передаваемым конструктору в качестве параметра.

И наконец, новую коллекцию типа BitArray можно создать из уже существующей, используя следующий конструктор.

```
public BitArray(BitArray bits)
```

Вновь сконструированный объект будет содержать такое же количество битов, как и в указываемой коллекции *bits*, а в остальном это будут две совершенно разные коллекции.

Коллекции типа BitArray подлежат индексированию. По каждому индексу указывается отдельный бит в коллекции, причем нулевой индекс обозначает младший бит.

В классе BitArray определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Методы этого класса приведены в табл. 25.9. Обратите внимание на то, что в классе BitArray не поддерживается метод Synchronized(). Это означает, что для коллекций данного класса синхронизированная оболочка недоступна, а свойство IsSynchronized всегда имеет логическое значение false. Тем не менее для управления доступом к коллекции типа BitArray ее можно синхронизировать для объекта, предоставляемого упоминавшимся ранее свойством SyncRoot.

Таблица 25.9. Методы, определенные в классе BitArray

Метод	Описание
public BitArray And(BitArray *value*)	Выполняет операцию логического умножения И битов вызывающего объекта и коллекции *value*. Возвращает коллекцию типа BitArray, содержащую результат
public bool Get(int *index*)	Возвращает значение бита, указываемого по индексу *index*
public BitArray Not()	Выполняет операцию поразрядного логического отрицания НЕ битов вызывающей коллекции и возвращает коллекцию типа BitArray, содержащую результат

Окончание табл. 25.9

Метод	Описание
`public BitArray Or(BitArray value)`	Выполняет операцию логического сложения ИЛИ битов вызывающего объекта и коллекции *value*. Возвращает коллекцию типа `BitArray`, содержащую результат
`public void Set(int index, bool value)`	Устанавливает бит, указываемый по индексу *index*, равным значению *value*
`public void SetAll(bool value)`	Устанавливает все биты равными значению *value*
`public BitArray Xor(BitArray value)`	Выполняет логическую операцию исключающее ИЛИ над битами вызывающего объекта и коллекции *value*. Возвращает коллекцию типа `BitArray`, содержащую результат

В классе `BitArray` определяется также собственное свойство, помимо тех, что указаны в интерфейсах, которые в нем реализуются.

```
public int Length { get; set; }
```

Свойство `Length` позволяет установить или получить количество битов в коллекции. Следовательно, оно возвращает такое же значение, как и стандартное свойство `Count`, определяемое для всех коллекций. В отличие от свойства `Count`, свойство `Length` доступно не только для чтения, но и для записи, а значит, с его помощью можно изменить размер коллекции типа `BitArray`. Так, при сокращении коллекции типа `BitArray` лишние биты усекаются, начиная со старшего разряда. А при расширении коллекции типа `BitArray` дополнительные биты, имеющие логическое значение `false`, вводятся в коллекцию, начиная с того же старшего разряда.

Кроме того, в классе `BitArray` определяется следующий индексатор.

```
public bool this[int index] { get; set; }
```

С помощью этого индексатора можно получать или устанавливать значение элемента.

В приведенном ниже примере демонстрируется применение класса `BitArray`.

```
// Продемонстрировать применение класса BitArray.

using System;
using System.Collections;

class BADemo {
  public static void ShowBits(string rem,
                              BitArray bits) {
    Console.WriteLine(rem);
    for(int i=0; i < bits.Count; i++)
      Console.Write("{0, -6} ", bits[i]);
    Console.WriteLine("\n");
  }

  static void Main() {
    BitArray ba = new BitArray(8);
    byte[] b = { 67 };
    BitArray ba2 = new BitArray(b);
```

```
    ShowBits("Исходное содержимое коллекции ba:", ba);

    ba = ba.Not();

    ShowBits("Содержимое коллекции ba после логической операции NOT:", ba);

    ShowBits("Содержимое коллекции ba2:", ba2);
    BitArray ba3 = ba.Xor(ba2);

    ShowBits("Результат логической операции ba XOR ba2:", ba3);
  }
}
```

Эта программа дает следующий результат.

```
Исходное содержимое коллекции ba:
False  False  False  False  False  False  False  False
Содержимое коллекции ba после логической операции NOT:
True  True  True  True  True  True  True  True
Содержимое коллекции ba2:
True  True  False  False  False  False  True  False
Результат логической операции ba XOR ba2:
False  False  True  True  True  True  False  True
```

Специальные коллекции

В среде .NET Framework предусмотрен ряд специальных коллекций, оптимизированных для работы с данными конкретного типа или для их обработки особым образом. Классы этих необобщенных коллекций определены в пространстве имен `System.Collections.Specialized` и перечислены ниже.

Класс специальной коллекции	Описание
`CollectionsUtil`	Содержит фабричные методы для создания коллекций
`HybridDictionary`	Предназначен для коллекций, в которых для хранения небольшого количества пар "ключ-значение" используется класс `ListDictionary`. При превышении коллекцией определенного размера автоматически используется класс `Hashtable` для хранения ее элементов
`ListDictionary`	Предназначен для коллекций, в которых для хранения пар "ключ-значение" используется связный список. Такие коллекции рекомендуются только для хранения небольшого количества элементов
`NameValueCollection`	Предназначен для отсортированных коллекций, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу `string`
`OrderedDictionary`	Предназначен для коллекций, в которых хранятся индексируемые пары "ключ-значение"
`StringCollection`	Предназначен для коллекций, оптимизированных для хранения символьных строк
`StringDictionary`	Предназначен для хеш-таблиц, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу `string`

Кроме того, в пространстве имен `System.Collections` определены три базовых абстрактных класса: `CollectionBase`, `ReadOnlyCollectionBase` и `DictionaryBase`. Эти классы могут наследоваться и служить в качестве отправной точки для разработки собственных специальных коллекций.

Обобщенные коллекции

Благодаря внедрению обобщений прикладной интерфейс Collections API значительно расширился, в результате чего количество классов коллекций и интерфейсов удвоилось. Обобщенные коллекции объявляются в пространстве имен `System.Collections.Generic`. Как правило, классы обобщенных коллекций являются не более чем обобщенными эквивалентами рассматривавшихся ранее классов необобщенных коллекций, хотя это соответствие не является взаимно однозначным. Например, в классе обобщенной коллекции `LinkedList` реализуется двунаправленный список, тогда как в необобщенном эквиваленте его не существует. В некоторых случаях одни и те же функции существуют параллельно в классах обобщенных и необобщенных коллекций, хотя и под разными именами. Так, обобщенный вариант класса `ArrayList` называется `List`, а обобщенный вариант класса `HashTable` — `Dictionary`. Кроме того, конкретное содержимое различных интерфейсов и классов реорганизуется с минимальными изменениями для переноса некоторых функций из одного интерфейса в другой. Но в целом, имея ясное представление о необобщенных коллекциях, можно без особого труда научиться применять и обобщенные коллекции.

Как правило, обобщенные коллекции действуют по тому же принципу, что и необобщенные, за исключением того, что обобщенные коллекции типизированы. Это означает, что в обобщенной коллекции можно хранить только те элементы, которые совместимы по типу с ее аргументом. Так, если требуется коллекция для хранения несвязанных друг с другом разнотипных данных, то для этой цели следует использовать классы необобщенных коллекций. А во всех остальных случаях, когда в коллекции должны храниться объекты только одного типа, выбор рекомендуется останавливать на классах обобщенных коллекций.

Обобщенные коллекции определяются в ряде интерфейсов и классов, реализующих эти интерфейсы. Все они описываются далее по порядку.

Интерфейсы обобщенных коллекций

В пространстве имен `System.Collections.Generic` определен целый ряд интерфейсов обобщенных коллекций, имеющих соответствующие аналоги среди интерфейсов необобщенных коллекций. Все эти интерфейсы сведены в табл. 25.10.

Таблица 25.10. Интерфейсы обобщенных коллекций

Интерфейс	Описание
`ICollection<T>`	Определяет основополагающие свойства обобщенных коллекций
`IComparer<T>`	Определяет обобщенный метод `Compare()` для сравнения объектов, хранящихся в коллекции
`IDictionary<Tkey, TValue>`	Определяет обобщенную коллекцию, состоящую из пар "ключ-значение"

Окончание табл. 25.10

Интерфейс	Описание
`IEnumerable<T>`	Определяет обобщенный метод `GetEnumerator()`, предоставляющий перечислитель для любого класса коллекции
`Enumerator<T>`	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
`IEqualityComparer<T>`	Сравнивает два объекта на предмет равенства
`IList<T>`	Определяет обобщенную коллекцию, доступ к которой можно получить с помощью индексатора

Интерфейс `ICollection<T>`

В интерфейсе `ICollection<T>` определен ряд свойств, которые являются общими для всех обобщенных коллекций. Интерфейс `ICollection<T>` является обобщенным вариантом необобщенного интерфейса `ICollection`, хотя между ними имеются некоторые отличия.

Итак, в интерфейсе `ICollection<T>` определены следующие свойства.

```
int Count { get; }
bool IsReadOnly { get; }
```

Свойство `Count` содержит ряд элементов, хранящихся в данный момент в коллекции. А свойство `IsReadOnly` имеет логическое значение `true`, если коллекция доступна только для чтения. Если же коллекция доступна как для чтения, так и для записи, то данное свойство имеет логическое значение `false`.

Кроме того, в интерфейсе `ICollection<T>` определены перечисленные ниже методы. Обратите внимание на то, что в этом обобщенном интерфейсе определено несколько большее количество методов, чем в его необобщенном аналоге.

Метод	Описание
`void Add(T item)`	Добавляет элемент `item` в вызывающую коллекцию. Генерирует исключение `NotSupportedException`, если коллекция доступна только для чтения
`void Clear()`	Удаляет все элементы из вызывающей коллекции
`bool Contains(T item)`	Возвращает логическое значение `true`, если вызывающая коллекция содержит элемент `item`, а иначе — логическое значение `false`
`void CopyTo(T[] array, int arrayIndex)`	Копирует содержимое вызывающей коллекции в массив `array`, начиная с элемента, указываемого по индексу `arrayIndex`
`void Remove(T item)`	Удаляет первое вхождение элемента `item` в вызывающей коллекции. Возвращает логическое значение `true`, если элемент `item` удален. А если этот элемент не найден в вызывающей коллекции, то возвращается логическое значение `false`

Некоторые из перечисленных выше методов генерируют исключение `NotSupportedException`, если коллекция доступна только для чтения.

А поскольку интерфейс ICollection<T> наследует от интерфейсов IEnumerable и IEnumerable<T>, то он включает в себя также обобщенную и необобщенную формы метода GetEnumerator().

Благодаря тому что в интерфейсе ICollection<T> реализуется интерфейс IEnumerable<T>, в нем поддерживаются также методы расширения, определенные в классе Enumerable. Несмотря на то что методы расширения предназначены главным образом для поддержки LINQ, им можно найти и другое применение, в том числе и в коллекциях.

Интерфейс IList<T>

В интерфейсе IList<T> определяется такое поведение обобщенной коллекции, которое позволяет осуществлять доступ к ее элементам по индексу с отсчетом от нуля. Этот интерфейс наследует от интерфейсов IEnumerable, IEnumerable<T> и ICollection<T> и поэтому является обобщенным вариантом необобщенного интерфейса IList. Методы, определенные в интерфейсе IList<T>, перечислены в табл. 25.11. В двух из этих методов предусматривается модификация коллекции. Если же коллекция доступна только для чтения или имеет фиксированный размер, то методы Insert() и RemoveAt() генерируют исключение NotSupportedException.

Таблица 25.11. Методы, определенные в интерфейсе IList<T>

Метод	Описание
`int IndexOf(T item)`	Возвращает индекс первого вхождения элемента *item* в вызывающей коллекции. Если элемент *item* не обнаружен, то метод возвращает значение -1
`void Insert(int index, T item)`	Вставляет в вызывающую коллекцию элемент *item* по индексу *index*
`void RemoveAt(int index)`	Удаляет из вызывающей коллекции элемент, расположенный по указанному индексу *index*

Кроме того, в интерфейсе IList<T> определяется индексатор

```
T this[int index] { get; set; }
```

который устанавливает или возвращает значение элемента коллекции по указанному индексу *index*.

Интерфейс IDictionary<TKey, TValue>

В интерфейсе IDictionary<TKey, TValue> определяется такое поведение обобщенной коллекции, которое позволяет преобразовать уникальные ключи в соответствующие значения. Это означает, что в данном интерфейсе определяется коллекция, в которой хранятся пары "ключ–значение". Интерфейс IDictionary<TKey, TValue> наследует от интерфейсов IEnumerable, IEnumerable<KeyValuePair<TKey, TValue>> и ICollection<KeyValuePair<TKey, TValue>> и поэтому является обобщенным вариантом необобщенного интерфейса IDictionary. Методы, объявленные в интерфейсе IDictionary<TKey, TValue>, приведены в табл. 25.12. Все эти методы генерируют исключение ArgumentNullException при попытке указать пустой ключ.

Таблица 25.12. Методы, определенные в интерфейсе `IDictionary<TKey, TValue>`

Метод	Описание
`void Add(TKey key, TValue value)`	Добавляет в вызывающую коллекцию пару "ключ-значение", определяемую параметрами *key* и *value*. Генерирует исключение `ArgumentException`, если ключ *key* уже находится в коллекции
`bool Contains(TKey key)`	Возвращает логическое значение `true`, если вызывающая коллекция содержит элемент *key* в качестве ключа, а иначе — логическое значение `false`
`bool Remove(TKey key)`	Удаляет из коллекции элемент, ключ которого равен значению *key*
`bool TryGetValue(TKey key, out TValue value)`	Предпринимает попытку извлечь значение из коллекции по указанному ключу *key* и присвоить это значение переменной *value*. При удачном исходе операции возвращается логическое значение `true`, а иначе — логическое значение `false`. Если ключ *key* не найден, переменной *value* присваивается значение, выбираемое по умолчанию

Кроме того, в интерфейсе `IDictionary<TKey, TValue>` определены перечисленные ниже свойства.

Свойство	Описание
`ICollection Keys<TKey> { get; }`	Получает коллекцию ключей
`ICollection Values<TValue> { get; }`	Получает коллекцию значений

Следует иметь в виду, что ключи и значения, содержащиеся в коллекции, доступны отдельными списками с помощью свойств `Keys` и `Values`.

И наконец, в интерфейсе `IDictionary<TKey, TValue>` определяется следующий индексатор.

```
TValue this[TKey key] { get; set; }
```

Этот индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. Следует, однако, иметь в виду, что в качестве индекса в данном случае служит ключ элемента, а не сам индекс.

Интерфейсы `IEnumerable<T>` и `IEnumerator<T>`

Интерфейсы `IEnumerable<T>` и `IEnumerator<T>` являются обобщенными эквивалентами рассмотренных ранее необобщенных интерфейсов `IEnumerable` и `IEnumerator`. В них объявляются аналогичные методы и свойства, да и действуют они по тому же принципу. Разумеется, обобщенные интерфейсы оперируют данными только того типа, который указывается в аргументе типа.

В интерфейсе `IEnumerable<T>` метод `GetEnumerator()` объявляется следующим образом.

```
IEnumerator<T> GetEnumerator()
```

Этот метод возвращает перечислитель типа `T` для коллекции. А это означает, что он возвращает типизированный перечислитель.

Кроме того, в интерфейсе IEnumerable<T> определяются два таких же метода, как и в необобщенном его варианте: MoveNext() и Reset(). В этом интерфейсе объявляется также обобщенный вариант свойства Current.

```
T Current { get; }
```

Это свойство возвращает ссылку типа T на следующий объект. А это означает, что обобщенный вариант свойства Current является типизированным.

Но между интерфейсами IEnumerator и IEnumerator<T> имеется одно важное различие: интерфейс IEnumerator<T> наследует от интерфейса IDisposable, тогда как интерфейс IEnumerator не наследует от него. В интерфейсе IDisposable определяется метод Dispose(), который служит для освобождения неуправляемых ресурсов.

ПРИМЕЧАНИЕ

В интерфейсе IEnumerable<T> реализуется также необобщенный интерфейс IEnumerable. Это означает, что в нем поддерживается необобщенный вариант метода GetEnumerator(). Кроме того, в интерфейсе IEnumerable<T> реализуется необобщенный интерфейс IEnumerator, а следовательно, в нем поддерживаются необобщенные варианты свойства Current.

Интерфейс IComparer<T>

Интерфейс IComparer<T> является обобщенным вариантом рассмотренного ранее интерфейса IComparer. Главное отличие между ними заключается в том, что интерфейс IComparer<T> обеспечивает типовую безопасность. В нем обобщенный вариант метода Compare() объявляется следующим образом.

```
int Compare(T x, T y)
```

В этом методе сравниваются объекты *x* и *y*. Он возвращает положительное значение, если значение объекта *x* больше, чем у объекта *y*; отрицательное — если значение объекта *x* меньше, чем у объекта *y*; и нулевое значение — если сравниваемые значения равны.

Интерфейс IEqualityComparer<T>

Интерфейс IEqualityComparer<T> полностью соответствует своему необобщенному аналогу EqualityComparer. В нем определяются два следующих метода.

```
bool Equals(T x, T y)
int GetHashCode(T obj)
```

Метод Equals() должен возвратить логическое значение true, если значения объектов *x* и *y* равны. А метод GetHashCode() возвращает хеш-код для объекта *obj*. Если два сравниваемых объекта равны, то их хеш-коды также должны быть одинаковы.

Интерфейс ISet<T>

Интерфейс ISet<T> был добавлен в версию 4.0 среды .NET Framework. Он определяет поведение обобщенной коллекции, реализующей ряд уникальных элементов. Этот интерфейс наследует от интерфейсов IEnumerable, IEnumerable<T>, а также ICollection<T>. В интерфейсе ISet<T> определен ряд методов, перечисленных

в табл. 25.13. Обратите внимание на то, что параметры этих методов указываются как относящиеся к типу `IEnumerable<T>`. Это означает, что в качестве второго аргумента методу можно передать нечто, отличающееся от объектов типа `ISet<T>`. Но чаще всего оба аргумента оказываются экземплярами объектов типа `ISet<T>`.

Таблица 25.13. Методы, определенные в интерфейсе `ISet<T>`

Метод	Описание
`void ExceptWith(Ienumerable<T> other)`	Удаляет из вызывающего множества те элементы, которые содержатся в другом множестве *other*
`void IntersectWith(IEnumerable<T> other)`	После вызова этого метода вызывающее множество содержит пересечение своих элементов с элементами другого множества *other*
`bool IsProperSubsetOf(IEnumerable<T> other)`	Возвращает логическое значение `true`, если вызывающее множество является правильным подмножеством другого множества *other*, а иначе — логическое значение `false`
`bool IsProperSupersetOf(IEnumerable<T> other)`	Возвращает логическое значение `true`, если вызывающее множество является правильным надмножеством другого множества *other*, а иначе — логическое значение `false`
`bool IsSubsetOf(IEnumerable<T> other)`	Возвращает логическое значение `true`, если вызывающее множество является подмножеством другого множества *other*, а иначе — логическое значение `false`
`bool IsSupersetOf(IEnumerable<T> other)`	Возвращает логическое значение `true`, если вызывающее множество является надмножеством другого множества *other*, а иначе — логическое значение `false`
`bool Overlaps(IEnumerable<T> other)`	Возвращает логическое значение `true`, если вызывающее множество и другое множество *other* содержат хотя бы один общий элемент, а иначе — логическое значение `false`
`bool SetEquals(IEnumerable<T> other)`	Возвращает логическое значение `true`, если все элементы вызывающего множества и другого множества *other* оказываются общими, а иначе — логическое значение `false`. Порядок расположения элементов не имеет значения, а дублирующиеся элементы во другом множестве *other* игнорируются
`void SymmetricExceptWith (IEnumerable<T> other)`	После вызова этого метода вызывающее множество будет содержать симметрическую разность своих элементов и элементов другого множества *other*
`void UnionWith(IEnumerable<T> other)`	После вызова этого метода вызывающее множество будет содержать объединение своих элементов и элементов другого множества *other*

Структура KeyValuePair<TKey, TValue>

В пространстве имен System.Collections.Generic определена структура KeyValuePair<TKey, TValue>. Она служит для хранения ключа и его значения и применяется в классах обобщенных коллекций, в которых хранятся пары "ключ–значение", как, например, в классе Dictionary<TKey, TValue>. В этой структуре определяются два следующих свойства.

```
public TKey Key { get; };
public TValue Value { get; };
```

В этих свойствах хранятся ключ и значение соответствующего элемента коллекции. Для построения объекта типа KeyValuePair<TKey, TValue> служит конструктор:

```
public KeyValuePair(TKey key, TValue value)
```

где *key* обозначает ключ, а *value* — значение.

Классы обобщенных коллекций

Как упоминалось ранее, классы обобщенных коллекций по большей части соответствуют своим необобщенным аналогам, хотя в некоторых случаях они носят другие имена. Отличаются они также своей организацией и функциональными возможностями. Классы обобщенных коллекций определяются в пространстве имен System.Collections.Generic. В табл. 25.14 приведены классы, рассматриваемые в этой главе. Эти классы составляют основу обобщенных коллекций.

Таблица 25.14. Основные классы обобщенных коллекций

Класс	Описание
Dictionary<Tkey, TValue>	Сохраняет пары "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс Hashtable
HashSet<T>	Сохраняет ряд уникальных значений, используя хеш-таблицу
LinkedList<T>	Сохраняет элементы в двунаправленном списке
List<T>	Создает динамический массив. Обеспечивает такие же функциональные возможности, как и необобщенный класс ArrayList
Queue<T>	Создает очередь. Обеспечивает такие же функциональные возможности, как и необобщенный класс Queue
SortedDictionary<TKey, TValue>	Создает отсортированный список из пар "ключ-значение"
SortedList<TKey, TValue>	Создает отсортированный список из пар "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс SortedList
SortedSet<T>	Создает отсортированное множество
Stack<T>	Создает стек. Обеспечивает такие же функциональные возможности, как и необобщенный класс Stack

ПРИМЕЧАНИЕ

В пространстве имен `System.Collections.Generic` находятся также следующие классы: класс `SynchronizedCollection<T>` синхронизированной коллекции на основе класса `IList<T>`; класс `SynchronizedReadOnlyCollection<T>`, доступной только для чтения синхронизированной коллекции на основе класса `IList<T>`; абстрактный класс `SynchronizedKeyCollection<K, T>`, служащий в качестве базового для класса коллекции `System.ServiceModel.UriSchemeKeyedCollection`; а также класс `KeyedByTypeCollection<T>` коллекции, в которой в качестве ключей используются отдельные типы данных.

Класс `List<T>`

В классе `List<T>` реализуется обобщенный динамический массив. Он ничем принципиально не отличается от класса необобщенной коллекции `ArrayList`. В этом классе реализуются интерфейсы `ICollection`, `ICollection<T>`, `IList`, `IList<T>`, `IEnumerable` и `IEnumerable<T>`. У класса `List<T>` имеются следующие конструкторы.

```
public List()
public List(IEnumerable<T> collection)
public List(int capacity)
```

Первый конструктор создает пустую коллекцию класса `List` с выбираемой по умолчанию первоначальной емкостью. Второй конструктор создает коллекцию типа `List` с количеством инициализируемых элементов, которое определяется параметром *`collection`* и равно первоначальной емкости массива. Третий конструктор создает коллекцию типа `List`, имеющую первоначальную емкость, задаваемую параметром *`capacity`*. В данном случае емкость обозначает размер базового массива, используемого для хранения элементов коллекции. Емкость коллекции, создаваемой в виде динамического массива, может увеличиваться автоматически по мере добавления в нее элементов.

В классе `List<T>` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов этого класса перечислены в табл. 25.15.

Таблица 25.15. Наиболее часто используемые методы, определенные в классе `List<T>`

Метод	Описание
`public virtual void AddRange(Icollection collection)`	Добавляет элементы из коллекции *`collection`* в конец вызывающей коллекции типа `ArrayList`
`public virtual int BinarySearch(T item)`	Выполняет поиск в вызывающей коллекции значения, задаваемого параметром *`item`*. Возвращает индекс совпавшего элемента. Если искомое значение не найдено, возвращается отрицательное значение. Вызывающий список должен быть отсортирован

Продолжение табл. 25.15

Метод	Описание
`public int BinarySearch(T item, IComparer<T> comparer)`	Выполняет поиск в вызывающей коллекции значения, задаваемого параметром *item*, используя для сравнения указанный способ, определяемый параметром *comparer*. Возвращает индекс совпавшего элемента. Если искомое значение не найдено, возвращается отрицательное значение. Вызывающий список должен быть отсортирован
`public int BinarySearch(int index, int count, T item, IComparer<T> comparer)`	Выполняет поиск в вызывающей коллекции значения, задаваемого параметром *item*, используя для сравнения указанный способ, определяемый параметром *comparer*. Поиск начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Метод возвращает индекс совпавшего элемента. Если искомое значение не найдено, возвращается отрицательное значение. Вызывающий список должен быть отсортирован
`public List<T> GetRange(int index, int count)`	Возвращает часть вызывающей коллекции. Часть возвращаемой коллекции начинается с элемента, указываемого по индексу *index*, и включает количество элементов, задаваемое параметром *count*. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект
`public int IndexOf(T item)`	Возвращает индекс первого вхождения элемента *item* в вызывающей коллекции. Если искомый элемент не обнаружен, возвращается значение -1
`public void InsertRange(int index, IEnumerable<T> collection)`	Вставляет элементы коллекции *collection* в вызывающую коллекцию, начиная с элемента, указываемого по индексу *index*
`public int LastIndexOf(T item)`	Возвращает индекс последнего вхождения элемента *item* в вызывающей коллекции. Если искомый элемент не обнаружен, возвращается значение -1
`public void RemoveRange(int index, int count)`	Удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемое параметром *count*
`public void Reverse()`	Располагает элементы вызывающей коллекции в обратном порядке
`public void Reverse(int index, int count)`	Располагает в обратном порядке часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемое параметром *count*
`public void Sort()`	Сортирует вызывающую коллекцию по нарастающей

Окончание табл. 25.15

Метод	Описание
`public void Sort(IComparer<T> comparer)`	Сортирует вызывающую коллекцию, используя для сравнения способ, задаваемый параметром *comparer*. Если параметр *comparer* имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию
`public void Sort(Comparison<T> comparison)`	Сортирует вызывающую коллекцию, используя для сравнения указанный делегат
`public void Sort(int index, int count, IComparer<T> comparer)`	Сортирует вызывающую коллекцию, используя для сравнения способ, задаваемый параметром *comparer*. Сортировка начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Если параметр *comparer* имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию
`public T[] ToArray()`	Возвращает массив, содержащий копии элементов вызывающего объекта
`public void TrimExcess()`	Сокращает емкость вызывающей коллекции таким образом, чтобы она не превышала 10% от количества элементов, хранящихся в ней на данный момент

В классе `List<T>` определяется также собственное свойство `Capacity`, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Это свойство объявляется следующим образом.

```
public int Capacity { get; set; }
```

Свойство `Capacity` позволяет установить и получить емкость вызывающей коллекции в качестве динамического массива. Эта емкость равна количеству элементов, которые может содержать коллекция до ее вынужденного расширения. Такая коллекция расширяется автоматически, и поэтому задавать ее емкость вручную необязательно. Но из соображений эффективности это иногда можно сделать, если заранее известно количество элементов коллекции. Благодаря этому исключаются издержки на выделение дополнительной памяти.

В классе `List<T>` реализуется также приведенный ниже индексатор, определенный в интерфейсе `IList<T>`.

```
public T this[int index] { get; set; }
```

С помощью этого индексатора устанавливается и получается значение элемента коллекции, указываемое по индексу *index*.

В приведенном ниже примере программы демонстрируется применение класса `List<T>`. Это измененный вариант примера, демонстрировавшего ранее класс `ArrayList`. Единственное изменение, которое потребовалось для этого, заключалось в замене класса `ArrayList` классом `List`, а также в использовании параметров обобщенного типа.

```
// Продемонстрировать применение класса List<T>.

using System;
using System.Collections.Generic;

class GenListDemo {
  static void Main() {
    // Создать коллекцию в виде динамического массива.
    List<char> lst = new List<char>();

      Console.WriteLine("Исходное количество элементов: " + lst.Count);

    Console.WriteLine();

    Console.WriteLine("Добавить 6 элементов");
    // Добавить элементы в динамический массив.
    lst.Add('C');
    lst.Add('A');
    lst.Add('E');
    lst.Add('B');
    lst.Add('D');
    lst.Add('F');

    Console.WriteLine("Количество элементов: " + lst.Count);

    // Отобразить содержимое динамического массива,
    // используя индексирование массива.
    Console.Write("Текущее содержимое: ");
    for(int i=0; i < lst.Count; i++)
      Console.Write(lst[i] + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Удалить 2 элемента ");
    // Удалить элементы из динамического массива.
    lst.Remove('F');
    lst.Remove('A');

    Console.WriteLine("Количество элементов: " + lst.Count);

    // Отобразить содержимое динамического массива, используя цикл foreach.
    Console.Write("Содержимое: ");
    foreach(char c in lst)
      Console.Write(c + " ");
    Console.WriteLine("\n");

    Console.WriteLine("Добавить еще 20 элементов");
    // Добавить количество элементов, достаточное для
    // принудительного расширения массива.
    for(int i=0; i < 20; i++)
      lst.Add((char)('a' + i));
    Console.WriteLine("Текущая емкость: " + lst.Capacity);
    Console.WriteLine("Количество элементов после добавления 20 новых: " +
                      lst.Count);
    Console.Write("Содержимое: ");
```

```
    foreach(char c in lst)
      Console.Write(c + " ");
    Console.WriteLine("\n");

    // Изменить содержимое динамического массива,
    // используя индексирование массива.
    Console.WriteLine("Изменить три первых элемента");
    lst[0] = 'X';
    lst[1] = 'Y';
    lst[2] = 'Z';

    Console.Write("Содержимое: ");
    foreach(char c in lst)
      Console.Write(c + " ");
    Console.WriteLine();

    // Следующая строка кода недопустима из-за
    // нарушения безопасности обобщенного типа.
// lst.Add(99); // Ошибка, поскольку это не тип char!
  }
}
```

Эта версия программы дает такой же результат, как и предыдущая.

```
Исходное количество элементов: 0

Добавить 6 элементов
Количество элементов: 6
Текущее содержимое: C A E B D F

Удалить 2 элемента
Количество элементов: 4
Содержимое: C E B D

Добавить еще 20 элементов
Текущая емкость: 32
Количество элементов после добавления 20 новых: 24
Содержимое: C E B D a b c d e f g h i j k l m n o p q r s t

Изменить три первых элемента
Содержимое: X Y Z D a b c d e f g h i j k l m n o p q r s t
```

Класс LinkedList<T>

В классе `LinkedList<T>` создается коллекция в виде обобщенного двунаправленного списка. В этом классе реализуются интерфейсы `ICollection`, `ICollection<T>`, `IEnumerable`, `IEnumerable<T>`, `ISerializable` и `IDeserializationCallback`. В двух последних интерфейсах поддерживается сериализация списка. В классе `LinkedList<T>` определяются два приведенных ниже открытых конструктора.

```
public LinkedList()
public LinkedList(IEnumerable<T> collection)
```

В первом конструкторе создается пустой связный список, а во втором конструкторе — список, инициализируемый элементами из коллекции *collection*.

Как и в большинстве других реализаций связных списков, в классе LinkedList<T> инкапсулируются значения, хранящиеся в *узлах* списка, где находятся также ссылки на предыдущие и последующие элементы списка. Эти узлы представляют собой объекты класса LinkedListNode<T>. В классе LinkedListNode<T> предоставляются четыре следующих свойства.

```
public LinkedListNode<T> Next { get; }
public LinkedListNode<T> Previous { get; }
public LinkedList<T> List { get; }
public T Value { get; set; }
```

С помощью свойств Next и Previous получаются ссылки на предыдущий и последующий узлы списка соответственно, что дает возможность обходить список в обоих направлениях. Если же предыдущий или последующий узел отсутствует, то возвращается пустая ссылка. Для получения ссылки на сам список служит свойство List. А с помощью свойства Value можно устанавливать и получать значение, находящееся в узле списка.

В классе LinkedList<T> определяется немало методов. В табл. 25.16 приведены наиболее часто используемые методы данного класса. Кроме того, в классе LinkedList<T> определяются собственные свойства, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Эти свойства приведены ниже.

```
public LinkedListNode<T> First { get; }
public LinkedListNode<T> Last { get; }
```

С помощью свойства First получается первый узел в списке, а с помощью свойства Last — последний узел в списке.

Таблица 25.16. Наиболее часто используемые методы, определенные в классе LinkedList<T>

Метод	Описание
public LinkedListNode<T> AddAfter(LinkedListNode<T> *node*, T *value*)	Добавляет в список узел со значением *value* непосредственно после указанного узла *node*. Указываемый узел *node* не должен быть пустым (null). Метод возвращает ссылку на узел, содержащий значение *value*
public void AddAfter(LinkedListNode<T> *node*, LinkedListNode<T> *newNode*)	Добавляет в список новый узел *newNode* непосредственно после указанного узла *node*. Указываемый узел *node* не должен быть пустым (null). Если узел *node* отсутствует в списке или если новый узел *newNode* является частью другого списка, то генерируется исключение InvalidOperationException
public LinkedListNode<T> AddBefore(LinkedListNode<T> *node*, T *value*)	Добавляет в список узел со значением *value* непосредственно перед указанным узлом *node*. Указываемый узел *node* не должен быть пустым (null). Метод возвращает ссылку на узел, содержащий значение *value*

Окончание табл. 25.16

Метод	Описание
`public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)`	Добавляет в список новый узел `newNode` непосредственно перед указанным узлом `node`. Указываемый узел `node` не должен быть пустым (`null`). Если узел `node` отсутствует в списке или если новый узел `newNode` является частью другого списка, то генерируется исключение `InvalidOperationException`
`public LinkedList<T> AddFirst(T value)`	Добавляет узел со значением `value` в начало списка. Метод возвращает ссылку на узел, содержащий значение `value`
`public void AddFirst(LinkedListNode node)`	Добавляет узел `node` в начало списка. Если узел `node` является частью другого списка, то генерируется исключение `InvalidOperationException`
`public LinkedList<T> AddLast(T value)`	Добавляет узел со значением `value` в конец списка. Метод возвращает ссылку на узел, содержащий значение `value`
`public void AddLast(LinkedListNode node)`	Добавляет узел `node` в конец списка. Если узел `node` является частью другого списка, то генерируется исключение `InvalidOperationException`
`public LinkedList<T> Find(T value)`	Возвращает ссылку на первый узел в списке, имеющий значение `value`. Если искомое значение `value` отсутствует в списке, то возвращается пустое значение
`public LinkedList<T> FindLast(T value)`	Возвращает ссылку на последний узел в списке, имеющий значение `value`. Если искомое значение `value` отсутствует в списке, то возвращается пустое значение
`public bool Remove(T value)`	Удаляет из списка первый узел, содержащий значение `value`. Возвращает логическое значение `true`, если узел удален, т.е. если узел со значением `value` обнаружен в списке и удален; в противном случае возвращает логическое значение `false`
`public void Remove(LinkedList<T> node)`	Удаляет из списка узел, соответствующий указанному узлу `node`. Если узел `node` отсутствует в списке, то генерируется исключение `InvalidOperationException`
`public void RemoveFirst()`	Удаляет из списка первый узел
`public void RemoveLast()`	Удаляет из списка последний узел

В приведенном ниже примере программы демонстрируется применение класса `LinkedList<T>`.

```
// Продемонстрировать применение класса LinkedList<T>.

using System;
using System.Collections.Generic;
```

```
class GenLinkedListDemo {
  static void Main() {
    // Создать связный список.
    LinkedList<char> ll = new LinkedList<char>();

    Console.WriteLine("Исходное количество элементов в списке: " + ll.Count);

    Console.WriteLine();

    Console.WriteLine("Добавить в список 5 элементов");
    // Добавить элементы в связный список.
    ll.AddFirst('A');
    ll.AddFirst('B');
    ll.AddFirst('C');
    ll.AddFirst('D');
    ll.AddFirst('E');

    Console.WriteLine("Количество элементов в списке: " + ll.Count);

    // Отобразить связный список, обойдя его вручную.
    LinkedListNode<char> node;

    Console.Write("Отобразить содержимое списка по ссылкам: ");
    for(node = ll.First; node != null; node = node.Next)
      Console.Write(node.Value + " ");

    Console.WriteLine("\n");

    // Отобразить связный список, обойдя его в цикле foreach.
    Console.Write("Отобразить содержимое списка в цикле foreach: ");
    foreach(char ch in ll)
      Console.Write(ch + " ");

    Console.WriteLine("\n");

    // Отобразить связный список, обойдя его вручную в обратном направлении.
    Console.Write("Следовать по ссылкам в обратном направлении: ");
    for(node = ll.Last; node != null; node = node.Previous)
      Console.Write(node.Value + " ");

    Console.WriteLine("\n");

    // Удалить из списка два элемента.
    Console.WriteLine("Удалить 2 элемента из списка");

    // Удалить элементы из связного списка.
    ll.Remove('C');
    ll.Remove('A');

    Console.WriteLine("Количество элементов в списке: " + ll.Count);

    // Отобразить содержимое видоизмененного списка в цикле foreach.
```

```
    Console.Write("Содержимое списка после удаления элементов: ");
    foreach(char ch in ll)
      Console.Write(ch + " ");

    Console.WriteLine("\n");

    // Добавить три элемента в конец списка.
    ll.AddLast('X');
    ll.AddLast('Y');
    ll.AddLast('Z');

    Console.Write("Содержимое списка после ввода элементов: ");
    foreach(char ch in ll)
      Console.Write(ch + " ");

    Console.WriteLine("\n");
}
}
```

Ниже приведен результат выполнения этой программы.

```
Исходное количество элементов в списке: 0

Добавить в список 5 элементов
Количество элементов в списке: 5
Отобразить содержимое списка по ссылкам: E D C B A

Отобразить содержимое списка в цикле foreach: E D C B A

Следовать по ссылкам в обратном направлении: A B C D E

Удалить 2 элемента из списка
Количество элементов в списке: 3
Содержимое списка после удаления элементов: E D B

Содержимое списка после ввода элементов: E D B X Y Z
```

Самое примечательное в этой программе — это обход списка в прямом и обратном направлении, следуя по ссылкам, предоставляемым свойствами `Next` и `Previous`. Двунаправленный характер подобных связных списков имеет особое значение для приложений, управляющих базами данных, где нередко требуется перемещаться по списку в обоих направлениях.

Класс `Dictionary<TKey, TValue>`

Класс `Dictionary<TKey, TValue>` позволяет хранить пары "ключ–значение" в коллекции как в словаре. Значения доступны в словаре по соответствующим ключам. В этом отношении данный класс аналогичен необобщенному классу `Hashtable`. В классе `Dictionary<TKey, TValue>` реализуются интерфейсы `IDictionary`, `IDictionary<TKey, TValue>`, `ICollection`, `ICollection<KeyValuePair<TKey, TValue>>`, `IEnumerable`, `IEnumerable<KeyValuePair<TKey, TValue>>`, `ISerializable` и `IDeserializationCallback`. В двух последних интерфейсах поддерживается сериализация списка. Словари имеют динамический характер, расширяясь по мере необходимости.

В классе `Dictionary<TKey, TValue>` предоставляется немало конструкторов. Ниже перечислены наиболее часто используемые из них.

```
public Dictionary()
public Dictionary(IDictionary<TKey, TValue> dictionary)
public Dictionary(int capacity)
```

В первом конструкторе создается пустой словарь с выбираемой по умолчанию первоначальной емкостью. Во втором конструкторе создается словарь с указанным количеством элементов *dictionary*. А в третьем конструкторе с помощью параметра *capacity* указывается емкость коллекции, создаваемой в виде словаря. Если размер словаря заранее известен, то, указав емкость создаваемой коллекции, можно исключить изменение размера словаря во время выполнения, что, как правило, требует дополнительных затрат вычислительных ресурсов.

В классе `Dictionary<TKey, TValue>` определяется также ряд методов. Некоторые наиболее часто используемые методы этого класса сведены в табл. 25.17.

Таблица 25.17. Наиболее часто используемые методы, определенные в классе `Dictionary<TKey, TValue>`

Метод	Описание
`public void Add(TKey key, TValue value)`	Добавляет в словарь пару "ключ-значение", определяемую параметрами *key* и *value*. Если ключ *key* уже находится в словаре, то его значение не изменяется, и генерируется исключение `ArgumentException`
`public bool ContainsKey(TKey key)`	Возвращает логическое значение `true`, если вызывающий словарь содержит объект *key* в качестве ключа; а иначе — логическое значение `false`
`public bool ContainsValue(TValue value)`	Возвращает логическое значение `true`, если вызывающий словарь содержит значение *value*; в противном случае — логическое значение `false`
`public bool Remove(TKey key)`	Удаляет ключ *key* из словаря. При удачном исходе операции возвращается логическое значение `true`, а если ключ *key* отсутствует в словаре — логическое значение `false`

Кроме того, в классе `Dictionary<TKey, TValue>` определяются собственные свойства, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Эти свойства приведены ниже.

Свойство	Описание
`public IEqualityComparer<TKey> Comparer { get; }`	Получает метод сравнения для вызывающего словаря
`public Dictionary<TKey, TValue>. KeyCollection Keys { get; }`	Получает коллекцию ключей
`public Dictionary<TKey, TValue>. ValueCollection Values { get; }`	Получает коллекцию значений

Следует иметь в виду, что ключи и значения, содержащиеся в коллекции, доступны отдельными списками с помощью свойств Keys и Values. В коллекциях типа Dictionary<TKey, TValue>.KeyCollection и Dictionary<TKey, TValue>.ValueCollection реализуются как обобщенные, так и необобщенные формы интерфейсов ICollection и IEnumerable.

И наконец, в классе Dictionary<TKey, TValue> реализуется приведенный ниже индексатор, определенный в интерфейсе IDictionary<TKey, TValue>.

```
public TValue this[TKey key] { get; set; }
```

Этот индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. Но в качестве индекса в данном случае служит ключ элемента, а не сам индекс.

При перечислении коллекции типа Dictionary<TKey, TValue> из нее возвращаются пары "ключ–значение" в форме структуры KeyValuePair<TKey, TValue>. Напомним, что в этой структуре определяются два поля.

```
public TKey Key;
public TValue Value;
```

В этих полях содержится ключ или значение соответствующего элемента коллекции. Как правило, структура KeyValuePair<TKey, TValue> не используется непосредственно, поскольку средства класса Dictionary<TKey, TValue> позволяют работать с ключами и значениями по отдельности. Но при перечислении коллекции типа Dictionary<TKey, TValue>, например, в цикле foreach перечисляемыми объектами являются пары типа KeyValuePair.

Все ключи в коллекции типа Dictionary<TKey, TValue> должны быть уникальными, причем ключ не должен изменяться до тех пор, пока он служит в качестве ключа. В то же время значения не обязательно должны быть уникальными. К тому же объекты не хранятся в коллекции типа Dictionary<TKey, TValue> в отсортированном порядке.

В приведенном ниже примере демонстрируется применение класса Dictionary<TKey, TValue>.

```
// Продемонстрировать применение класса обобщенной
// коллекции Dictionary<TKey, TValue>.

using System;
using System.Collections.Generic;

class GenDictionaryDemo {
  static void Main() {
    // Создать словарь для хранения имен и фамилий
    // работников и их зарплаты.
    Dictionary<string, double> dict =
                new Dictionary<string, double>();

    // Добавить элементы в коллекцию.
    dict.Add("Батлер, Джон", 73000);
    dict.Add("Шварц, Сара", 59000);
    dict.Add("Пайк, Томас", 45000);
    dict.Add("Фрэнк, Эд", 99000);
```

```
    // Получить коллекцию ключей, т.е. фамилий и имен.
    ICollection<string> c = dict.Keys;

    // Использовать ключи для получения значений, т.е. зарплаты.
    foreach(string str in c)
     Console.WriteLine("{0},  зарплата: {1:C}", str, dict[str]);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Батлер, Джон, зарплата: $73,000.00
Шварц, Сара, зарплата: $59,000.00
Пайк, Томас, зарплата: $45,000.00
Фрэнк, Эд, зарплата: $99,000.00
```

Класс SortedDictionary<TKey, TValue>

В коллекции класса `SortedDictionary<TKey, TValue>` пары "ключ–значение" хранятся таким же образом, как и в коллекции класса `Dictionary<TKey, TValue>`, за исключением того, что они отсортированы по соответствующему ключу. В классе `SortedDictionary<TKey, TValue>` реализуются интерфейсы `IDictionary`, `IDictionary<TKey, TValue>`, `ICollection`, `ICollection<KeyValuePair<TKey, TValue>>`, `IEnumerable` и `IEnumerable<KeyValuePair<TKey, TValue>>`. В классе `SortedDictionary<TKey, TValue>` предоставляются также следующие конструкторы.

```
public SortedDictionary()
public SortedDictionary(IDictionary<TKey, TValue> dictionary)
public SortedDictionary(IComparer<TKey> comparer)
public SortedDictionary(IDictionary<TKey, TValue> dictionary,
                        IComparer<TKey> comparer)
```

В первом конструкторе создается пустой словарь, во втором конструкторе — словарь с указанным количеством элементов *dictionary*. В третьем конструкторе допускается указывать с помощью параметра *comparer* типа `IComparer` способ сравнения, используемый для сортировки, а в четвертом конструкторе — инициализировать словарь, помимо указания способа сравнения.

В классе `SortedDictionary<TKey, TValue>` определен ряд методов. Некоторые наиболее часто используемые методы этого класса сведены в табл. 25.18.

Таблица 25.18. Наиболее часто используемые методы, определенные в классе SortedDictionary<TKey, TValue>

Метод	Описание
`public void Add(TKey key, TValue value)`	Добавляет в словарь пару "ключ-значение", определяемую параметрами *key* и *value*. Если ключ *key* уже находится в словаре, то его значение не изменяется, и генерируется исключение `ArgumentException`
`public bool ContainsKey(TKey key)`	Возвращает логическое значение `true`, если вызывающий словарь содержит объект *key* в качестве ключа; в противном случае — логическое значение `false`

Окончание табл. 25.18

Метод	Описание
`public bool ContainsValue(TValue value)`	Возвращает логическое значение `true`, если вызывающий словарь содержит значение *value*; в противном случае — логическое значение `false`
`public bool Remove(TKey key)`	Удаляет ключ *key* из словаря. При удачном исходе операции возвращается логическое значение `true`, а если ключ *key* отсутствует в словаре — логическое значение `false`

Кроме того, в классе `SortedDictionary<TKey, TValue>` определяются собственные свойства, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Эти свойства приведены ниже.

Свойство	Описание
`public Icomparer<TKey> Comparer { get; }`	Получает метод сравнения для вызывающего словаря
`public SortedDictionary<TKey, TValue>.KeyCollection Keys { get; }`	Получает коллекцию ключей
`public SortedDictionary<TKey, TValue>.ValueCollection Values { get; }`	Получает коллекцию значений

Следует иметь в виду, что ключи и значения, содержащиеся в коллекции, доступны отдельными списками с помощью свойств `Keys` и `Values`. В коллекциях типа `SortedDictionary<TKey, TValue>.KeyCollection` и `SortedDictionary<TKey, TValue>.ValueCollection` реализуются как обобщенные, так и необобщенные формы интерфейсов `ICollection` и `IEnumerable`.

И наконец, в классе `SortedDictionary<TKey, TValue>` реализуется приведенный ниже индексатор, определенный в интерфейсе `IDictionary<TKey, TValue>`.

```
public TValue this[TKey key] { get; set; }
```

Этот индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. Но в данном случае в качестве индекса служит ключ элемента, а не сам индекс.

При перечислении коллекции типа `SortedDictionary<TKey, TValue>` из нее возвращаются пары "ключ–значение" в форме структуры `KeyValuePair<TKey, TValue>`. Напомним, что в этой структуре определяются два следующих поля.

```
public TKey Key;
public TValue Value;
```

В этих полях содержится ключ или значение соответствующего элемента коллекции. Как правило, структура `KeyValuePair<TKey, TValue>` не используется непосредственно, поскольку средства класса `SortedDictionary<TKey, TValue>` позволяют работать с ключами и значениями по отдельности. Но при перечислении коллекции типа `SortedDictionary<TKey, TValue>`, например в цикле `foreach`, перечисляемыми объектами являются пары типа `KeyValuePair`.

Все ключи в коллекции типа SortedDictionary<TKey, TValue> должны быть уникальными, причем ключ не должен изменяться до тех пор, пока он служит в качестве ключа. В то же время значения не обязательно должны быть уникальными.

В приведенном ниже примере демонстрируется применение класса SortedDictionary<TKey, TValue>. Это измененный вариант предыдущего примера, демонстрировавшего применение класса Dictionary<TKey, TValue>. В данном варианте база данных работников отсортирована по фамилии и имени работника, которые служат в качестве ключа.

```
// Продемонстрировать применение класса обобщенной
// коллекции SortedDictionary<TKey, TValue>.

using System;
using System.Collections.Generic;

class GenSortedDictionaryDemo {
  static void Main() {
    // Создать словарь для хранения имен и фамилий
    // работников и их зарплаты.
    SortedDictionary<string, double> dict =
          new SortedDictionary<string, double>();

    // Добавить элементы в коллекцию.
    dict.Add("Батлер, Джон", 73000);
    dict.Add("Шварц, Сара", 59000);
    dict.Add("Пайк, Томас", 45000);
    dict.Add("Фрэнк, Эд", 99000);

    // Получить коллекцию ключей, т.е. фамилий и имен.
    ICollection<string> c = dict.Keys;

    // Использовать ключи для получения значений, т.е. зарплаты.
    foreach(string str in c)
      Console.WriteLine("{0}, зарплата: {1:C}", str, dict[str]);
  }
}
```

Эта программа дает следующий результат.

```
Батлер, Джон, зарплата: $73,000.00
Пайк, Томас, зарплата: $45,000.00
Фрэнк, Эд, зарплата: $99,000.00
Шварц, Сара, зарплата: $59,000.00
```

Как видите, список работников и их зарплаты отсортированы по ключу, в качестве которого в данном случае служит фамилия и имя работника.

Класс SortedList<TKey, TValue>

В коллекции класса SortedList<TKey, TValue> хранится отсортированный список пар "ключ–значение". Это обобщенный эквивалент класса необобщенной коллекции SortedList. В классе SortedList<TKey, TValue> реализуются интерфейсы IDictionary, IDictionary<TKey, TValue>, ICollection, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable и IEnumerable<KeyValuePair<TKey, TValue>>. Размер

коллекции типа SortedList<TKey, TValue> изменяется динамически, автоматически увеличиваясь по мере необходимости. Класс SortedList<TKey, TValue> подобен классу SortedDictionary<TKey, TValue>, но у него другие рабочие характеристики. В частности, класс SortedList<TKey, TValue> использует меньше памяти, тогда как класс SortedDictionary<TKey, TValue> позволяет быстрее вставлять неупорядоченные элементы в коллекцию.

В классе SortedList<TKey, TValue> предоставляется немало конструкторов. Ниже перечислены наиболее часто используемые конструкторы этого класса.

```
public SortedList()
public SortedList(IDictionary<TKey, TValue> dictionary)
public SortedList(int capacity)
public SortedList(IComparer<TK> comparer)
```

В первой форме конструктора создается пустой список с выбираемой по умолчанию первоначальной емкостью. Во второй форме конструктора создается отсортированный список с указанным количеством элементов *dictionary*. В третьей форме конструктора с помощью параметра *capacity* задается емкость коллекции, создаваемой в виде отсортированного списка. Если размер списка заранее известен, то, указав емкость создаваемой коллекции, можно исключить изменение размера списка во время выполнения, что, как правило, требует дополнительных затрат вычислительных ресурсов. И в четвертой форме конструктора допускается указывать с помощью параметра *comparer* способ сравнения объектов, содержащихся в списке.

Емкость коллекции типа SortedList<TKey, TValue> увеличивается автоматически по мере необходимости, когда в список добавляются новые элементы. Если текущая емкость коллекции превышается, то она увеличивается. Преимущество указания емкости коллекции типа SortedList<TKey, TValue> при ее создании заключается в снижении или полном исключении издержек на изменение размера коллекции. Разумеется, указывать емкость коллекции целесообразно лишь в том случае, если заранее известно, сколько элементов требуется хранить в ней.

В классе SortedList<TKey, TValue> определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Некоторые из наиболее часто используемых методов этого класса перечислены в табл. 25.19. Следует иметь в виду, что перечислитель, возвращаемый методом GetEnumerator(), служит для перечисления пар "ключ–значение", хранящихся в отсортированном списке в виде объектов типа KeyValuePair.

Таблица 25.19. Наиболее часто используемые методы, определенные в классе SortedList<TKey, TValue>

Метод	Описание
public void Add(TKey *key*, TValue *value*)	Добавляет в список пару "ключ-значение", определяемую параметрами *key* и *value*. Если ключ *key* уже находится в списке, то его значение не изменяется, и генерируется исключение ArgumentException
public bool ContainsKey(TK *key*)	Возвращает логическое значение true, если вызывающий список содержит объект *key* в качестве ключа; а иначе — логическое значение false

Окончание табл. 25.19

Метод	Описание
`public bool ContainsValue(TValue value)`	Возвращает логическое значение `true`, если вызывающий список содержит значение `value`; в противном случае — логическое значение `false`
`public IEnumerator<KeyValuePair <TKey, TValue>> GetEnumerator()`	Возвращает перечислитель для вызывающего словаря
`public int IndexOfKey(TKey key)`	Возвращает индекс ключа `key`. Если искомый ключ не обнаружен в списке, возвращается значение `-1`
`public int IndexOfValue(TValue value)`	Возвращает индекс первого вхождения значения `value` в вызывающем списке. Если искомое значение не обнаружено в списке, возвращается значение `-1`
`public bool Remove(TKey key)`	Удаляет из списка пару "ключ-значение" по указанному ключу `key`. При удачном исходе операции возвращается логическое значение `true`, а если ключ `key` отсутствует в списке — логическое значение `false`
`public void RemoveAt(int index)`	Удаляет из списка пару "ключ-значение" по указанному индексу `index`
`public void TrimExcess()`	Сокращает избыточную емкость вызывающей коллекции в виде отсортированного списка

Кроме того, в классе `SortedList<TK, TV>` определяются собственные свойства, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Эти свойства приведены ниже.

Свойство	Описание
`public int Capacity { get; set; }`	Получает или устанавливает емкость вызывающей коллекции в виде отсортированного списка
`public IComparer<TK> Comparer { get; }`	Получает метод сравнения для вызывающего списка
`public IList<TK> Keys { get; }`	Получает коллекцию ключей
`public IList<TV> Values { get; }`	Получает коллекцию значений

И наконец, в классе `SortedList<TKey, TValue>` реализуется приведенный ниже индексатор, определенный в интерфейсе `IDictionary<TKey, TValue>`.

```
public TValue this[TKey key] { get; set; }
```

Этот индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. Но в данном случае в качестве индекса служит ключ элемента, а не сам индекс.

В приведенном ниже примере демонстрируется применение класса `SortedList<TKey, TValue>`. Это еще один измененный вариант представленного

ранее примера базы данных работников. В данном варианте база данных хранится в коллекции типа SortedList.

```
// Продемонстрировать применение класса обобщенной
// коллекции SortedList<TKey, TValue>.

using System;
using System.Collections.Generic;

class GenSLDemo {
  static void Main() {
    // Создать коллекцию в виде отсортированного списка
    // для хранения имен и фамилий работников и их зарплаты.
    SortedList<string, double> sl =
          new SortedList<string, double>();

    // Добавить элементы в коллекцию.
    sl.Add("Батлер, Джон", 73000);
    sl.Add("Шварц, Сара", 59000);
    sl.Add("Пайк, Томас", 45000);
    sl.Add("Фрэнк, Эд", 99000);

    // Получить коллекцию ключей, т.е. фамилий и имен.
    ICollection<string> c = sl.Keys;

    // Использовать ключи для получения значений, т.е. зарплаты.
    foreach(string str in c)
      Console.WriteLine("{0}, зарплата: {1:C}", str, sl[str]);

    Console.WriteLine();
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Батлер, Джон, зарплата: $73,000.00
Пайк, Томас, зарплата: $45,000.00
Фрэнк, Эд, зарплата: $99,000.00
Шварц, Сара, зарплата: $59,000.00
```

Как видите, список работников и их зарплаты отсортированы по ключу, в качестве которого в данном случае служит фамилия и имя работника.

Класс Stack<T>

Класс Stack<T> является обобщенным эквивалентом класса необобщенной коллекции Stack. В нем поддерживается стек в виде списка, действующего по принципу "первым пришел — последним обслужен". В этом классе реализуются интерфейсы Collection, IEnumerable и IEnumerable<T>. Кроме того, в классе Stack<T> непосредственно реализуются методы Clear(), Contains() и CopyTo(), определенные в интерфейсе ICollection<T>. А методы Add() и Remove() в этом классе не поддерживаются, как, впрочем, и свойство IsReadOnly. Коллекция класса Stack<T> имеет динамический характер, расширяясь по мере необходимости, чтобы вместить все элементы, которые должны в ней храниться. В классе Stack<T> определяются следующие конструкторы.

```
public Stack()
public Stack(int capacity)
public Stack(IEnumerable<T> collection)
```

В первой форме конструктора создается пустой стек с выбираемой по умолчанию первоначальной емкостью, а во второй форме — пустой стек, первоначальный размер которого определяет параметр `capacity`. И в третьей форме создается стек, содержащий элементы коллекции, определяемой параметром `collection`. Его первоначальная емкость равна количеству указанных элементов.

В классе `Stack<T>` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются, а также в интерфейсе `ICollection<T>`. Некоторые из наиболее часто используемых методов этого класса перечислены в табл. 25.20. Как и в классе `Stack`, эти методы обычно применяются следующим образом. Для того чтобы поместить объект на вершине стека, вызывается метод `Push()`. А для того чтобы извлечь и удалить объект из вершины стека, вызывается метод `Pop()`. Если же объект требуется только извлечь, но не удалить из вершины стека, то вызывается метод `Peek()`. А если вызвать метод `Pop()` или `Peek()`, когда вызывающий стек пуст, то сгенерируется исключение `InvalidOperationException`.

Таблица 25.20. Методы, определенные в классе `Stack<T>`

Метод	Описание
`public T Peek()`	Возвращает элемент, находящийся на вершине стека, но не удаляет его
`public T Pop()`	Возвращает элемент, находящийся на вершине стека, удаляя его в процессе работы
`public void Push(T item)`	Помещает элемент `item` в стек
`public T[] ToArray()`	Возвращает массив, содержащий копии элементов вызывающего стека
`public void TrimExcess()`	Сокращает избыточную емкость вызывающей коллекции в виде стека

В приведенном ниже примере программы демонстрируется применение класса `Stack<T>`.

```
// Продемонстрировать применение класса Stack<T>.

using System;
using System.Collections.Generic;

class GenStackDemo {
  static void Main() {
    Stack<string> st = new Stack<string>();

    st.Push("один");
    st.Push("два");
    st.Push("три");
    st.Push("четыре");
    st.Push("пять");

    while(st.Count > 0) {
      string str = st.Pop();
```

```
      Console.Write(str + " ");
    }

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
пять четыре три два один
```

Класс Queue<T>

Класс `Queue<T>` является обобщенным эквивалентом класса необобщенной коллекции `Queue`. В нем поддерживается очередь в виде списка, действующего по принципу "первым пришел — первым обслужен". В этом классе реализуются интерфейсы `ICollection`, `IEnumerable` и `IEnumerable<T>`. Кроме того, в классе `Queue<T>` непосредственно реализуются методы `Clear()`, `Contains()` и `CopyTo()`, определенные в интерфейсе `ICollection<T>`. А методы `Add()` и `Remove()` в этом классе не поддерживаются, как, впрочем, и свойство `IsReadOnly`. Коллекция класса `Queue<T>` имеет динамический характер, расширяясь по мере необходимости, чтобы вместить все элементы, которые должны храниться в ней. В классе `Queue<T>` определяются следующие конструкторы.

```
public Queue()
public Queue(int capacity)
public Queue(IEnumerable<T> collection)
```

В первой форме конструктора создается пустая очередь с выбираемой по умолчанию первоначальной емкостью, а во второй форме — пустая очередь, первоначальный размер которой определяет параметр *capacity*. И в третьей форме создается очередь, содержащая элементы коллекции, определяемой параметром *collection*. Ее первоначальная емкость равна количеству указанных элементов.

В классе `Queue<T>` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются, а также в интерфейсе `ICollection<T>`. Некоторые из наиболее часто используемых методов этого класса перечислены в табл. 25.21. Как и в классе `Queue`, эти методы обычно применяются следующим образом. Для того чтобы поместить объект в очередь, вызывается метод `Enqueue()`. Если требуется извлечь и удалить первый объект из начала очереди, то вызывается метод `Dequeue()`. Если же требуется извлечь, но не удалять следующий объект из очереди, то вызывается метод `Peek()`. А если методы `Dequeue()` и `Peek()` вызываются, когда очередь пуста, то генерируется исключение `InvalidOperationException`.

Таблица 25.21. Методы, определенные в классе Queue<T>

Метод	Описание
`public T Dequeue()`	Возвращает объект из начала вызывающей очереди. Возвращаемый объект удаляется из очереди
`public void Enqueue(T item)`	Добавляет элемент *item* в конец очереди
`public T Peek()`	Возвращает элемент из начала вызывающей очереди, но не удаляет его

Окончание табл. 25.21

Метод	Описание
`public virtual T[] ToArray()`	Возвращает массив, который содержит копии элементов из вызывающей очереди
`public void TrimExcess()`	Сокращает избыточную емкость вызывающей коллекции в виде очереди

В приведенном ниже примере демонстрируется применение класса `Queue<T>`.

```
// Продемонстрировать применение класса Queue<T>.

using System;
using System.Collections.Generic;

class GenQueueDemo {
  static void Main() {
    Queue<double> q = new Queue<double>();

    q.Enqueue(98.6);
    q.Enqueue(212.0);
    q.Enqueue(32.0);
    q.Enqueue(3.1416);

    double sum = 0.0;
    Console.Write("Очередь содержит: ");
    while(q.Count > 0) {
      double val = q.Dequeue();
      Console.Write(val + " ");
      sum += val;
     }

    Console.WriteLine("\nИтоговая сумма равна " + sum);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
Очередь содержит: 98.6 212 32 3.1416
Итоговая сумма равна 345.7416
```

Класс `HashSet<T>`

В классе `HashSet<T>` поддерживается коллекция, реализующая множество. Для хранения элементов этого множества в нем используется хеш-таблица. В классе `HashSet<T>` реализуются интерфейсы `ICollection<T>`, `ISet<T>`, `IEnumerable`, `IEnumerable<T>`, `ISerializable`, а также `IDeserializationCallback`. В коллекции типа `HashSet<T>` реализуется множество, все элементы которого являются уникальными. Иными словами, дубликаты в таком множестве не допускаются. Порядок следования элементов во множестве не указывается. В классе `HashSet<T>` определяется полный набор операций с множеством, определенных в интерфейсе `ISet<T>`, включая пересечение, объединение и разноименность. Благодаря этому класс `HashSet<T>` оказывается идеальным средством для работы с множествами объектов, когда порядок расположения элементов во множестве особого значения не имеет. Коллекция типа

HashSet<T> имеет динамический характер и расширяется по мере необходимости, чтобы вместить все элементы, которые должны в ней храниться.

Ниже перечислены наиболее употребительные конструкторы, определенные в классе HashSet<T>.

```
public HashSet()
public HashSet(IEnumerable<T> collection)
public HashSet(IEqualityCompare comparer)
public HashSet(IEnumerable<T> collection, IEqualityCompare comparer)
```

В первой форме конструктора создается пустое множество, а во второй форме — множество, состоящее из элементов указываемой коллекции *collection*. В третьей форме конструктора допускается указывать способ сравнения с помощью параметра *comparer*. А в четвертой форме создается множество, состоящее из элементов указываемой коллекции *collection*, и используется заданный способ сравнения *comparer*. Имеется также пятая форма конструктора данного класса, в которой допускается инициализировать множество последовательно упорядоченными данными.

В классе HashSet<T> реализуется интерфейс ISet<T>, а следовательно, в нем предоставляется полный набор операций со множествами. В этом классе предоставляется также метод RemoveWhere(), удаляющий из множества элементы, не удовлетворяющие заданному условию, или предикату.

Помимо свойств, определенных в интерфейсах, которые реализуются в классе HashSet<T>, в него введено дополнительное свойство Comparer, приведенное ниже.

```
public IEqualityComparer<T> Comparer { get; }
```

Оно позволяет получать метод сравнения для вызывающего хеш-множества.

Ниже приведен конкретный пример применения класса HashSet<T>.

```
// Продемонстрировать применение класса HashSet<T>.

using System;
using System.Collections.Generic;

class HashSetDemo {

  static void Show(string msg, HashSet<char> set) {

    Console.Write(msg);
    foreach(char ch in set)
      Console.Write(ch + " ");
    Console.WriteLine();
  }

  static void Main() {
    HashSet<char> setA = new HashSet<char>();
    HashSet<char> setB = new HashSet<char>();

    setA.Add('A');
    setA.Add('B');
    setA.Add('C');

    setB.Add('C');
    setB.Add('D');
```

```
    setB.Add('E');

    Show("Исходное содержимое множества setA: ", setA);
    Show("Исходное содержимое множества setB: ", setB);

    setA.SymmetricExceptWith(setB);
    Show("Содержимое множества setA после " +
         "разноименности со множеством SetB: ", setA);

    setA.UnionWith(setB);
     Show("Содержимое множества setA после " +
          "объединения со множеством SetB: ", setA);

    setA.ExceptWith(setB);
    Show("Содержимое множества setA после " +
         "вычитания из множества setB: ", setA);

    Console.WriteLine();
  }
}
```

Ниже приведен результат выполнения программы из данного примера.

```
Исходное содержимое множества setA: A B C
Исходное содержимое множества setB: C D E
Содержимое множества setA после разноименности со множеством SetB: A B D E
Содержимое множества setA после объединения со множеством SetB: A B D E C
Содержимое множества setA после вычитания из множества setB: A B
```

Класс SortedSet<T>

Класс `SortedSet<T>` представляет собой новую разновидность коллекции, введенную в версию 4.0 среды .NET Framework. В нем поддерживается коллекция, реализующая отсортированное множество. В классе `SortedSet<T>` реализуются интерфейсы `ISet<T>`, `ICollection`, `ICollection<T>`, `IEnumerable`, `IEnumerable<T>`, `ISerializable`, а также `IDeserializationCallback`. В коллекции типа `SortedSet<T>` реализуется множество, все элементы которого являются уникальными. Иными словами, дубликаты в таком множестве не допускаются. В классе `SortedSet<T>` определяется полный набор операций с множеством, определенных в интерфейсе `ISet<T>`, включая пересечение, объединение и разноименность. Благодаря тому что все элементы коллекции типа `SortedSet<T>` сохраняются в отсортированном порядке, класс `SortedSet<T>` оказывается идеальным средством для работы с отсортированными множествами объектов. Коллекция типа `SortedSet<T>` имеет динамический характер и расширяется по мере необходимости, чтобы вместить все элементы, которые должны в ней храниться.

Ниже перечислены четыре наиболее часто используемые конструктора, определенных в классе `SortedSet<T>`.

```
public SortedSet()
public SortedSet(IEnumerable<T> collection)
public SortedSet(IComparer comparer)
public SortedSet(IEnumerable<T> collection, IComparer comparer)
```

В первой форме конструктора создается пустое множество, а во второй форме — множество, состоящее из элементов указываемой коллекции *collection*. В третьей форме конструктора допускается указывать способ сравнения с помощью параметра *comparer*. А в четвертой форме создается множество, состоящее из элементов указываемой коллекции *collection*, и используется заданный способ сравнения *comparer*. Имеется также пятая форма конструктора данного класса, в которой допускается инициализировать множество последовательно упорядоченными данными.

В классе `SortedSet<T>` реализуется интерфейс `ISet<T>`, а следовательно, в нем предоставляется полный набор операций со множествами. В этом классе предоставляется также метод `GetViewBetween()`, возвращающий часть множества в форме объекта типа `SortedSet<T>`, метод `RemoveWhere()`, удаляющий из множества элементы, не удовлетворяющие заданному условию, или предикату, а также метод `Reverse()`, возвращающий объект типа `IEnumerable<T>`, который циклически проходит множество в обратном порядке.

Помимо свойств, определенных в интерфейсах, которые реализуются в классе `SortedSet<T>`, в него введены дополнительные свойства, приведенные ниже.

```
public IComparer<T> Comparer { get; }
public T Max { get; }
public T Min { get; }
```

Свойство `Comparer` получает способ сравнения для вызывающего множества. Свойство `Max` получает наибольшее значение во множестве, а свойство `Min` — наименьшее значение во множестве.

В качестве примера применения класса `SortedSet<T>` на практике просто замените обозначение `HashSet` на `SortedSet` в исходном коде программы из предыдущего подраздела, посвященного коллекциям типа `HashSet<T>`.

Параллельные коллекции

В версию 4.0 среды .NET Framework добавлено новое пространство имен `System.Collections.Concurrent`. Оно содержит коллекции, которые являются потокобезопасными и специально предназначены для параллельного программирования. Это означает, что они могут безопасно использоваться в многопоточной программе, где возможен одновременный доступ к коллекции со стороны двух или больше параллельно исполняемых потоков. Ниже перечислены классы параллельных коллекций.

Параллельная коллекция	Описание
`BlockingCollection<T>`	Предоставляет оболочку для блокирующей реализации интерфейса `IProducerConsumerCollection<T>`
`ConcurrentBag<T>`	Обеспечивает неупорядоченную реализацию интерфейса `IProducerConsumerCollection<T>`, которая оказывается наиболее пригодной в том случае, когда информация вырабатывается и потребляется в одном потоке
`ConcurrentDictionary <TKey, TValue>`	Сохраняет пары "ключ-значение", а значит, реализует параллельный словарь
`ConcurrentQueue<T>`	Реализует параллельную очередь и соответствующий вариант интерфейса `IProducerConsumerCollection<T>`
`ConcurrentStack<T>`	Реализует параллельный стек и соответствующий вариант интерфейса `IproducerConsumerCollection<T>`

Как видите, в нескольких классах параллельных коллекций реализуется интерфейс `IProducerConsumerCollection`. Этот интерфейс также определен в пространстве имен `System.Collections.Concurrent`. Он служит в качестве расширения интерфейсов `IEnumerable`, `IEnumerable<T>` и `ICollection`. Кроме того, в нем определены методы `TryAdd()` и `TryTake()`, поддерживающие шаблон "поставщик-потребитель". (Классический шаблон "поставщик-потребитель" отличается решением двух задач. Первая задача производит элементы коллекции, а другая потребляет их.) Метод `TryAdd()` пытается добавить элемент в коллекцию, а метод `TryTake()` — удалить элемент из коллекции. Ниже приведены формы объявления обоих методов.

```
bool TryAdd(T item)
bool TryTake(out T item)
```

Метод `TryAdd()` возвращает логическое значение `true`, если в коллекцию добавлен элемент *item*. А метод `TryTake()` возвращает логическое значение `true`, если элемент *item* удален из коллекции. Если метод `TryAdd()` выполнен успешно, то элемент *item* будет содержать объект. (Кроме того, в интерфейсе `IProducerConsumerCollection` указывается перегружаемый вариант метода `CopyTo()`, определяемого в интерфейсе `ICollection`, а также метода `ToArray()`, копирующего коллекцию в массив.)

Параллельные коллекции зачастую применяются в комбинации с библиотекой распараллеливания задач (TPL) или языком PLINQ. В силу особого характера этих коллекций все их классы не будут рассматриваться далее подробно. Вместо этого на конкретных примерах будет дан краткий обзор класса `BlockingCollection<T>`. Усвоив основы построения класса `BlockingCollection<T>`, вы сможете без особого труда разобраться и в остальных классах параллельных коллекций.

В классе `BlockingCollection<T>`, по существу, реализуется блокирующая очередь. Это означает, что в такой очереди автоматически устанавливается ожидание любых попыток вставить элемент в коллекцию, когда она заполнена, а также попыток удалить элемент из коллекции, когда она пуста. Это идеальное решение для тех ситуаций, которые связаны с применением шаблона "поставщик-потребитель". В классе `BlockingCollection<T>` реализуются интерфейсы `ICollection`, `IEnumerable`, `IEnumerable<T>`, а также `IDisposable`.

В классе `BlockingCollection<T>` определяются следующие конструкторы.

```
public BlockingCollection()
public BlockingCollection(int boundedCapacity)
public BlockingCollection(IProducerConsumerCollection<T> collection)
public BlockingCollection(IProducerConsumerCollection<T> collection,
                          int boundedCapacity)
```

В двух первых конструкторах в оболочку класса `BlockingCollection<T>` заключается коллекция, являющаяся экземпляром объекта типа `ConcurrentQueue<T>`. А в двух других конструкторах можно указать коллекцию, которая должна быть положена в основу коллекции типа `BlockingCollection<T>`. Если указывается параметр *boundedCapacity*, то он должен содержать максимальное количество объектов, которые коллекция должна содержать перед тем, как она окажется заблокированной. Если же параметр *boundedCapacity* не указан, то коллекция оказывается неограниченной.

Помимо методов `TryAdd()` и `TryTake()`, определяемых параллельно с теми, что указываются в интерфейсе `IProducerConsumerCollection<T>`, в классе `BlockingCollection<T>` определяется также ряд собственных методов. Ниже представлены методы, которые будут использоваться в приведенных далее примерах.

```
public void Add(T item)
public T Take()
```

Когда метод `Add()` вызывается для неограниченной коллекции, он добавляет элемент *item* в коллекцию и затем возвращает управление вызывающей части программы. А когда метод `Add()` вызывается для ограниченной коллекции, он блокирует доступ к ней, если она заполнена. После того как из коллекции будет удален один элемент или больше, указанный элемент *item* будет добавлен в коллекцию, и затем произойдет возврат из данного метода. Метод `Take()` удаляет элемент из коллекции и возвращает управление вызывающей части программы. (Имеются также варианты обоих методов, принимающие в качестве параметра признак задачи как экземпляр объекта типа `CancellationToken`.)

Применяя методы `Add()` и `Take()`, можно реализовать простой шаблон "поставщик-потребитель", как показано в приведенном ниже примере программы. В этой программе создается поставщик, формирующий символы от A до Z, а также потребитель, получающий эти символы. При этом создается коллекция типа `BlockingCollection<T>`, ограниченная 4 элементами.

```
// Простой пример коллекции типа BlockingCollection.

using System;
using System.Threading.Tasks;
using System.Threading;
using System.Collections.Concurrent;

class BlockingDemo {
  static BlockingCollection<char> bc;

   // Произвести и поставить символы от A до Z.
   static void Producer() {
    for(char ch = 'A'; ch <= 'Z'; ch++) {
      bc.Add(ch);
      Console.WriteLine("Производится символ " + ch);
    }
  }

  // Потребить 26 символов.
  static void Consumer() {
    for(int i=0; i < 26; i++)
      Console.WriteLine("Потребляется символ " + bc.Take());
  }

  static void Main() {
    // Использовать блокирующую коллекцию, ограниченную 4 элементами.
    bc = new BlockingCollection<char>(4);

    // Создать задачи поставщика и потребителя.
    Task Prod = new Task(Producer);
    Task Con = new Task(Consumer);

    // Запустить задачи.
    Con.Start();
    Prod.Start();
```

```
    // Ожидать завершения обеих задач.
    try {
      Task.WaitAll(Con, Prod);
    } catch(AggregateException exc) {
      Console.WriteLine(exc);
    } finally {
      Con.Dispose();
      Prod.Dispose();
      bc.Dispose();
    }
  }
}
```

Если запустить эту программу на выполнение, то на экране появится смешанный результат, выводимый поставщиком и потребителем. Отчасти это объясняется тем, что коллекция `bc` ограничена 4 элементами, а это означает, что в нее может быть добавлено только четыре элемента, прежде чем ее придется сократить. В качестве эксперимента попробуйте сделать коллекцию `bc` неограниченной и понаблюдайте за полученными результатами. В некоторых средах выполнения это приведет к тому, что все элементы коллекции будут сформированы до того, как начнется какое-либо их потребление. Кроме того, попробуйте ограничить коллекцию одним элементом. В этом случае одновременно может быть сформирован лишь один элемент.

Для работы с коллекцией типа `BlockingCollection<T>` может оказаться полезным и метод `CompleteAdding()`. Ниже приведена форма его объявления.

```
public void CompleteAdding()
```

Вызов этого метода означает, что в коллекцию не будет больше добавлено ни одного элемента. Это приводит к тому, что свойство `IsAddingComplete` принимает логическое значение `true`. Если же коллекция пуста, то свойство `IsCompleted` принимает логическое значение `true`, и в этом случае вызовы метода `Take()` не блокируются.

Ниже приведены формы объявления свойств `IsAddingComplete` и `IsCompleted`.

```
public bool IsCompleted { get; }
public bool IsAddingComplete { get; }
```

Когда коллекция типа `BlockingCollection<T>` только начинает формироваться, эти свойства содержат логическое значение `false`. А после вызова метода `CompleteAdding()` они принимают логическое значение `true`.

Ниже приведен вариант предыдущего примера программы, измененный с целью продемонстрировать применение метода `CompleteAdding()`, свойства `IsCompleted` и метода `TryTake()`.

```
// Применение методов CompleteAdding(), TryTake() и свойства IsCompleted.

using System;
using System.Threading.Tasks;
using System.Threading;
using System.Collections.Concurrent;

class BlockingDemo {
  static BlockingCollection<char> bc;
  // Произвести и поставить символы от A до Z.
  static void Producer() {
```

```
    for(char ch = 'A'; ch <= 'Z'; ch++) {
      bc.Add(ch);
      Console.WriteLine("Производится символ " + ch);
    }
    bc.CompleteAdding();
  }

  // Потреблять символы до тех пор, пока их будет производить поставщик.
  static void Consumer() {
    char ch;

    while(!bc.IsCompleted) {
      if(bc.TryTake(out ch))
        Console.WriteLine("Потребляется символ " + ch);
    }
  }

static void Main() {
    // Использовать блокирующую коллекцию, ограниченную 4 элементами.
    bc = new BlockingCollection<char>(4);

    // Создать задачи поставщика и потребителя.
    Task Prod = new Task(Producer);
    Task Con = new Task(Consumer);

    // Запустить задачи.
    Con.Start();
    Prod.Start();

    // Ожидать завершения обеих задач.
    try {
      Task.WaitAll(Con, Prod);
    } catch(AggregateException exc) {
      Console.WriteLine(exc);
    } finally {
      Con.Dispose();
      Prod.Dispose();
      bc.Dispose();
    }
  }
}
```

Этот вариант программы дает такой же результат, как и предыдущий. Главное его отличие заключается в том, что теперь метод `Producer()` может производить и поставлять сколько угодно элементов. С этой целью он просто вызывает метод `CompleteAdding()`, когда завершает создание элементов. А метод `Consumer()` лишь "потребляет" произведенные элементы до тех пор, пока свойство `IsCompleted` не примет логическое значение `true`.

Несмотря на специфический до некоторой степени характер параллельных коллекций, предназначенных в основном для параллельного программирования, у них, тем не менее, имеется немало общего с обычными, непараллельными коллекциями, описанными в предыдущих разделах. Если же вам приходится работать в среде параллельного программирования, то для организации одновременного доступа к данным из нескольких потоков вам, скорее всего, придется воспользоваться параллельными коллекциями.

Сохранение объектов, определяемых пользователем классов, в коллекции

Ради простоты приведенных выше примеров в коллекции, как правило, сохранялись объекты встроенных типов, в том числе `int`, `string` и `char`. Но ведь в коллекции можно хранить не только объекты встроенных типов. Достоинство коллекций в том и состоит, что в них допускается хранить объекты любого типа, включая объекты определяемых пользователем классов.

Рассмотрим сначала простой пример применения класса необобщенной коллекции `ArrayList` для хранения информации о товарных запасах. В этом классе инкапсулируется класс `Inventory`.

```
// Простой пример коллекции товарных запасов.

using System;
using System.Collections;

class Inventory {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
   return
   String.Format("{0,-10}Стоимость: {1,6:C} Наличие: {2}",
                 name, cost, onhand);
  }
}

class InventoryList {
  static void Main() {
    ArrayList inv = new ArrayList();

    // Добавить элементы в список.
    inv.Add(new Inventory("Кусачки", 5.95, 3));
    inv.Add(new Inventory("Отвертки", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Дрели", 19.88, 8));

    Console.WriteLine("Перечень товарных запасов:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
  }
}
```

При выполнении программы из данного примера получается следующий результат.

```
Перечень товарных запасов:
    Кусачки      Стоимость:  $5.95    Наличие: 3
    Отвертки     Стоимость:  $8.29    Наличие: 2
    Молотки      Стоимость:  $3.50    Наличие: 4
    Дрели        Стоимость: $19.88    Наличие: 8
```

Обратите внимание на то, что в данном примере программы не потребовалось никаких специальных действий для сохранения в коллекции объектов типа `Inventory`. Благодаря тому что все типы наследуют от класса `object`, в необобщенной коллекции можно хранить объекты любого типа. Именно поэтому в необобщенной коллекции нетрудно сохранить объекты определяемых пользователем классов. Безусловно, это также означает, что такая коллекция не типизирована.

Для того чтобы сохранить объекты определяемых пользователем классов в типизированной коллекции, придется воспользоваться классами обобщенных коллекций. В качестве примера ниже приведен измененный вариант программы из предыдущего примера. В этом варианте используется класс обобщенной коллекции `List<T>`, а результат получается таким же, как и прежде.

```
// Пример сохранения объектов класса Inventory в
// обобщенной коллекции класса List<T>.

using System;
using System.Collections.Generic;

class Inventory {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
    String.Format("{0,-10}Стоимость: {1,6:C} Наличие: {2}",
                  name, cost, onhand);
  }
}

class TypeSafeInventoryList {
  static void Main() {
    List<Inventory> inv = new List<Inventory>();

    // Добавить элементы в список.
    inv.Add(new Inventory("Кусачки", 5.95, 3));
    inv.Add(new Inventory("Отвертки", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Дрели", 19.88, 8));
```

```
    Console.WriteLine("Перечень товарных запасов:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
  }
}
```

Данный пример отличается от предыдущего лишь передачей типа `Inventory` в качестве аргумента типа конструктору класса `List<T>`. А в остальном оба примера рассматриваемой здесь программы практически одинаковы. Это, по существу, означает, что для применения обобщенной коллекции не требуется никаких особых усилий, но при сохранении в такой коллекции объекта конкретного типа строго соблюдается типовая безопасность.

Тем не менее для обоих примеров рассматриваемой здесь программы характерна еще одна особенность: они довольно кратки. Если учесть, что для организации динамического массива, где можно хранить, извлекать и обрабатывать данные товарных запасов, потребуется не менее 40 строк кода, то преимущества коллекций сразу же становятся очевидными. Нетрудно догадаться, что рассматриваемая здесь программа получится длиннее в несколько раз, если попытаться закодировать все эти функции коллекции вручную. Коллекции предлагают готовые решения самых разных задач программирования, и поэтому их следует использовать при всяком удобном случае.

У рассматриваемой здесь программы имеется все же один не совсем очевидный недостаток: коллекция не подлежит сортировке. Дело в том, что в классах `ArrayList` и `List<T>` отсутствуют средства для сравнения двух объектов типа `Inventory`. Но из этого положения имеются два выхода. Во-первых, в классе `Inventory` можно реализовать интерфейс `IComparable`, в котором определяется метод сравнения объектов данного класса. И во-вторых, для целей сравнения можно указать объект типа `IComparer`. Оба подхода рассматриваются далее по очереди.

Реализация интерфейса `IComparable`

Если требуется отсортировать коллекцию, состоящую из объектов определяемого пользователем класса, при условии, что они не сохраняются в коллекции класса `SortedList`, где элементы располагаются в отсортированном порядке, то в такой коллекции должен быть известен способ сортировки содержащихся в ней объектов. С этой целью можно, в частности, реализовать интерфейс `IComparable` для объектов сохраняемого типа. Интерфейс `IComparable` доступен в двух формах: обобщенной и необобщенной. Несмотря на сходство применения обеих форм данного интерфейса, между ними имеются некоторые, хотя и небольшие, отличия, рассматриваемые ниже.

Реализация интерфейса `IComparable` для необобщенных коллекций

Если требуется отсортировать объекты, хранящиеся в необобщенной коллекции, то для этой цели придется реализовать необобщенный вариант интерфейса `IComparable`. В этом варианте данного интерфейса определяется только один метод, `CompareTo()`, который определяет порядок выполнения самого сравнения. Ниже приведена общая форма объявления метода `CompareTo()`.

```
int CompareTo(object obj)
```

В методе `CompareTo()` вызывающий объект сравнивается с объектом *obj*. Для сортировки объектов по нарастающей конкретная реализация данного метода должна возвращать нулевое значение, если значения сравниваемых объектов равны; положительное — если значение вызывающего объекта больше, чем у объекта *obj*; и отрицательное — если значение вызывающего объекта меньше, чем у объекта *obj*. А для сортировки по убывающей можно обратить результат сравнения объектов. Если же тип объекта *obj* не подходит для сравнения с вызывающим объектом, то в методе `CompareTo()` может быть сгенерировано исключение `ArgumentException`.

В приведенном ниже примере программы демонстрируется конкретная реализация интерфейса `IComparable`. В этой программе интерфейс `IComparable` вводится в класс `Inventory`, разработанный в двух последних примерах из предыдущего раздела. В классе `Inventory` реализуется метод `CompareTo()` для сравнения полей `name` объектов данного класса, что дает возможность отсортировать товарные запасы по наименованию. Как показано в данном примере программы, коллекция объектов класса `Inventory` подлежит сортировке благодаря реализации интерфейса `IComparable` в этом классе.

```
// Реализовать интерфейс IComparable.

using System;
using System.Collections;

// Реализовать необобщенный вариант интерфейса IComparable.
class Inventory : IComparable {
  string name;
  double cost;
  int onhand;
  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
    String.Format("{0,-10}Стоимость: {1,6:C} Наличие: {2}",
                  name, cost, onhand);
  }

  // Реализовать интерфейс IComparable.
  public int CompareTo(object obj) {
    Inventory b;
    b = (Inventory) obj;
    return name.CompareTo(b.name);
  }
}

class IComparableDemo {
  static void Main() {
    ArrayList inv = new ArrayList();

    // Добавить элементы в список.
```

```
    inv.Add(new Inventory("Кусачки", 5.95, 3));
    inv.Add(new Inventory("Отвертки", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Дрели", 19.88, 8));

    Console.WriteLine("Перечень товарных запасов до сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
    Console.WriteLine();

    // Отсортировать список.
    inv.Sort();
    Console.WriteLine("Перечень товарных запасов после сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
  }
}
```

Ниже приведен результат выполнения данной программы. Обратите внимание на то, что после вызова метода `Sort()` товарные запасы оказываются отсортированными по наименованию.

```
Перечень товарных запасов до сортировки:

Кусачки     Стоимость:  $5.95    Наличие: 3
Отвертки    Стоимость:  $8.29    Наличие: 2
Молотки     Стоимость:  $3.50    Наличие: 4
Дрели       Стоимость: $19.88    Наличие: 8

Перечень товарных запасов после сортировки:
Дрели       Стоимость: $19.88    Наличие: 8
Кусачки     Стоимость:  $5.95    Наличие: 3
Молотки     Стоимость:  $3.50    Наличие: 4
Отвертки    Стоимость:  $8.29    Наличие: 2
```

Реализация интерфейса `IComparable` для обобщенных колекций

Если требуется отсортировать объекты, хранящиеся в обобщенной коллекции, то для этой цели придется реализовать обобщенный вариант интерфейса `IComparable<T>`. В этом варианте интерфейса `IComparable` определяется приведенная ниже обобщенная форма метода `CompareTo()`.

```
int CompareTo(T other)
```

В методе `CompareTo()` вызывающий объект сравнивается с другим объектом *other*. Для сортировки объектов по нарастающей конкретная реализация данного метода должна возвращать нулевое значение, если значения сравниваемых объектов равны; положительное — если значение вызывающего объекта больше, чем у объекта другого *other*; и отрицательное — если значение вызывающего объекта меньше, чем у другого объекта *other*. А для сортировки по убывающей можно обратить результат сравнения объектов. При реализации обобщенного интерфейса `IComparable<T>` имя типа реализующего класса обычно передается в качестве аргумента типа.

Приведенный ниже пример программы является вариантом предыдущего примера, измененным с целью реализовать и использовать обобщенный интерфейс IComparable<T>. Обратите внимание на применение класса обобщенной коллекции List<T> вместо класса необобщенной коллекции ArrayList.

```
// Реализовать интерфейс IComparable<T>.

using System;
using System.Collections.Generic;

// Реализовать обобщенный вариант интерфейса IComparable<T>.
class Inventory : IComparable<Inventory> {
  string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
   String.Format("{0,-10}Стоимость: {1,6:C} Наличие: {2}",
                 name, cost, onhand);
  }

  // Реализовать интерфейс IComparable<T>.
  public int CompareTo(Inventory obj) {
    return name.CompareTo(obj.name);
  }
}

class GenericIComparableDemo {
  static void Main() {
    List<Inventory> inv = new List<Inventory>();

    // Добавить элементы в список.
    inv.Add(new Inventory("Кусачки", 5.95, 3));
    inv.Add(new Inventory("Отвертки", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Дрели", 19.88, 8));

    Console.WriteLine("Перечень товарных запасов до сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
    Console.WriteLine();

    // Отсортировать список.
    inv.Sort();
    Console.WriteLine("Перечень товарных запасов после сортировки:");
    foreach(Inventory i in inv) {
```

```
      Console.WriteLine("    " + i);
    }
  }
}
```

Эта версия программы дает такой же результат, как и предыдущая, необобщенная версия.

Применение интерфейса IComparer

Для сортировки объектов определяемых пользователем классов зачастую проще всего реализовать в этих классах интерфейс `IComparable`. Тем не менее данную задачу можно решить и с помощью интерфейса `IComparer`. Для этой цели необходимо сначала создать класс, реализующий интерфейс `IComparer`, а затем указать объект этого класса, когда потребуется сравнение.

Интерфейс `IComparer` существует в двух формах: обобщенной и необобщенной. Несмотря на сходство применения обеих форм данного интерфейса, между ними имеются некоторые, хотя и небольшие, отличия, рассматриваемые ниже.

Применение необобщенного интерфейса IComparer

В необобщенном интерфейсе `IComparer` определяется только один метод, `Compare()`.

```
int Compare(object x, object y)
```

В методе `Compare()` сравниваются объекты `x` и `y`. Для сортировки объектов по нарастающей конкретная реализация данного метода должна возвращать нулевое значение, если значения сравниваемых объектов равны; положительное — если значение объекта *x* больше, чем у объекта *y*; и отрицательное — если значение объекта *x* меньше, чем у объекта *y*. А для сортировки по убывающей можно обратить результат сравнения объектов. Если же тип объекта *x* не подходит для сравнения с объектом *y*, то в методе `CompareTo()` может быть сгенерировано исключение `ArgumentException`.

Объект типа `IComparer` может быть указан при конструировании объекта класса `SortedList`, при вызове метода `ArrayList.Sort(IComparer)`, а также в ряде других мест в классах коллекций. Главное преимущество применения интерфейса `IComparer` заключается в том, что сортировке подлежат объекты тех классов, в которых интерфейс `IComparable` не реализуется.

Приведенный ниже пример программы является вариантом рассматривавшегося ранее необобщенного примера программы учета товарных запасов, переделанного с целью воспользоваться интерфейсом `IComparer` для сортировки перечня товарных запасов. В этом варианте программы сначала создается класс `CompInv`, в котором реализуется интерфейс `IComparer` и сравниваются два объекта класса `Inventory`. А затем объект класса `CompInv` указывается в вызове метода `Sort()` для сортировки перечня товарных запасов.

```
// Использовать необобщенный вариант интерфейса IComparer.

using System;
using System.Collections;
```

```
// Создать объект типа IComparer для объектов класса Inventory.
class CompInv : IComparer {
  // Реализовать интерфейс IComparer.
  public int Compare(object x, object y) {
    Inventory a, b;
    a = (Inventory) x;
    b = (Inventory) y;
    return string.Compare(a.name, b.name, StringComparison.Ordinal);
  }
}

class Inventory {
  public string name;
  double cost;
  int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10} Цена: {1,6:C} В наличии: {2}",
                    name, cost, onhand);
  }
}

class IComparerDemo {
  static void Main() {
    CompInv comp = new CompInv();
    ArrayList inv = new ArrayList();

    // Добавить элементы в список.
    inv.Add(new Inventory("Кусачки", 5.95, 3));
    inv.Add(new Inventory("Отвертки", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Дрели", 19.88, 8));

    Console.WriteLine("Перечень товарных запасов до сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
    Console.WriteLine();

    // Отсортировать список, используя интерфейс IComparer.
    inv.Sort(comp);

    Console.WriteLine("Перечень товарных запасов после сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
  }
}
```

Эта версия программы дает такой же результат, как и предыдущая.

Применение обобщенного интерфейса IComparer<T>

Интерфейс IComparer<T> является обобщенным вариантом интерфейса IComparer. В нем определяется приведенный ниже обобщенный вариант метода Compare().

```
int Compare(T x, T y)
```

В этом методе сравниваются объекты *x* и *y* и возвращается нулевое значение, если значения сравниваемых объектов равны; положительное — если значение объекта *x* больше, чем у объекта *y*; и отрицательное — если значение объекта *x* меньше, чем у объекта *y*.

Ниже приведена обобщенная версия предыдущей программы учета товарных запасов, в которой теперь используется интерфейс IComparer<T>. Она дает такой же результат, как и необобщенная версия этой же программы.

```
// Использовать обобщенный вариант интерфейса IComparer<T>.

using System;
using System.Collections.Generic;

// Создать объект типа IComparer<T> для объектов класса Inventory.
class CompInv<T> : IComparer<T> where T : Inventory {

  // Реализовать интерфейс IComparer<T>.
  public int Compare(T x, T y) {
    return string.Compare(x.name, y.name, StringComparison.Ordinal);
  }
}

class Inventory {
  public string name;
    double cost;
    int onhand;

  public Inventory(string n, double c, int h) {
    name = n;
    cost = c;
    onhand = h;
  }

  public override string ToString() {
    return
      String.Format("{0,-10} Цена: {1,6:C} В наличии: {2}",
                    name, cost, onhand);
  }
}

class GenericIComparerDemo {
  static void Main() {
    CompInv<Inventory> comp = new CompInv<Inventory>();
    List<Inventory> inv = new List<Inventory>();

    // Добавить элементы в список.
    inv.Add(new Inventory("Кусачки", 5.95, 3));
```

```
    inv.Add(new Inventory("Отвертки", 8.29, 2));
    inv.Add(new Inventory("Молотки", 3.50, 4));
    inv.Add(new Inventory("Дрели", 19.88, 8));

    Console.WriteLine("Перечень товарных запасов до сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
    Console.WriteLine();

    // Отсортировать список, используя интерфейс IComparer.
    inv.Sort(comp);

    Console.WriteLine("Перечень товарных запасов после сортировки:");
    foreach(Inventory i in inv) {
      Console.WriteLine("    " + i);
    }
  }
}
```

Применение класса StringComparer

В простых примерах из этой главы указывать явно способ сравнения символьных строк совсем не обязательно. Но это может потребоваться в тех случаях, когда строки сохраняются в отсортированной коллекции или когда строки ищутся либо сортируются в коллекции. Так, если строки должны быть отсортированы с учетом настроек одной культурной среды, а затем их приходится искать с учетом настроек другой культурной среды, то во избежание ошибок, вероятнее всего, потребуется указать способ сравнения символьных строк. Аналогичная ситуация возникает и при хешировании коллекции. Для подобных (и других) случаев в конструкторах классов некоторых коллекций предусмотрена поддержка параметра типа `IComparer`. С целью явно указать способ сравнения символьных строк этому параметру передается в качестве аргумента экземпляр объекта класса `StringComparer`.

Класс `StringComparer` был подробно описан в главе 21 при рассмотрении вопросов сортировки и поиска в массивах. В этом классе реализуются интерфейсы `IComparer`, `IComparer<String>`, `IEqualityComparer`, а также `IEqualityComparer<String>`. Следовательно, экземпляр объекта типа `StringComparer` может быть передан параметру типа `IComparer` в качестве аргумента. В классе `StringComparer` определяется несколько доступных только для чтения свойств, возвращающих экземпляр объекта типа `StringComparer`, который поддерживает различные способы сравнения символьных строк. Как пояснялось в главе 21, к числу этих свойств относятся следующие: `CurrentCulture`, `CurrentCultureIgnoreCase`, `InvariantCulture`, `InvariantCultureIgnoreCase`, `Ordinal`, а также `OrdinalIgnoreCase`. Все эти свойства можно использовать для явного указания способа сравнения символьных строк.

В качестве примера ниже показано, как коллекция типа `SortedList<TKey, TValue>` конструируется для хранения символьных строк, ключи которых сравниваются порядковым способом.

```
SortedList<string, int> users =
          new SortedList<string, int>(StringComparer.Ordinal);
```

Доступ к коллекции с помощью перечислителя

К элементам коллекции нередко приходится обращаться циклически, например, для отображения каждого элемента коллекции. С этой целью можно, с одной стороны, организовать цикл `foreach`, как было показано в приведенных выше примерах, а с другой — воспользоваться перечислителем. *Перечислитель* — это объект, который реализует необобщенный интерфейс `IEnumerator` или обобщенный интерфейс `IEnumerator<T>`.

В интерфейсе `IEnumerator` определяется одно свойство, `Current`, необобщенная форма которого приведена ниже.

```
object Current { get; }
```

А в интерфейсе `IEnumerator<T>` объявляется следующая обобщенная форма свойства `Current`.

```
T Current { get; }
```

В обеих формах свойства `Current` получается текущий перечисляемый элемент коллекции. Но поскольку свойство `Current` доступно только для чтения, то перечислитель может служить только для извлечения, но не видоизменения объектов в коллекции.

В интерфейсе `IEnumerator` определяются два метода. Первым из них является метод `MoveNext()`, объявляемый следующим образом.

```
bool MoveNext()
```

При каждом вызове метода `MoveNext()` текущее положение перечислителя смещается к следующему элементу коллекции. Этот метод возвращает логическое значение `true`, если следующий элемент коллекции доступен, и логическое значение `false`, если достигнут конец коллекции. Перед первым вызовом метода `MoveNext()` значение свойства `Current` оказывается неопределенным. (В принципе до первого вызова метода `MoveNext()` перечислитель обращается к несуществующему элементу, который должен находиться перед первым элементом коллекции. Именно поэтому приходится вызывать метод `MoveNext()`, чтобы перейти к первому элементу коллекции.)

Для установки перечислителя в исходное положение, соответствующее началу коллекции, вызывается приведенный ниже метод `Reset()`.

```
void Reset()
```

После вызова метода `Reset()` перечисление вновь начинается с самого начала коллекции. Поэтому, прежде чем получить первый элемент коллекции, следует вызвать метод `MoveNext()`.

В интерфейсе `IEnumerator<T>` методы `MoveNext()` и `Reset()` действуют по тому же самому принципу.

Необходимо также обратить внимание на два следующих момента. Во-первых, перечислитель нельзя использовать для изменения содержимого перечисляемой с его помощью коллекции. Следовательно, перечислители действуют по отношению к коллекции как к доступной только для чтения. И во-вторых, любое изменение в перечисляемой коллекции делает перечислитель недействительным.

Применение обычного перечислителя

Прежде чем получить доступ к коллекции с помощью перечислителя, необходимо получить его. В каждом классе коллекции для этой цели предоставляется метод `GetEnumerator()`, возвращающий перечислитель в начало коллекции. Используя этот перечислитель, можно получить доступ к любому элементу коллекции по очереди. В целом, для циклического обращения к содержимому коллекции с помощью перечислителя рекомендуется придерживаться приведенной ниже процедуры.

1. Получить перечислитель, устанавливаемый в начало коллекции, вызвав для этой коллекции метод `GetEnumerator()`.
2. Организовать цикл, в котором вызывается метод `MoveNext()`. Повторять цикл до тех пор, пока метод `MoveNext()` возвращает логическое значение `true`.
3. Получить в цикле каждый элемент коллекции с помощью свойства `Current`.

Ниже приведен пример программы, в которой реализуется данная процедура. В этой программе используется класс `ArrayList`, но общие принципы циклического обращения к элементам коллекции с помощью перечислителя остаются неизменными для коллекций любого типа, в том числе и обобщенных.

```
// Продемонстрировать применение перечислителя.

using System;
using System.Collections;

class EnumeratorDemo {
  static void Main() {
    ArrayList list = new ArrayList(1);

    for(int i=0; i < 10; i++)
      list.Add(i);

    // Использовать перечислитель для доступа к списку.
    IEnumerator etr = list.GetEnumerator();
    while(etr.MoveNext())
      Console.Write(etr.Current + " ");

    Console.WriteLine();

    // Повторить перечисление списка.
    etr.Reset();
    while(etr.MoveNext())
      Console.Write(etr.Current + " ");

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Вообще говоря, для циклического обращения к элементам коллекции цикл `foreach` оказывается более удобным, чем перечислитель. Тем не менее перечислитель предоставляет больше возможностей для управления, поскольку его можно при желании всегда установить в исходное положение.

Применение перечислителя типа `IDictionaryEnumerator`

Если для организации коллекции в виде словаря, например типа `Hashtable`, реализуется необобщенный интерфейс `IDictionary`, то для циклического обращения к элементам такой коллекции следует использовать перечислитель типа `IDictionaryEnumerator` вместо перечислителя типа `IEnumerator`. Интерфейс `IDictionaryEnumerator` наследует от интерфейса `IEnumerator` и имеет три дополнительных свойства. Первым из них является следующее свойство.

```
DictionaryEntry Entry { get; }
```

Свойство `Entry` позволяет получить пару "ключ–значение" из перечислителя в форме структуры `DictionaryEntry`. Напомним, что в структуре `DictionaryEntry` определяются два свойства, `Key` и `Value`, с помощью которых можно получать доступ к ключу или значению, связанному с элементом коллекции. Ниже приведены два других свойства, определяемых в интерфейсе `IDictionaryEnumerator`.

```
object Key { get; }
object Value { get; }
```

С помощью этих свойств осуществляется непосредственный доступ к ключу или значению.

Перечислитель типа `IDictionaryEnumerator` используется аналогично обычному перечислителю, за исключением того, что текущее значение в данном случае получается с помощью свойств `Entry`, `Key` или `Value`, а не свойства `Current`. Следовательно, приобретя перечислитель типа `IDictionaryEnumerator`, необходимо вызвать метод `MoveNext()`, чтобы получить первый элемент коллекции. А для получения остальных ее элементов следует продолжить вызовы метода `MoveNext()`. Этот метод возвращает логическое значение `false`, когда в коллекции больше нет ни одного элемента.

В приведенном ниже примере программы элементы коллекции типа `Hashtable` перечисляются с помощью перечислителя типа `IDictionaryEnumerator`.

```
// Продемонстрировать применение перечислителя типа IDictionaryEnumerator.

using System;
using System.Collections;

class IDicEnumDemo {
  static void Main() {
    // Создать хеш-таблицу.
    Hashtable ht = new Hashtable();

    // Добавить элементы в таблицу.
    ht.Add("Кен", "555-7756");
    ht.Add("Мэри", "555-9876");
    ht.Add("Том", "555-3456");
    ht.Add("Тодд", "555-3452");
```

```
    // Продемонстрировать применение перечислителя.
    IDictionaryEnumerator etr = ht.GetEnumerator();
    Console.WriteLine("Отобразить информацию с помощью свойства Entry.");
    while(etr.MoveNext())
      Console.WriteLine(etr.Entry.Key + ": " + etr.Entry.Value);

    Console.WriteLine();

    Console.WriteLine("Отобразить информацию " +
                      "с помощью свойств Key и Value.");
    etr.Reset();
    while(etr.MoveNext())
      Console.WriteLine(etr.Key + ": " + etr.Value);
  }
}
```

Ниже приведен результат выполнения этой программы.

```
Отобразить информацию с помощью свойства Entry.
Мэри: 555-9876
Том: 555-3456
Тодд: 555-3452
Кен: 555-7756

Отобразить информацию с помощью свойств Key и Value.
Мэри: 555-9876
Том: 555-3456
Тодд: 555-3452
Кен: 555-7756
```

Реализация интерфейсов `IEnumerable` и `IEnumerator`

Как упоминалось выше, для циклического обращения к элементам коллекции зачастую проще (да и лучше) организовать цикл `foreach`, чем пользоваться непосредственно методами интерфейса `IEnumerator`. Тем не менее ясное представление о принципе действия подобных интерфейсов важно иметь по еще одной причине: если требуется создать класс, содержащий объекты, перечисляемые в цикле `foreach`, то в этом классе следует реализовать интерфейсы `IEnumerator` и `IEnumerable`. Иными словами, для того чтобы обратиться к объекту определяемого пользователем класса в цикле `foreach`, необходимо реализовать интерфейсы `IEnumerator` и `IEnumerable` в их обобщенной или необобщенной форме. Правда, сделать это будет нетрудно, поскольку оба интерфейса не очень велики.

В приведенном ниже примере программы интерфейсы `IEnumerator` и `IEnumerable` реализуются в необобщенной форме, с тем чтобы перечислить содержимое массива, инкапсулированного в классе `MyClass`.

```
// Реализовать интерфейсы IEnumerable и IEnumerator.

using System;
using System.Collections;
```

```
class MyClass : IEnumerator, IEnumerable {
  char[] chrs = { 'A', 'B', 'C', 'D' };
  int idx = -1;

  // Реализовать интерфейс IEnumerable.
  public IEnumerator GetEnumerator() {
    return this;
  }

  // В следующих методах реализуется интерфейс IEnumerator

  // Возвратить текущий объект.
  public object Current {
    get {
      return chrs[idx];
    }
  }

  // Перейти к следующему объекту.
  public bool MoveNext() {
    if(idx == chrs.Length-1) {
      Reset(); // установить перечислитель в конец
      return false;
    }

    idx++;
    return true;
  }

  // Установить перечислитель в начало.
  public void Reset() { idx = -1; }
  }

class EnumeratorImplDemo {
  static void Main() {
    MyClass mc = new MyClass();

    // Отобразить содержимое объекта mc.
    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();

    // Вновь отобразить содержимое объекта mc.
    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

Эта программа дает следующий результат.

```
A B C D
A B C D
```

В данной программе сначала создается класс `MyClass`, в котором инкапсулируется небольшой массив типа `char`, состоящий из символов A–D. Индекс этого массива хранится в переменной `idx`, инициализируемой значением –1. Затем в классе `MyClass` реализуются оба интерфейса, `IEnumerator` и `IEnumerable`. Метод `GetEnumerator()` возвращает ссылку на перечислитель, которым в данном случае оказывается текущий объект. Свойство `Current` возвращает следующий символ в массиве, т.е. объект, указываемый по индексу `idx`. Метод `MoveNext()` перемещает индекс `idx` в следующее положение. Этот метод возвращает логическое значение `false`, если достигнут конец коллекции, в противном случае — логическое значение `true`. Напомним, что перечислитель оказывается неопределенным вплоть до первого вызова метода `MoveNext()`. Следовательно, метод `MoveNext()` автоматически вызывается в цикле `foreach` перед обращением к свойству `Current`. Именно поэтому первоначальное значение переменной `idx` устанавливается равным –1. Оно становится равным нулю на первом шаге цикла `foreach`. Обобщенная реализация рассматриваемых здесь интерфейсов будет действовать по тому же самому принципу.

Далее в методе `Main()` создается объект `mc` типа `MyClass`, и содержимое этого объекта дважды отображается в цикле `foreach`.

Применение итераторов

Как следует из предыдущих примеров, реализовать интерфейсы `IEnumerator` и `IEnumerable` нетрудно. Но еще проще воспользоваться *итератором*, который представляет собой метод, оператор или аксессор, возвращающий по очереди члены совокупности объектов от ее начала и до конца. Так, если некоторый массив состоит из пяти элементов, то итератор данного массива возвратит все эти элементы по очереди. Реализовав итератор, можно обращаться к объектам определяемого пользователем класса в цикле `foreach`.

Обратимся сначала к простому примеру итератора. Приведенная ниже программа является измененной версией предыдущей программы, в которой вместо явной реализации интерфейсов `IEnumerator` и `IEnumerable` применяется итератор.

```
// Простой пример применения итератора.

using System;
using System.Collections;

class MyClass {
  char[] chrs = { 'A', 'B', 'C', 'D' };

  // Этот итератор возвращает символы из массива chrs.
  public IEnumerator GetEnumerator() {
  foreach(char ch in chrs)
    yield return ch;
  }
}

class ItrDemo {
  static void Main() {
    MyClass mc = new MyClass();
```

```
    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

При выполнении этой программы получается следующий результат.

```
A B C D
```

Как видите, содержимое массива `mc.chrs` перечислено.

Рассмотрим эту программу более подробно. Во-первых, обратите внимание на то, что в классе `MyClass` не указывается `IEnumerator` в качестве реализуемого интерфейса. При создании итератора компилятор реализует этот интерфейс автоматически. И во-вторых, обратите особое внимание на метод `GetEnumerator()`, который ради удобства приводится ниже еще раз.

```
// Этот итератор возвращает символы из массива chrs.
public IEnumerator GetEnumerator() {
  foreach(char ch in chrs)
    yield return ch;
}
```

Это и есть итератор для объектов класса `MyClass`. Как видите, в нем явно реализуется метод `GetEnumerator()`, определенный в интерфейсе `IEnumerable`. А теперь перейдем непосредственно к телу данного метода. Оно состоит из цикла `foreach`, в котором возвращаются элементы из массива `chrs`. И делается это с помощью оператора `yield return`. Этот оператор возвращает следующий объект в коллекции, которым в данном случае оказывается очередной символ в массиве `chrs`. Благодаря этому средству обращение к объекту `mc` типа `MyClass` организуется в цикле `foreach` внутри метода `Main()`.

Обозначение `yield` служит в языке C# в качестве *контекстного ключевого слова*. Это означает, что оно имеет специальное назначение только в блоке итератора. А вне этого блока оно может быть использовано аналогично любому другому идентификатору.

Следует особо подчеркнуть, что итератор не обязательно должен опираться на массив или коллекцию другого типа. Он должен просто возвращать следующий элемент из совокупности элементов. Это означает, что элементы могут быть построены динамически с помощью соответствующего алгоритма. В качестве примера ниже приведена версия предыдущей программы, в которой возвращаются все буквы английского алфавита, набранные в верхнем регистре. Вместо массива буквы формируются в цикле `for`.

```
// Пример динамического построения значений,
// возвращаемых по очереди с помощью итератора.

using System;
using System.Collections;

class MyClass {
  char ch = 'A';

  // Этот итератор возвращает буквы английского
  // алфавита, набранные в верхнем регистре.
```

```
  public IEnumerator GetEnumerator() {
    for(int i=0; i < 26; i++)
      yield return (char) (ch + i);
  }
}

class ItrDemo2 {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Прерывание итератора

Для преждевременного прерывания итератора служит следующая форма оператора `yield`.

```
yield break;
```

Когда этот оператор выполняется, итератор уведомляет о том, что достигнут конец коллекции. А это, по существу, останавливает сам итератор.

Приведенная ниже программа является версией предыдущей программы, измененной с целью отобразить только первые десять букв английского алфавита.

```
// Пример прерывания итератора.

using System;
using System.Collections;

class MyClass {
  char ch = 'A';

  // Этот итератор возвращает первые 10 букв английского алфавита.
  public IEnumerator GetEnumerator() {
    for(int i=0; i < 26; i++) {
      if(i == 10) yield break; // прервать итератор преждевременно
      yield return (char) (ch + i);
    }
  }
}

class ItrDemo3 {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
```

```
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

Эта программа дает следующий результат.

```
A B C D E F G H I J
```

Применение нескольких операторов `yield`

В итераторе допускается применение нескольких операторов `yield`. Но каждый такой оператор должен возвращать следующий элемент в коллекции. В качестве примера рассмотрим следующую программу.

```
// Пример применения нескольких операторов yield.

using System;
using System.Collections;

class MyClass {
  // Этот итератор возвращает буквы A, B, C, D и E.
  public IEnumerator GetEnumerator() {
    yield return 'A';
    yield return 'B';
    yield return 'C';
    yield return 'D';
    yield return 'E';
  }
}

class ItrDemo5 {
  static void Main() {
    MyClass mc = new MyClass();

    foreach(char ch in mc)
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

Ниже приведен результата выполнения этой программы.

```
A B C D E
```

В данной программе внутри метода `GetEnumerator()` выполняются пять операторов `yield`. Следует особо подчеркнуть, что они выполняются по очереди и каждый раз, когда из коллекции получается очередной элемент. Таким образом, на каждом шаге цикла `foreach` в методе `Main()` возвращается только один символ.

Создание именованного итератора

В приведенных выше примерах был продемонстрирован простейший способ реализации итератора. Но ему имеется альтернатива в виде именованного итератора. В данном случае создается метод, оператор или аксессор, возвращающий ссылку на

объект типа IEnumerable. Именно этот объект используется в коде для предоставления итератора. Именованный итератор представляет собой метод, общая форма которого приведена ниже:

```
public IEnumerable имя_итератора(список_параметров) {
  // ...
  yield return obj;
}
```

где *имя_итератора* обозначает конкретное имя метода; *список_параметров* — от нуля до нескольких параметров, передаваемых методу итератора; *obj* — следующий объект, возвращаемый итератором. Как только именованный итератор будет создан, его можно использовать везде, где он требуется, например для управления циклом foreach.

Именованные итераторы оказываются весьма полезными в некоторых ситуациях, поскольку они позволяют передавать аргументы итератору, управляющему процессом получения конкретных элементов из коллекции. Например, итератору можно передать начальный и конечный пределы совокупности элементов, возвращаемых из коллекции итератором. Эту форму итератора можно перегрузить, расширив ее функциональные возможности. В приведенном ниже примере программы демонстрируются два способа применения именованного итератора для получения элементов коллекции. В одном случае элементы перечисляются в заданных начальном и конечном пределах, а в другом — элементы перечисляются с начала последовательности и до указанного конечного предела.

```
// Использовать именованные итераторы.

using System;
using System.Collections;

class MyClass {
  char ch = 'A';

  // Этот итератор возвращает буквы английского алфавита,
  // начиная с буквы A и кончая указанным конечным пределом.
  public IEnumerable MyItr(int end) {
    for(int i=0; i < end; i++)
      yield return (char) (ch + i);
  }

  // Этот итератор возвращает буквы в заданных пределах.
  public IEnumerable MyItr(int begin, int end) {
    for(int i=begin; i < end; i++)
      yield return (char) (ch + i);
  }
}

class ItrDemo4 {
  static void Main() {
    MyClass mc = new MyClass();

    Console.WriteLine("Возвратить по очереди первые 7 букв:");
    foreach(char ch in mc.MyItr(7))
```

```
      Console.Write(ch + " ");

    Console.WriteLine("\n");

    Console.WriteLine("Возвратить по очереди буквы от F до L:");
    foreach(char ch in mc.MyItr(5, 12))
      Console.Write(ch + " ");

    Console.WriteLine();
  }
}
```

Эта программа дает следующий результат.

```
Возвратить по очереди первые 7 букв:
A B C D E F G

Возвратить по очереди буквы от F до L:
F G H I J K L
```

Создание обобщенного итератора

В приведенных выше примерах применялись необобщенные итераторы, но, конечно, ничто не мешает создать обобщенные итераторы. Для этого достаточно возвратить объект обобщенного типа `IEnumerator<T>` или `IEnumerable<T>`. Ниже приведен пример создания обобщенного итератора.

```
// Простой пример обобщенного итератора.

using System;
using System.Collections.Generic;

class MyClass<T> {
  T[] array;

  public MyClass(T[] a) {
    array = a;
  }

  // Этот итератор возвращает символы из массива chrs.
  public IEnumerator<T> GetEnumerator() {
    foreach(T obj in array)
      yield return obj;
  }
}

class GenericItrDemo {
  static void Main() {
    int[] nums = { 4, 3, 6, 4, 7, 9 };
    MyClass<int> mc = new MyClass<int>(nums);

    foreach(int x in mc)
      Console.Write(x + " ");
```

```
    Console.WriteLine();

    bool[] bVals = { true, true, false, true };
    MyClass<bool> mc2 = new MyClass<bool>(bVals);

    foreach(bool b in mc2)
      Console.Write(b + " ");

    Console.WriteLine();
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
4 3 6 4 7 9
True True False True
```

В данном примере массив, состоящий из возвращаемых по очереди объектов, передается конструктору класса `MyClass`. Тип этого массива указывает в качестве аргумента типа в конструкторе класса `MyClass`.

Метод `GetEnumerator()` оперирует данными обобщенного типа `T` и возвращает перечислитель типа `IEnumerator<T>`. Следовательно, итератор, определенный в классе `MyClass`, способен перечислять данные любого типа.

Инициализаторы коллекций

В C# имеется специальное средство, называемое *инициализатором коллекции* и упрощающее инициализацию некоторых коллекций. Вместо того чтобы явно вызывать метод `Add()`, при создании коллекции можно указать список инициализаторов. После этого компилятор организует автоматические вызовы метода `Add()`, используя значения из этого списка. Синтаксис в данном случае ничем не отличается от инициализации массива. Обратимся к следующему примеру, в котором создается коллекция типа `List<char>`, инициализируемая символами C, A, E, B, D и F.

```
List<char> lst = new List<char>() { 'C', 'A', 'E', 'B', 'D', 'F' };
```

После выполнения этого оператора значение свойства `lst.Count` будет равно 6, поскольку именно таково число инициализаторов. А после выполнения следующего цикла `foreach`:

```
foreach(ch in lst)
  Console.Write(ch + " ");
```

получится такой результат:

```
C A E B D F
```

Для инициализации коллекции типа `LinkedList<TKey, TValue>`, в которой хранятся пары "ключ–значение", инициализаторы приходится предоставлять парами, как показано ниже.

```
SortedList<int, string> lst =
    new SortedList<int, string>() { {1, "один"}, {2, "два" }, {3, "три"} };
```

Компилятор передаст каждую группу значений в качестве аргументов методу `Add()`. Следовательно, первая пара инициализаторов преобразуется компилятором в вызов `Add(1, "один")`.

Компилятор вызывает метод `Add()` автоматически для ввода инициализаторов в коллекцию, и поэтому инициализаторы коллекций можно использовать только в коллекциях, поддерживающих открытую реализацию метода `Add()`. Это означает, что инициализаторы коллекций нельзя использовать в коллекциях типа `Stack`, `Stack<T>`, `Queue` или `Queue<T>`, поскольку в них метод `Add()` не поддерживается. Их нельзя применять также в тех коллекциях типа `LinkedList<T>`, где метод `Add()` предоставляется как результат явной реализации соответствующего интерфейса.

26 ГЛАВА

Сетевые средства подключения к Интернету

Язык C# предназначен для программирования в современной вычислительной среде, где Интернету, естественно, принадлежит весьма важная роль. Одной из главных целей разработки C# было внедрение в этот язык программирования средств, необходимых для доступа к Интернету. Такой доступ можно было осуществить и в предыдущих версиях языков программирования, включая C и C++, но поддержка операций на стороне сервера, загрузка файлов и получение сетевых ресурсов в этих языках не вполне отвечали потребностям большинства программистов. Эта ситуация коренным образом изменилась в C#. Используя стандартные средства C# и среды .NET Framework, можно довольно легко сделать приложения совместимыми с Интернетом и написать другие виды программ, ориентированных на подключение к Интернету.

Поддержка сетевого подключения осуществляется через несколько пространств имен, определенных в среде .NET Framework, и главным среди них является пространство имен `System.Net`. В нем определяется целый ряд высокоуровневых, но простых в использовании классов, поддерживающих различные виды операций, характерных для работы с Интернетом. Для этих целей доступен также ряд пространств, вложенных в пространство имен `System.Net`. Например, средства низкоуровневого сетевого управления через сокеты находятся в пространстве имен `System.Net.Sockets`, поддержка электронной почты — в пространстве имен `System.Net.Mail`, а поддержка защищенных сетевых потоков — в пространстве имен `System.Net.Security`. Дополнительные функциональные возможности предоставляются в ряде других вложенных пространств имен. К числу других не менее важных пространств имен,

связанных с сетевым подключением к Интернету, относится пространство `System.Web`. Это и вложенные в него пространства имен поддерживают сетевые приложения на основе технологии ASP.NET.

В среде .NET Framework имеется достаточно гибких средств и возможностей для сетевого подключения к Интернету. Тем не менее для разработки многих приложений более предпочтительными оказываются функциональные возможности, доступные в пространстве имен `System.Net`. Они и удобны, и просты в использовании. Именно поэтому пространству имен `System.Net` будет уделено основное внимание в этой главе.

Члены пространства имен `System.Net`

Пространство имен `System.Net` довольно обширно и состоит из многих членов. Полное их описание и обсуждение всех аспектов программирования для Интернета выходит далеко за рамки этой главы. (На самом деле для подробного рассмотрения всех вопросов, связанных с сетевым подключением к Интернету и его поддержкой в C#, потребуется отдельная книга.) Однако целесообразно хотя бы перечислить члены пространства имен `System.Net`, чтобы дать какое-то представление о том, что именно доступно для использования в этом пространстве.

Ниже перечислены классы, определенные в пространстве имен `System.Net`.

`AuthenticationManager`	`Authorization`
`Cookie`	`CookieCollection`
`CookieContainer`	`CookieException`
`CredentialCache`	`Dns`
`DnsEndPoint`	`DnsPermission`
`DnsPermissionAttribute`	`DownloadDataCompletedEventArgs`
`DownloadProgressChangedEventArgs`	`DownloadStringCompletedEventArgs`
`EndPoint`	`EndpointPermission`
`FileWebRequest`	`FileWebResponse`
`FtpWebRequest`	`FtpWebResponse`
`HttpListener`	`HttpListenerBasicIdentity`
`HttpListenerContext`	`HttpListenerException`
`HttpListenerPrefixCollection`	`HttpListenerRequest`
`HttpListenerResponse`	`HttpVersion`
`HttpWebRequest`	`HttpWebResponse`
`IPAddress`	`IPEndPoint`
`IPEndPointCollection`	`IPHostEntry`
`IrDAEndPoint`	`NetworkCredential`
`OpenReadCompletedEventArgs`	`OpenWriteCompletedEventArgs`
`ProtocolViolationException`	`ServicePoint`
`ServicePointManager`	`SocketAddress`
`SocketPermission`	`SocketPermissionAttribute`
`TransportContext`	`UploadDataCompletedEventArgs`
`UploadFileCompletedEventArgs`	`UploadProgressChangedEventArgs`
`UploadStringCompletedEventArgs`	`UploadValuesCompletedEventArgs`
`WebClient`	`WebException`
`WebHeaderCollection`	`WebPermission`

WebPermissionAttribute	WebProxy
WebRequest	WebRequestMethods
WebRequestMethods.File	WebRequestMethods.Ftp
WebRequestMethods.Http	WebResponse
WebUtility	

Кроме того, в пространстве имен System.Net определены перечисленные ниже интерфейсы.

AuthenticationModule	IcertificatePolicy	ICredentialPolicy
ICredentials	IcredentialsByHost	IWebProxy
IWebProxyScript	IWebRequestCreate	

В этом пространстве имен определяются также приведенные ниже перечисления.

AuthenticationSchemes	DecompressionMethods	FtpStatusCode
HttpRequestHeader	HttpResponseHeader	HttpStatusCode
NetworkAccess	SecurityProtocolType	TransportType
WebExceptionStatus		

Помимо этого, в пространстве имен System.Net определен ряд делегатов.

Несмотря на то что в пространстве имен System.Net определено немало членов, лишь немногие из них на самом деле требуются при решении наиболее типичных задач программирования для Интернета. Основу сетевых программных средств составляют абстрактные классы WebRequest и WebResponse. От этих классов наследуют все классы, поддерживающие конкретные сетевые протоколы. (*Протокол* определяет правила передачи данных по сети.) Например, к производным классам, поддерживающим стандартный сетевой протокол HTTP, относятся классы HttpWebRequest и HttpWebResponse.

Классы HttpWebRequest и HttpWebResponse довольно просты в использовании. Тем не менее решение некоторых задач можно еще больше упростить, применяя подход, основанный на классе WebClient. Так, если требуется только загрузить или выгрузить файл, то для этой цели лучше всего подойдет класс WebClient.

Универсальные идентификаторы ресурсов

В основу программирования для Интернета положено понятие *универсального идентификатора ресурса* (URI), иногда еще называемого *унифицированным указателем информационного ресурса* (URL). Этот идентификатор описывает местоположение ресурса в сети. В корпорации Microsoft принято пользоваться сокращением URI при описании членов пространства имен System.Net, и поэтому в данной книге выбрано именно это сокращение для обозначения универсального идентификатора ресурса. Идентификаторы URI, без сомнения, известны каждому, кто хотя бы раз пользовался браузером для поиска информации в Интернете. По существу, это адрес информационного ресурса, который указывается в соответствующем поле окна браузера.

Ниже приведена общая форма идентификатора URI:

Протокол://*Идентификационный_номер_сервера*/*Путь_к_файлу*?*Запрос*

где *Протокол* — это применяемый протокол, например HTTP; *Идентификационный_номер_сервера* — конкретный сервер, например mhprofessional.com или

HerbSchildt.com; *Путь_к_файлу* — путь к конкретному файлу. Если же *Путь_к_файлу* не указан, то получается страница, доступная на указанном сервере по умолчанию. И наконец, Запрос обозначает информацию, отправляемую на сервер. Указывать Запрос необязательно. В C# идентификаторы URI инкапсулированы в класс Uri, рассматриваемый далее в этой главе.

Основы организации доступа к Интернету

В классах, находящихся в пространстве имен System.Net, поддерживается модель взаимодействия с Интернетом по принципу запроса и ответа. При таком подходе пользовательская программа, являющаяся клиентом, запрашивает информацию у сервера, а затем переходит в состояние ожидания ответа. Например, в качестве запроса программа может отправить на сервер идентификатор URI некоторого веб-сайта. В ответ она получит гипертекстовую страницу, соответствующую указанному идентификатору URI. Такой принцип запроса и ответа удобен и прост в применении, поскольку большинство деталей сетевого взаимодействия реализуются автоматически.

На вершине иерархии сетевых классов находятся классы WebRequest и WebResponse, реализующие так называемые *подключаемые протоколы*. Как должно быть известно большинству читателей, для передачи данных в сети применяется несколько разнотипных протоколов. К числу наиболее распространенных в Интернете относятся протокол передачи гипертекстовых файлов (HTTP), а также протокол передачи файлов (FTP). При создании идентификатора URI его префикс обозначает применяемый сетевой протокол. Например, в идентификаторе http://www.HerbSchildt.com используется префикс http, обозначающий протокол передачи гипертекстовых файлов (HTTP).

Как упоминалось выше, классы WebRequest и WebResponse являются абстрактными, а следовательно, в них определенны в самом общем виде операции запроса и ответа, типичные для всех протоколов. От этих классов наследуют более конкретные производные классы, в которых реализуются отдельные протоколы. Эти производные классы регистрируются самостоятельно, используя для этой цели статический метод RegisterPrefix(), определенный в классе WebRequest. При создании объекта типа WebRequest автоматически используется протокол, указываемый в префиксе URI, если, конечно, он доступен. Преимущество такого принципа "подключения" протоколов заключается в том, что бо́льшая часть кода пользовательской программы остается без изменения независимо от типа применяемого протокола.

В среде выполнения .NET Runtime протоколы HTTP, HTTPS и FTP определяются автоматически. Так, если указать идентификатор URI с префиксом HTTP, то будет автоматически получен HTTP-совместимый класс, который поддерживает протокол HTTP. А если указать идентификатор URI с префиксом FTP, то будет автоматически получен FTP-совместимый класс, поддерживающий протокол FTP.

При сетевом подключении к Интернету чаще всего применяется протокол HTTP, поэтому именно он и рассматривается главным образом в этой главе. (Тем не менее аналогичные приемы распространяются и на все остальные поддерживаемые протоколы.) Протокол HTTP поддерживается в классах HttpWebRequest и HttpWebResponse. Эти классы наследуют от классов WebRequest и WebResponse, а кроме того, имеют собственные дополнительные члены, применимые непосредственно к протоколу HTTP.

В пространстве имен `System.Net` поддерживается как синхронная, так и асинхронная передача данных. В Интернете предпочтение чаще всего отдается синхронным транзакциям, поскольку ими легче пользоваться. При синхронной передаче данных пользовательская программа посылает запрос и затем ожидает ответа от сервера. Но для некоторых разновидностей высокопроизводительных приложений более подходящей оказывается асинхронная передача данных. При таком способе передачи данных пользовательская программа продолжает обработку данных, ожидая ответа на переданный запрос. Но организовать асинхронную передачу данных труднее. Кроме того, не во всех программах можно извлечь выгоды из асинхронной передачи данных. Например, когда требуется получить информацию из Интернета, то зачастую ничего другого не остается, как ожидать ее. В подобных случаях потенциал асинхронной передачи данных используется не полностью. Вследствие того что синхронный доступ к Интернету реализуется проще и намного чаще, именно он и будет рассматриваться в этой главе.

Далее речь пойдет прежде всего о классах `WebRequest` и `WebResponse`, поскольку именно они положены в основу сетевых программных средств, доступных в пространстве имен `System.Net`.

Класс `WebRequest`

Класс `WebRequest` управляет сетевым запросом. Он является абстрактным, поскольку в нем не реализуется конкретный протокол. Тем не менее в нем определяются те методы и свойства, которые являются общими для всех сетевых запросов. В табл. 26.1 сведены методы, определенные в классе `WebRequest` и поддерживающие синхронную передачу данных, а в табл. 26.2 — свойства, объявляемые в классе `WebRequest`. Устанавливаемые по умолчанию значения свойств задаются в производных классах. Открытые конструкторы в классе `WebRequest` не определены.

Для того чтобы отправить запрос по адресу URI, необходимо сначала создать объект класса, производного от класса `WebRequest` и реализующего требуемый протокол. С этой целью вызывается статический метод `Create()`, определенный в классе `WebRequest`. Метод `Create()` возвращает объект класса, наследующего от класса `WebRequest` и реализующего конкретный протокол.

Таблица 26.1. Методы, определенные в классе `WebRequest`

Метод	Описание
`public static WebRequest Create(string` *`requestUriString`*`)`	Создает объект типа `WebRequest` для идентификатора URI, указываемого в строке *`requestUriString`*. Возвращаемый объект реализует протокол, заданный префиксом идентификатора URI. Следовательно, возвращаемый объект будет экземпляром класса, призводного от класса `WebRequest`. Если затребованный протокол недоступен, то генерируется исключение `NotSupportedException`. А если недействителен указанный формат идентификатора URI, то генерируется исключение `UriFormatException`

Окончание табл. 26.1

Метод	Описание
`public static WebRequest Create(Uri requestUri)`	Создает объект типа `WebRequest` для идентификатора URI, указываемого с помощью параметра *`requestUri`*. Возвращаемый объект реализует протокол, заданный префиксом идентификатора URI. Следовательно, возвращаемый объект будет экземпляром класса, призводного от класса `WebRequest`. Если затребованный протокол недоступен, то генерируется исключение `NotSupportedException`
`public virtual Stream GetRequestStream()`	Возвращает поток вывода, связанный с запрошенным ранее идентификатором URI
`public virtual WebResponse GetResponse()`	Отправляет предварительно сформированный запрос и ожидает ответа. Получив ответ, возвращает его в виде объекта класса `WebReponse`. Этот объект используется затем в программе для получения информации по указанному адресу URI

Таблица 26.2. Свойства, определенные в классе `WebRequest`

Свойство	Описание
`public AuthenticationLevel AuthenticationLevel{ get; set; }`	Получает или устанавливает уровень аутентификации
`public virtual RequestCachePolicy CachePolicy { get; set; }`	Получает или устанавливает правила использования кеша, определяющие момент получения ответа из кеша
`public virtual string ConnectionGroupName { get; set; }`	Получает или устанавливает имя группы подключения. Группы подключения представляют собой способ создания ряда запросов. Они не нужны для простых транзакций в Интернете
`public virtual long ContentLength { get; set; }`	Получает или устанавливает длину передаваемого содержимого
`public virtual string ContentType { get; set; }`	Получает или устанавливает описание передаваемого содержимого
`public virtual Icredentials Credentials { get; set; }`	Получает или устанавливает мандат, т.е. учетные данные пользователя
`public static RequestCachePolicy DefaultCachePolicy { get; set; }`	Получает или устанавливает правила использования кеша по умолчанию, определяющие момент получения ответа из кеша
`public static IWebProxy DefaultWebProxy { get; set; }`	Получает или устанавливает используемый по умолчанию прокси-сервер
`public virtual WebHeaderCollection Headers{ get; set; }`	Получает или устанавливает коллекцию заголовков
`public TokenImpersonationLevel ImpersonationLevel { get; set; }`	Получает или устанавливает уровень анонимного воплощения

Окончание табл. 26.2

Свойство	Описание
`public virtual string Method { get; set; }`	Получает или устанавливает протокол
`public virtual bool PreAuthenticate { get; set; }`	Если принимает логическое значение `true`, то в отправляемый запрос включается информация для аутентификации. А если принимает логическое значение `false`, то информация для аутентификации предоставляется только по требованию адресата URI
`public virtual IWebProxy Proxy { get; set; }`	Получает или устанавливает прокси-сервер. Применимо только в тех средах, где используется прокси-сервер
`public virtual Uri RequestUri { get; }`	Получает идентификатор URI конкретного запроса
`public virtual int Timeout { get; set; }`	Получает или устанавливает количество миллисекунд, в течение которых будет ожидаться ответ на запрос. Для установки бесконечного ожидания используется значение `Timeout.Infinite`
`public virtual bool UseDefaultCredential { get; set; }`	Получает или устанавливает значение, которое определяет, используется ли для аутентификации устанавливаемый по умолчанию мандат. Если имеет логическое значение `true`, то используется устанавливаемый по умолчанию мандат, т.е. учетные данные пользователя, в противном случае этот мандат не используется

Класс WebResponse

В классе `WebResponse` инкапсулируется ответ, получаемый по запросу. Этот класс является абстрактным. В наследующих от него классах создаются отдельные его версии, поддерживающие конкретный протокол. Объект класса `WebResponse` обычно получается в результате вызова метода `GetResponse()`, определенного в классе `WebRequest`. Этот объект будет экземпляром отдельного класса, производного от класса `WebResponse` и реализующего конкретный протокол. Методы, определенные в классе `WebResponse`, сведены в табл. 26.3, а свойства, объявляемые в этом классе, — в табл. 26.4. Значения этих свойств устанавливаются на основании каждого запроса в отдельности. Открытые конструкторы в классе `WebResponse` не определяются.

Таблица 26.3. Наиболее часто используемые методы, определенные в классе WebResponse

Метод	Описание
`public virtual void Close()`	Закрывает ответный поток. Закрывает также поток ввода ответа, возвращаемый методом `GetResponseStream()`
`public virtual Stream GetResponseStream()`	Возвращает поток ввода, связанный с запрашиваемым URI. Из этого потока могут быть введены данные из запрашиваемого URI

Таблица 26.4. Свойства, определенные в классе `WebResponse`

Свойство	Описание
`public virtual long ContentLength { get; set; }`	Получает или устанавливает длину принимаемого содержимого. Устанавливается равным -1, если данные о длине содержимого недоступны
`public virtual string ContentType { get; set; }`	Получает или устанавливает описание принимаемого содержимого
`public virtual WebHeaderCollection Headers { get; }`	Получает или устанавливает коллекцию заголовков, связанных с URI
`public virtual bool IsFromCache { get; }`	Принимает логическое значение `true`, если запрос получен из кэша. А если запрос доставлен по сети, то принимает логическое значение `false`
`public virtual bool IsMutuallyAuthenticated { get; }`	Принимает логическое значение `true`, если клиент и сервер опознают друг друга, а иначе — принимает логическое значение `false`
`public virtual Uri ResponseUri { get; }`	Получает URI, по которому был сформирован ответ. Этот идентификатор может отличаться от запрашиваемого, если ответ был переадресован по другому URI

Классы `HttpWebRequest` и `HttpWebResponse`

Оба класса, `HttpWebRequest` и `HttpWebResponse`, наследуют от классов `WebRequest` и `WebResponse` и реализуют протокол HTTP. В ходе этого процесса в обоих классах вводится ряд дополнительных свойств, предоставляющих подробные сведения о транзакции по протоколу HTTP. О некоторых из этих свойств речь пойдет далее в настоящей главе. Но для выполнения простых операций в Интернете эти дополнительные свойства, как правило, не требуются.

Первый простой пример

Доступ к Интернету организуется на основе классов `WebRequest` и `WebResponse`. Поэтому, прежде чем рассматривать этот процесс более подробно, было бы полезно обратиться к простому примеру, демонстрирующему порядок доступа к Интернету по принципу запроса и ответа. Глядя на то, как эти классы применяются на практике, легче понять, почему они организованы именно так, а не как-то иначе.

В приведенном ниже примере программы демонстрируется простая, но весьма типичная для Интернета операция получения гипертекстового содержимого из конкретного веб-сайта. В данном случае содержимое получается из веб-сайта издательства McGraw-Hill по адресу `www.McGraw-Hill.com`, но вместо него можно подставить адрес любого другого веб-сайта. В этой программе гипертекстовое содержимое выводится на экран монитора отдельными порциями по 400 символов, чтобы полученную информацию можно было просматривать, не прибегая к прокрутке экрана.

```
// Пример доступа к веб-сайту.

using System;
```

```
using System.Net;
using System.IO;

class NetDemo {
  static void Main() {
    int ch;

    // Сначала создать объект запроса типа WebRequest по указанному URI.
    HttpWebRequest req = (HttpWebRequest)
    WebRequest.Create("http://www.McGraw-Hill.com");

    // Затем отправить сформированный запрос и получить на него ответ.
    HttpWebResponse resp = (HttpWebResponse)
    req.GetResponse();

    // Получить из ответа поток ввода.
    Stream istrm = resp.GetResponseStream();

    /* А теперь прочитать и отобразить гипертекстовое содержимое,
       полученное по указанному URI. Это содержимое выводится на экран
       отдельными порциями по 400 символов. После каждой такой порции
       следует нажать клавишу <ENTER>, чтобы вывести на экран
       следующую порцию из 400 символов. */
    for(int i=1; ; i++) {
      ch = istrm.ReadByte();
      if(ch == -1) break;
        Console.Write((char) ch);
        if((i%400)==0) {
          Console.Write("\nНажмите клавишу <Enter>.");
          Console.ReadLine();
        }
    }

    // Закрыть ответный поток. При этом закрывается также поток ввода istrm.
    resp.Close();
  }
}
```

Ниже приведена первая часть получаемого результата. (Разумеется, это содержимое может со временем измениться в связи с обновлением запрашиваемого веб-сайта, и поэтому у вас оно может оказаться несколько иным.)

```
<html>
<head>
<title>Home - The McGraw-Hill Companies</title>
<meta name="keywords" content="McGraw-Hill Companies,McGraw-Hill, McGraw Hill,
Aviation Week, BusinessWeek, Standard and Poor's, Standard & Poor's,CTB/McGraw-
Hill,Glencoe/McGraw-Hill,The Grow Network/McGraw-Hill,Macmillan/McGraw-Hill,
McGraw-Hill Contemporary,McGraw-Hill Digital Learning,McGraw-Hill Professional
Development,SRA/McGraw

Нажмите клавишу <Enter>.
-Hill,Wright Group/McGraw-Hill,McGraw-Hill Higher Education,McGraw-Hill/Irwin,
McGraw-Hill/Primis Custom Publishing,McGraw-Hill/Ryerson,Tata/McGraw-Hill,
```

```
McGraw-Hill Interamericana,Open University Press, Healthcare Information Group,
Platts, McGraw-Hill Construction, Information & Media Services" />
<meta name="description" content="The McGraw-Hill Companies Corporate Website." />
<meta http-equiv

Нажмите клавишу <Enter>.
```

Итак, выше приведена часть гипертекстового содержимого, полученного из веб-сайта издательства McGraw-Hill по адресу `www.McGraw-Hill.com`. В рассматриваемом здесь примере программы это содержимое просто выводится в исходном виде на экран посимвольно и не форматируется в удобочитаемом виде, как это обычно делается в окне браузера.

Проанализируем данную программу построчно. Прежде всего обратите внимание на использование в ней пространства имен `System.Net`. Как пояснялось ранее, в этом пространстве имен находятся классы сетевого подключения к Интернету. Обратите также внимание на то, что в данную программу включено пространство имен `System.IO`, которое требуется для того, чтобы прочитать полученную на веб-сайте информацию, используя объект типа `Stream`.

В начале программы создается объект типа `WebRequest`, содержащий требуемый URI. Как видите, для этой цели используется метод `Create()`, а не конструктор. Это статический член класса `WebRequest`. Несмотря на то что класс `WebRequest` является абстрактным, это обстоятельство не мешает вызывать статический метод данного класса. Метод `Create()` возвращает объект типа `HttpWebRequest`. Разумеется, его значение требуется привести к типу `HttpWebRequest`, прежде чем присвоить его переменной `req` ссылки на объект типа `HttpWebRequest`. На этом формирование запроса завершается, но его еще нужно отправить по указанному URI.

Для того чтобы отправить запрос, в рассматриваемой здесь программе вызывается метод `GetResponse()` для объекта типа `WebRequest`. Отправив запрос, метод `GetResponse()` переходит в состояние ожидания ответа. Как только ответ будет получен, метод `GetResponse()` возвратит объект типа `WebResponse`, в котором инкапсулирован ответ. Этот объект присваивается переменной `resp`. Но в данном случае ответ принимается по протоколу HTTP, и поэтому полученный результат приводится к типу `HttpWebResponse`. Среди прочего в ответе содержится поток, предназначаемый для чтения данных из источника по указанному URI.

Далее поток ввода получается в результате вызова метода `GetResponseStream()` для объекта `resp`. Это стандартный объект класса `Stream` со всеми атрибутами и средствами, необходимыми для организации потока ввода. Ссылка на этот поток присваивается переменной `istrm`, с помощью которой данные могут быть прочитаны из источника по указанному URI, как из обычного файла.

После этого в программе выполняется чтение данных из веб-сайта издательства McGraw-Hill по адресу `www.McGraw-Hill.com` и последующий их вывод на экран. А поскольку этих данных много, то они выводятся на экран отдельными порциями по 400 символов, после чего в программе ожидается нажатие клавиши <Enter>, чтобы продолжить вывод. Благодаря этому выводимые данные можно просматривать без прокрутки экрана. Обратите внимание на то, что данные читаются посимвольно с помощью метода `ReadByte()`. Напомним, что этот метод возвращает очередной байт из потока ввода в виде значения типа `int`, которое требуется привести к типу `char`. По достижении конца потока этот метод возвращает значение –1.

И наконец, ответный поток закрывается при вызове метода `Close()` для объекта `resp`. Вместе с ответным потоком автоматически закрывается и поток ввода. Ответный

поток следует закрывать в промежутках между последовательными запросами. В противном случае сетевые ресурсы могут быть исчерпаны, препятствуя очередному подключению к Интернету.

И в заключение анализа рассматриваемого здесь примера следует обратить особое внимание на следующее: для отображения гипертекстового содержимого, получаемого от сервера, совсем не обязательно использовать объект типа `HttpWebRequest` или `HttpWebResponse`. Ведь для решения этой задачи в данной программе оказалось достаточно стандартных методов, определенных в классах `WebRequest` и `WebResponse`, и не потребовалось прибегать к специальным средствам протокола HTTP. Следовательно, вызовы методов `Create()` и `GetResponse()` можно было бы написать следующим образом.

```
// Сначала создать объект запроса типа WebRequest по указанному URI.
WebRequest req = WebRequest.Create("http://www.McGraw-Hill.com");

// Затем отправить сформированный запрос и получить на него ответ.
WebResponse resp = req.GetResponse();
```

В тех случаях, когда не требуется приведение к конкретному типу реализации протокола, лучше пользоваться классами `WebRequest` и `WebResponse`, так как это дает возможность менять протокол, не оказывая никакого влияния на код программы. Но поскольку во всех примерах, приведенных в этой главе, используется протокол HTTP, то в ряде примеров демонстрируются специальные средства этого протокола из классов `HttpWebRequest` и `HttpWebResponse`.

Обработка сетевых ошибок

Программа из предыдущего примера составлена верно, но она совсем не защищена от простейших сетевых ошибок, которые способны преждевременно прервать ее выполнение. Конечно, для программы, служащей в качестве примера, это не так важно, как для реальных приложений. Для полноценной обработки сетевых исключений, которые могут быть сгенерированы программой, необходимо организовать контроль вызовов методов `Create()`, `GetResponse()` и `GetResponseStream()`. Следует особо подчеркнуть, что генерирование конкретных исключений зависит от используемого протокола. И ниже речь пойдет об ошибках, которые могут возникнуть при использовании протокола HTTP, поскольку средства сетевого подключения к Интернету, доступные в C#, рассматриваются в настоящей главе на примере именно этого протокола.

Исключения, генерируемые методом `Create()`

Метод `Create()`, определенный в классе `WebRequest`, может генерировать четыре исключения. Так, если протокол, указываемый в префиксе URI, не поддерживается, то генерируется исключение `NotSupportedException`. Если формат URI оказывается недействительным, то генерируется исключение `UriFormatException`. А если у пользователя нет соответствующих полномочий для доступа к запрашиваемому сетевому ресурсу, то генерируется исключение `System.Security.SecurityException`. Кроме того, метод `Create()` генерирует исключение `ArgumentNullException`, если он вызывается с пустой ссылкой, хотя этот вид ошибки не имеет непосредственного отношения к сетевому подключению.

Исключения, генерируемые методом GetResponse()

При вызове метода GetResponse() для получения ответа по протоколу HTTP может произойти целый ряд ошибок. Эти ошибки представлены следующими исключениями: InvalidOperationException, ProtocolViolationException, NotSupportedException и WebException. Наибольший интерес среди них вызывает исключение WebException.

У исключения WebException имеются два свойства, связанных с сетевыми ошибками: Response и Status. С помощью свойства Response можно получить ссылку на объект типа WebResponse в обработчике исключений. Для соединения по протоколу HTTP этот объект описывает характер возникшей ошибки. Свойство Response объявляется следующим образом.

```
public WebResponse Response { get; }
```

Когда возникает ошибка, то с помощью свойства Status типа WebException можно выяснить, что именно произошло. Это свойство объявляется следующим образом:

```
public WebExceptionStatus Status {get; }
```

где WebExceptionStatus — это перечисление, которое содержит приведенные ниже значения.

CacheEntryNotFound	ConnectFailure	ConnectionClosed
KeepAliveFailure	MessageLengthLimitExceeded	NameResolutionFailure
Pending	PipelineFailure	ProtocolError
ProxyNameResolutionFailure	ReceiveFailure	RequestCanceled
RequestProhibitedByCachePolicy	RequestProhibitedByProxy	SecureChannelFailure
SendFailure	ServerProtocolViolation	Success
Timeout	TrustFailure	UnknownError

Как только будет выяснена причина ошибки, в программе могут быть предприняты соответствующие действия.

Исключения, генерируемые методом GetResponseStream()

Для соединения по протоколу HTTP метод GetResponseStream() из класса WebResponse может сгенерировать исключение ProtocolViolationException, которое в целом означает, что в работе по указанному протоколу произошла ошибка. Что же касается метода GetResponseStream(), то это означает, что ни один из действительных ответных потоков недоступен. Исключение ObjectDisposedException генерируется в том случае, если ответ уже утилизирован. А исключение IOException, конечно, генерируется при ошибке чтения из потока, в зависимости от того, как организован ввод данных.

Обработка исключений

В приведенном ниже примере программы демонстрируется обработка всевозможных сетевых исключений, которые могут возникнуть в связи с выполнением программы из предыдущего примера, в которую теперь добавлены соответствующие обработчики исключений.

```
// Пример обработки сетевых исключений.

using System;
using System.Net;
using System.IO;

class NetExcDemo {
  static void Main() {
    int ch;

    try {
      // Сначала создать объект запроса типа WebRequest по указанному URI.
      HttpWebRequest req = (HttpWebRequest)
      WebRequest.Create("http://www.McGraw-Hill.com");

      // Затем отправить сформированный запрос и получить на него ответ.
      HttpWebResponse resp = (HttpWebResponse)
      req.GetResponse();

      // Получить из ответа поток ввода.
      Stream istrm = resp.GetResponseStream();

      /* А теперь прочитать и отобразить гипертекстовое содержимое,
         полученное по указанному URI. Это содержимое выводится на экран
         отдельными порциями по 400 символов. После каждой такой порции
         следует нажать клавишу <ENTER>, чтобы вывести на экран следующую
         порцию, состоящую из 400 символов. */
      for(int i=1; ; i++) {
        ch = istrm.ReadByte();
        if(ch == -1) break;
          Console.Write((char) ch);
          if((i%400)==0) {
            Console.Write("\nНажмите клавишу <Enter>.");
            Console.ReadLine();
          }
        }

      // Закрыть ответный поток. При этом закрывается
      // также поток ввода istrm.
      resp.Close();
    } catch(WebException exc) {
      Console.WriteLine("Сетевая ошибка: " + exc.Message +
                        "\nКод состояния: " + exc.Status);
    } catch(ProtocolViolationException exc) {
      Console.WriteLine("Протокольная ошибка: " + exc.Message);
    } catch(UriFormatException exc) {
      Console.WriteLine("Ошибка формата URI: " + exc.Message);
    } catch(NotSupportedException exc) {
      Console.WriteLine("Неизвестный протокол: " + exc.Message);
    } catch(IOException exc) {
      Console.WriteLine("Ошибка ввода-вывода: " + exc.Message);
    } catch(System.Security.SecurityException exc) {
      Console.WriteLine("Исключение в связи с нарушением безопасности: " +
                        exc.Message);
```

```
    } catch(InvalidOperationException exc) {
      Console.WriteLine("Недопустимая операция: " + exc.Message);
    }
  }
}
```

Теперь перехватываются все исключения, которые могут быть сгенерированы сетевыми методами. Так, если изменить вызов метода `Create()` следующим образом:

```
WebRequest.Create("http://www.McGraw-Hill.com/moonrocket");
```

а затем перекомпилировать и еще раз выполнить программу, то в результате может быть выдано приведенное ниже сообщение об ошибке.

```
Сетевая ошибка: Удаленный сервер возвратил ошибку: (404) Не найден.
Код состояния: ProtocolError
```

На веб-сайте по адресу `www.McGraw-Hill.com` отсутствует раздел `moonrocket`, и поэтому он не найден по указанному URI, что и подтверждает приведенный выше результат.

Ради краткости и ясности в программах большинства примеров из этой главы отсутствует полноценная обработка исключений. Но в реальных приложениях она просто необходима.

Класс `Uri`

Как следует из табл. 26.1, метод `WebRequest.Create()` существует в двух вариантах. В одном варианте он принимает идентификатор URI в виде строки. Именно этот вариант и был использован в предыдущих примерах программ. А во втором варианте этот метод принимает идентификатор URI в виде экземпляра объекта класса `Uri`, определенного в пространстве имен `System`. Класс `Uri` инкапсулирует идентификатор URI. Используя класс `Uri`, можно сформировать URI, чтобы затем передать этот идентификатор методу `Create()`. Кроме того, идентификатор URI можно разделить на части. Для выполнения многих простых операций в Интернете класс `Uri` малопригоден. Тем не менее он может оказаться весьма полезным в более сложных ситуациях сетевого подключения к Интернету.

В классе `Uri` определяется несколько конструкторов. Ниже приведены наиболее часто используемые конструкторы этого класса.

```
public Uri(string uriString)
public Uri(Uri baseUri, string relativeUri)
```

В первой форме конструктора объект класса `Uri` создается по идентификатору URI, заданному в виде строки *`uriString`*. А во второй форме конструктора он создается по относительному URI, заданному в виде строки *`relativeUri`* относительно абсолютного URI, обозначаемого в виде объекта *`baseUri`* типа `Uri`. Абсолютный URI определяет полный адрес URI, а относительный URI — только путь к искомому ресурсу.

В классе `Uri` определяются многие поля, свойства и методы, оказывающие помощь в управлении идентификаторами URI или в получении доступа к различным частям URI. Особый интерес представляют приведенные ниже свойства.

Свойство	Описание
`public string Host { get; }`	Получает имя сервера
`public string LocalPath { get; }`	Получает локальный путь к файлу
`public string PathAndQuery { get; }`	Получает абсолютный путь и строку запроса
`public int Port { get; }`	Получает номер порта для указанного протокола. Так, для протокола HTTP номер порта равен 80
`public string Query { get; }`	Получает строку запроса
`public string Scheme { get; }`	Получает протокол

Перечисленные выше свойства полезны для разделения URI на составные части. Применение этих свойств демонстрируется в приведенном ниже примере программы.

```
// Пример применения свойств из класса Uri.

using System;
using System.Net;

class UriDemo {
  static void Main() {
    Uri sample = new
   Uri("http://HerbSchildt.com/somefile.txt?SomeQuery");
    Console.WriteLine("Хост: " + sample.Host);
    Console.WriteLine("Порт: " + sample.Port);
    Console.WriteLine("Протокол: " + sample.Scheme);
    Console.WriteLine("Локальный путь: " + sample.LocalPath);
    Console.WriteLine("Запрос: " + sample.Query);
    Console.WriteLine("Путь и запрос: " + sample.PathAndQuery);
  }
}
```

Эта программа дает следующий результат.

```
Хост: HerbSchildt.com
Порт: 80
Протокол: http
Локальный путь: /somefile.txt
Запрос: ?SomeQuery
Путь и запрос: /somefile.txt?SomeQuery
```

Доступ к дополнительной информации, получаемой в ответ по протоколу HTTP

С помощью сетевых средств, имеющихся в классе `HttpWebResponse`, можно получить доступ к другой информации, помимо содержимого указываемого ресурса. К этой информации, в частности, относится время последней модификации ресурса, а также имя сервера. Она оказывается доступной с помощью различных свойств, связанных с получаемым ответом. Все эти свойства, включая и те что, определены в классе `WebResponse`, сведены в табл. 26.5. В приведенных далее примерах программ демонстрируется применение этих свойств на практике.

Таблица 26.5. Свойства, определенные в классе `HttpWebResponse`

Свойство	Описание
`public string CharacterSet { get; }`	Получает название используемого набора символов
`public string ContentEncoding { get; }`	Получает название схемы кодирования
`public long ContentLength { get; }`	Получает длину принимаемого содержимого. Если она недоступна, свойство имеет значение -1
`public string ContentType { get; }`	Получает описание содержимого
`public CookieCollection Cookies { get; set; }`	Получает или устанавливает список cookie-наборов, присоединяемых к ответу
`public WebHeaderCollection Headers{ get; }`	Получает коллекцию заголовков, присоединяемых к ответу
`public bool IsFromCache { get; }`	Принимает логическое значение `true`, если запрос получен из кеша. А если запрос доставлен по сети, то принимает логическое значение `false`
`public bool IsMutuallyAuthenticated { get; }`	Принимает логическое значение `true`, если клиент и сервер опознают друг друга, а иначе — принимает логическое значение `false`
`public DateTime LastModified { get; }`	Получает время последней модификации ресурса
`public string Method { get; }`	Получает строку, которая задает способ ответа
`public Version ProtocolVersion { get; }`	Получает объект типа `Version`, описывающий версию протокола HTTP, используемую в транзакции
`public Uri ReponseUri { get; }`	Получает URI, по которому был сформирован ответ. Этот идентификатор может отличаться от запрашиваемого, если ответ был переадресован по другому URI
`public string Server { get; }`	Получает строку, обозначающую имя сервера
`public HttpStatusCode StatusCode { get; }`	Получает объект типа `HttpStatusCode`, описывающий состояние транзакции
`public string StatusDescription { get; }`	Получает строку, обозначающую состояние транзакции в удобочитаемой форме

Доступ к заголовку

Для доступа к заголовку с информацией, получаемой в ответ по протоколу HTTP, служит свойство `Headers`, определенное в классе `HttpWebResponse`.

```
public WebHeaderCollection Headers{ get; }
```

Заголовок протокола HTTP состоит из пар "имя–значение", представленных строками. Каждая пара "имя–значение" хранится в коллекции класса `WebHeaderCollection`. Эта коллекция специально предназначена для хранения пар "имя–значение" и приме-

няется аналогично любой другой коллекции (подробнее об этом см. в главе 25). Строковый массив имен может быть получен из свойства `AllKeys`, а отдельные значения — по соответствующему имени при вызове метода `GetValues()`. Этот метод возвращает массив строк, содержащий значения, связанные с заголовком, передаваемым в качестве аргумента. Метод `GetValues()` перегружается, чтобы принять числовой индекс или имя заголовка.

В приведенной ниже программе отображаются заголовки, связанные с сетевым ресурсом, доступным по адресу `www.McGraw-Hill.com`.

```
// Проверить заголовки.

using System;
using System.Net;

class HeaderDemo {
  static void Main() {

    // Создать объект запроса типа WebRequest по указанному URI.
    HttpWebRequest req = (HttpWebRequest)
    WebRequest.Create("http://www.McGraw-Hill.com");

    // Отправить сформированный запрос и получить на него ответ.
    HttpWebResponse resp = (HttpWebResponse)
    req.GetResponse();

    // Получить список имен.
    string[] names = resp.Headers.AllKeys;

    // Отобразить пары "имя-значение" из заголовка.
    Console.WriteLine("{0,-20}{1}\n", "Имя", "Значение");
    foreach(string n in names) {
      Console.Write("{0,-20}", n);
      foreach(string v in resp.Headers.GetValues(n))
        Console.WriteLine(v);
    }

    // Закрыть ответный поток.
    resp.Close();
  }
}
```

Ниже приведен полученный результат. Не следует забывать, что информация в заголовке периодически меняется, поэтому у вас результат может оказаться несколько иным.

```
Имя                  Значение
Transfer-encoding    chunked
Content-Type         text/html
Date                 Sun, 06 Dec 2009 20:32:06 GMT
Server               Sun-ONE-Web-Server/6.1
```

Доступ к cookie-наборам

Для доступа к cookie-наборам, получаемым в ответ по протоколу HTTP, служит свойство `Cookies`, определенное в классе `HttpWebResponse`. В cookie-наборах содержится информация, сохраняемая браузером. Они состоят из пар "имя–значение"

и упрощают некоторые виды доступа к веб-сайтам. Ниже показано, каким образом определяется свойство Cookies.

```
public CookieCollection Cookies { get; set; }
```

В классе CookieCollection реализуются интерфейсы ICollection и IEnumerable, и поэтому его можно использовать аналогично классу любой другой коллекции (подробнее об этом см. в главе 25). У этого класса имеется также индексатор, позволяющий получать cookie-набор по указанному индексу или имени.

В коллекции типа CookieCollection хранятся объекты класса Cookie. В классе Cookie определяется несколько свойств, предоставляющих доступ к различным фрагментам информации, связанной с cookie-набором. Ниже приведены два свойства, Name и Value, используемые в примерах программ из этой главы.

```
public string Name { get; set; }
public string Value { get; set; }
```

Имя cookie-набора содержится в свойстве Name, а его значение — в свойстве Value.

Для того чтобы получить список cookie-наборов из принятого ответа, необходимо предоставить cookie-контейнер с запросом. И для этой цели в классе HttpWebRequest определяется свойство CookieContainer, приведенное ниже.

```
public CookieContainer CookieContainer { get; set; }
```

В классе CookieContainer предоставляются различные поля, свойства и методы, позволяющие хранить cookie-наборы. По умолчанию свойство CookieContainer содержит пустое значение. Для того чтобы воспользоваться cookie-наборами, необходимо установить это свойство равным экземпляру класса CookieContainer. Во многих приложениях свойство CookieContainer не применяется непосредственно, а вместо него из принятого ответа составляется и затем используется коллекция типа CookieCollection. Свойство CookieContainer просто обеспечивает внутренний механизм сохранения cookie-наборов.

В приведенном ниже примере программы отображаются имена и значения cookie-наборов, получаемых из источника по URI, указываемому в командной строке. Следует, однако, иметь в виду, что cookie-наборы используются не на всех веб-сайтах, поэтому нужно еще найти такой веб-сайт, который поддерживает cookie-наборы.

```
/* Пример проверки cookie-наборов.
   Для того чтобы проверить, какие именно cookie-наборы
   используются на веб-сайте, укажите его имя в командной строке.
   Так, если назвать эту программу CookieDemo, то по команде

   CookieDemo http://msn.com

    отобразятся cookie-наборы с веб-сайта по адресу www.msn.com. */

using System;
using System.Net;

class CookieDemo {
  static void Main(string[] args) {

    if(args.Length != 1) {
```

```
      Console.WriteLine("Применение: CookieDemo <uri>");
      return;
    }

    // Создать объект запроса типа WebRequest по указанному URI.
    HttpWebRequest req = (HttpWebRequest)
    WebRequest.Create(args[0]);

    // Получить пустой контейнер.
    req.CookieContainer = new CookieContainer();

    // Отправить сформированный запрос и получить на него ответ.
    HttpWebResponse resp = (HttpWebResponse)
    req.GetResponse();

    // Отобразить cookie-наборы.
    Console.WriteLine("Количество cookie-наборов: " +
                      resp.Cookies.Count);
    Console.WriteLine("{0,-20}{1}", "Имя", "Значение");
    for(int i=0; i < resp.Cookies.Count; i++)
      Console.WriteLine("{0, -20}{1}",
                        resp.Cookies[i].Name,
                        resp.Cookies[i].Value);

    // Закрыть ответный поток.
    resp.Close();
  }
}
```

Применение свойства LastModified

Иногда требуется знать, когда именно сетевой ресурс был обновлен в последний раз. Это нетрудно сделать, пользуясь сетевыми средствами класса `HttpWebResponse`, среди которых определено свойство `LastModified`, приведенное ниже.

```
public DateTime LastModified { get; }
```

С помощью свойства `LastModified` получается время обновления содержимого сетевого ресурса в последний раз.

В приведенном ниже примере программы отображаются дата и время, когда был в последний раз обновлен ресурс, указываемый по URI в командной строке.

```
/* Использовать свойство LastModified.
   Для того чтобы проверить дату последнего обновления веб-сайта,
   введите его URI в командной строке. Так, если назвать эту программу
   LastModifiedDemo, то для проверки даты последней модификации веб-сайта
   по адресу www.HerbSchildt.com введите команду

   LastModifiedDemo http://HerbSchildt.com
*/

using System;
using System.Net;

class LastModifiedDemo {
```

```
  static void Main(string[] args) {

    if(args.Length != 1) {
      Console.WriteLine("Применение: LastModifiedDemo <uri>");
      return;
    }

    HttpWebRequest req = (HttpWebRequest)
        WebRequest.Create(args[0]);

    HttpWebResponse resp = (HttpWebResponse)
        req.GetResponse();

    Console.WriteLine("Последняя модификация: " + resp.LastModified);

    resp.Close();
  }
}
```

Практический пример создания программы MiniCrawler

Для того чтобы показать, насколько просто программировать для Интернета средствами классов `WebRequest` и `WebReponse`, обратимся к разработке скелетного варианта *поискового робота* под названием MiniCrawler. Поисковый робот представляет собой программу последовательного перехода от одной ссылки на сетевой ресурс к другой. Поисковые роботы применяются в поисковых механизмах для каталогизации содержимого. Разумеется, поисковый робот MiniCrawler не обладает такими развитыми возможностями, как те, что применяются в поисковых механизмах. Эта программа начинается с ввода пользователем конкретного адреса URI, по которому затем читается содержимое и осуществляется поиск в нем ссылки. Если ссылка найдена, то программа запрашивает пользователя, желает ли он перейти по этой ссылке к обнаруженному сетевому ресурсу, найти другую ссылку на имеющейся странице или выйти из программы. Несмотря на всю простоту такого алгоритма поиска сетевых ресурсов, он служит интересным и наглядным примером доступа к Интернету средствами C#.

Программе MiniCrawler присущ ряд ограничений. Во-первых, в ней обнаруживаются только абсолютные ссылки, указываемые по гипертекстовой команде `href="http`. Относительные ссылки при этом не обнаруживаются. Во-вторых, возврат к предыдущей ссылке в программе не предусматривается. И в-третьих, в ней отображаются только ссылки, но не окружающее их содержимое. Несмотря на все указанные ограничения данного скелетного варианта поискового робота, он вполне работоспособен и может быть без особых хлопот усовершенствован для решения других задач. На самом деле добавление новых возможностей в программу MiniCrawler — это удобный случай освоить на практике сетевые классы и узнать больше о сетевом подключении к Интернету.

Ниже приведен полностью исходный код программы MiniCrawler.

```
/* MiniCrawler: скелетный вариант поискового робота.
   Применение: для запуска поискового робота укажите URI
   в командной строке. Например, для того чтобы начать поиск
   с адреса www.McGraw-Hill.com, введите следующую команду:

      MiniCrawler http://McGraw-Hill.com
```

```
*/

using System;
using System.Net;
using System.IO;

class MiniCrawler {

  // Найти ссылку в строке содержимого.
  static string FindLink(string htmlstr,
                         ref int startloc) {
    int i;
    int start, end;
    string uri = null;

    i = htmlstr.IndexOf("href=\"http", startloc,
                        StringComparison.OrdinalIgnoreCase);
    if(i != -1) {
      start = htmlstr.IndexOf('"', i) + 1;
      end = htmlstr.IndexOf('"', start);
      uri = htmlstr.Substring(start, end-start);
      startloc = end;
    }

    return uri;
  }

  static void Main(string[] args) {
    string link = null;
    string str;
    string answer;

    int curloc; // содержит текущее положение в ответе
    if(args.Length != 1) {
      Console.WriteLine("Применение: MiniCrawler <uri>");
      return ;
    }

    string uristr = args[0]; // содержит текущий URI

    HttpWebResponse resp = null;

    try {

      do {
        Console.WriteLine("Переход по ссылке " + uristr);

        // Создать объект запроса типа WebRequest по указанному URI.
        HttpWebRequest req = (HttpWebRequest)
               WebRequest.Create(uristr);

        uristr = null; // запретить дальнейшее использование этого URI

        // Отправить сформированный запрос и получить на него ответ.
        resp = (HttpWebResponse) req.GetResponse();

        // Получить поток ввода из принятого ответа.
```

```
    Stream istrm = resp.GetResponseStream();

    // Заключить поток ввода в оболочку класса StreamReader.
    StreamReader rdr = new StreamReader(istrm);

    // Прочитать всю страницу.
    str = rdr.ReadToEnd();

    curloc = 0;

    do {
      // Найти следующий URI для перехода по ссылке.
      link = FindLink(str, ref curloc);

      if(link != null) {
        Console.WriteLine("Найдена ссылка: " + link);

        Console.Write("Перейти по ссылке, Искать дальше, Выйти?");
        answer = Console.ReadLine();

        if(string.Equals(answer, "П",
               StringComparison.OrdinalIgnoreCase)) {
          uristr = string.Copy(link);
          break;
        } else if(string.Equals(answer, "В",
               StringComparison.OrdinalIgnoreCase)) {
          break;
        } else if(string.Equals(answer, "И",
               StringComparison.OrdinalIgnoreCase)) {
          Console.WriteLine("Поиск следующей ссылки.");
        }
      } else {
        Console.WriteLine("Больше ссылок не найдено.");
        break;
      }

    } while(link.Length > 0);

    // Закрыть ответный поток.
    if(resp != null) resp.Close();
  } while(uristr != null);

} catch(WebException exc) {
  Console.WriteLine("Сетевая ошибка: " + exc.Message +
                    "\nКод состояния: " + exc.Status);
} catch(ProtocolViolationException exc) {
  Console.WriteLine("Протокольная ошибка: " + exc.Message);
} catch(UriFormatException exc) {
  Console.WriteLine("Ошибка формата URI: " + exc.Message);
} catch(NotSupportedException exc) {
  Console.WriteLine("Неизвестный протокол: " + exc.Message);
} catch(IOException exc) {
  Console.WriteLine("Ошибка ввода-вывода: " + exc.Message);
} finally {
  if(resp != null) resp.Close();
}
```

```
    Console.WriteLine("Завершение программы MiniCrawler.");
  }
}
```

Ниже приведен пример сеанса поиска, начиная с адреса `www.McGraw-Hill.com`. Следует иметь в виду, что конкретный результат поиска зависит от состояния содержимого на момент поиска.

```
Переход по ссылке http://mcgraw-hill.com
Найдена ссылка: http://sti.mcgraw-hill.com:9000/cgi-bin/query?mss=search&pg=aq
Перейти по ссылке, Искать дальше, Выйти? И
Поиск следующей ссылки.
Найдена ссылка: http://investor.mcgraw-hill.com/phoenix.zhtml?c=96562&p=irol-irhome
Перейти по ссылке, Искать дальше, Выйти? П
Переход по ссылке http://investor.mcgraw-hill.com/phoenix.zhtml?c=96562&p=irol-irhome
Найдена ссылка: http://www.mcgraw-hill.com/index.html
Перейти по ссылке, Искать дальше, Выйти? П
Переход по ссылке http://www.mcgraw-hill.com/index.html
Найдена ссылка: http://sti.mcgraw-hill.com:9000/cgi-bin/query?mss=search&pg=aq
Перейти по ссылке, Искать дальше, Выйти? В
Завершение программы MiniCrawler.
```

Рассмотрим подробнее работу программы MiniCrawler. Она начинается с ввода пользователем конкретного URI в командной строке. В методе `Main()` этот URI сохраняется в строковой переменной `uristr`. Затем по указанному URI формируется запрос, и переменной `uristr` присваивается пустое значение, указывающее на то, что данный URI уже использован. Далее отправляется запрос и получается ответ. После этого содержимое читается из потока ввода, возвращаемого методом `GetResponseStream()` и заключаемого в оболочку класса `StreamReader`. Для этой цели вызывается метод `ReadToEnd()`, возвращающий все содержимое в виде строки из потока ввода.

Далее программа осуществляет поиск ссылки в полученном содержимом. Для этого вызывается статический метод `FindLink()`, определяемый в программе MiniCrawler. Этот метод вызывается со строкой содержимого и исходным положением, с которого начинается поиск в полученном содержимом. Эти значения передаются методу `FindLink()` в виде параметров `htmlstr` и `startloc` соответственно. Обратите внимание на то, что параметр `startloc` относится к типу `ref`. Сначала в методе `FindLink()` создается копия строки содержимого в нижнем регистре, а затем осуществляется поиск подстроки `href="http`, обозначающей ссылку. Если эта подстрока найдена, то URI копируется в строковую переменную `uri`, а значение параметра `startloc` обновляется и становится равным концу ссылки. Но поскольку параметр `startloc` относится к типу `ref`, то это приводит к обновлению соответствующего аргумента метода `Main()`, активизируя поиск с того места, где он был прерван. В конечном итоге возвращается значение переменной `uri`. Эта переменная инициализирована пустым значением, и поэтому если ссылка не найдена, то возвращается пустая ссылка, обозначающая неудачный исход поиска.

Если ссылка, возвращаемая методом `FindLink()`, не является пустой, то она отображается в методе `Main()`, и далее программа запрашивает у пользователя очередные действия. Пользователю предоставляются одна из трех следующих возможностей: перейти по найденной ссылке, нажав клавишу <П>, искать следующую ссылку в имеющемся содержимом, нажав клавишу <И>, или же выйти из программы, нажав клавишу <В>. Если пользователь нажмет клавишу <П>, то программа осуществит переход по найденной ссылке и получит новое содержимое по этой ссылке. После этого поиск

очередной ссылки будет начат уже в новом содержимом. Этот процесс продолжается до тех пор, пока не будут исчерпаны все возможные ссылки.

В качестве упражнения вы сами можете усовершенствовать программу MiniCrawler, дополнив ее, например, возможностью перехода по относительным ссылкам. Сделать это не так уж и трудно. Кроме того, вы можете полностью автоматизировать поисковый робот, чтобы он сам переходил по найденной ссылке без вмешательства со стороны пользователя, начиная со ссылки, обнаруженной на самой первой странице полученного содержимого, и продолжая переход по ссылкам на новых страницах. Как только будет достигнут тупик, поисковый робот должен вернуться на один уровень назад, найти следующую ссылку и продолжить переход по ссылке. Для организации именно такого алгоритма работы программы вам потребуется стек, в котором должны храниться идентификаторы URI и текущее состояние поиска в строке URI. С этой целью можно, в частности, воспользоваться коллекцией класса `Stack`. В качестве более сложной, но интересной задачи попробуйте организовать вывод ссылок в виде дерева.

Применение класса `WebClient`

В заключение этой главы уместно рассмотреть класс `WebClient`. Как упоминалось в самом ее начале, класс `WebClient` рекомендуется использовать вместо классов `WebRequest` и `WebResponse` в том случае, если в приложении требуется лишь выгружать или загружать данные из Интернета. Преимущество класса `WebClient` заключается в том, что он автоматически выполняет многие операции, освобождая от их программирования вручную.

В классе `WebClient` определяется единственный конструктор.

```
public WebClient()
```

Кроме того, в классе `WebClient` определяются свойства, сведенные в табл. 26.6, а также целый ряд методов, поддерживающих как синхронную, так и асинхронную передачу данных. Но поскольку рассмотрение асинхронной передачи данных выходит за рамки этой главы, то в табл. 26.7 приведены только те методы, которые поддерживают синхронную передачу данных. Все методы класса `WebClient` генерируют исключение `WebException`, если во время передачи данных возникает ошибка.

Таблица 26.6. Свойства, определенные в классе `WebClient`

Свойство	Описание
`public string BaseAddress { get; set; }`	Получает или устанавливает базовый адрес требуемого URI. Если это свойство установлено, то адреса, задаваемые в методах класса `WebClient`, должны определяться относительно этого базового адреса
`public RequestCachePolicy CachePolicy { get; set; }`	Получает или устанавливает правила, определяющие, когда именно используется кэш
`public ICredentials Credentials { get; set; }`	Получает или устанавливает мандат, т.е. учетные данные пользователя. По умолчанию это свойство имеет пустое значение
`public Encoding Encoding { get; set; }`	Получает или устанавливает схему кодирования символов при передаче строк

Окончание табл. 26.6

Свойство	Описание
`public WebHeaderCollection Headers{ get; set; }`	Получает или устанавливает коллекцию заголовков запроса
`public bool IsBusy( get; }`	Принимает логическое значение `true`, если данные по-прежнему передаются по запросу, а иначе — логическое значение `false`
`public IWebProxy Proxy { get; set; }`	Получает или устанавливает прокси-сервер
`public NameValueCollection QueryString { get; set; }`	Получает или устанавливает строку запроса, состоящую из пар "имя-значение", которые могут быть присоединены к запросу. Строка запроса отделяется от URI символом ?. Если же таких пар несколько, то каждая из них отделяется символом @
`public WebHeaderCollection ResponseHeaders{ get; }`	Получает коллекцию заголовков ответа
`public bool UseDefaultCredentials { get; set; }`	Получает или устанавливает значение, которое определяет, используется ли для аутентификации устанавливаемый по умолчанию мандат. Если принимает логическое значение `true`, то используется мандат, устанавливаемый по умолчанию, т.е. учетные данные пользователя, в противном случае этот мандат не используется

Таблица 26.7. Методы синхронной передачи, определенные в классе `WebClient`

Метод	Определение
`public byte[] DownloadData(string address)`	Загружает информацию по адресу URI, обозначаемому параметром *address*. Возвращает результат в виде массива байтов
`public byte[] DownloadData(Uri address)`	Загружает информацию по адресу URI, обозначаемому параметром *address*. Возвращает результат в виде массива байтов
`public void DownloadFile(string uri, string fileName)`	Загружает информацию по адресу URI, обозначаемому параметром *fileName*. Сохраняет результат в файле *fileName*
`public void DownloadFile(Uri address, string fileName)`	Загружает информацию по адресу URI, обозначаемому параметром *address*. Сохраняет результат в файле *fileName*
`public string DownloadString(string address)`	Загружает информацию по адресу URI, обозначаемому параметром *address*. Возвращает результат в виде символьной строки типа `string`
`public string DownloadString(Uri address)`	Загружает информацию по адресу URI, обозначаемому параметром *address*. Возвращает результат в виде символьной строки типа `string`
`public Stream OpenRead(string address)`	Возвращает поток ввода для чтения информации по адресу URI, обозначаемому параметром *address*. По окончании чтения информации этот поток необходимо закрыть

Продолжение табл. 26.7

Метод	Определение
`public Stream OpenRead(Uri address)`	Возвращает поток ввода для чтения информации по адресу URI, обозначаемому параметром *address*. По окончании чтения информации этот поток необходимо закрыть
`public Stream OpenWrite(string address)`	Возвращает поток вывода для записи информации по адресу URI, обозначаемому параметром *address*. По окончании записи информации этот поток необходимо закрыть
`public Stream OpenWrite(Uri address)`	Возвращает поток вывода для записи информации по адресу URI, обозначаемому параметром *address*. По окончании записи информации этот поток необходимо закрыть
`public Stream OpenWrite(string address, string method)`	Возвращает поток вывода для записи информации по адресу URI, обозначаемому параметром *address*. По окончании записи информации этот поток необходимо закрыть. В строке, передаваемой в качестве параметра *method*, указывается, как именно следует записывать информацию
`public Stream OpenWrite(Uri address, string method)`	Возвращает поток вывода для записи информации по адресу URI, обозначаемому параметром *address*. По окончании записи информации этот поток необходимо закрыть. В строке, передаваемой в качестве параметра *method*, указывается, как именно следует записывать информацию
`public byte[] UploadData(string address, byte[] data)`	Записывает информацию из массива *data* по адресу URI, обозначаемому параметром *address*. В итоге возвращается ответ
`public byte[] UploadData(Uri address, byte[] data)`	Записывает информацию из массива *data* по адресу URI, обозначаемому параметром *address*. В итоге возвращается ответ
`public byte[] UploadData(string address, string method, byte[] data)`	Записывает информацию из массива *data* по адресу URI, обозначаемому параметром *address*. В итоге возвращается ответ. В строке, передаваемой в качестве параметра *method*, указывается, как именно следует записывать информацию
`public byte[] UploadData(Uri address, string method, byte[] data)`	Записывает информацию из массива *data* по адресу URI, обозначаемому параметром *address*. В итоге возвращается ответ. В строке, передаваемой в качестве параметра *method*, указывается, как именно следует записывать информацию
`public byte[] UploadFile(string address, string fileName)`	Записывает информацию в файл *fileName* по адресу URI, обозначаемому параметром *address*. В итоге возвращается ответ
`public byte[] UploadFile(Uri address, string fileName)`	Записывает информацию в файл *fileName* по адресу URI, обозначаемому параметром *address*. В итоге возвращается ответ

Окончание табл. 26.7

Метод	Определение
`public byte[] UploadFile(string address, string method, string fileName)`	Записывает информацию в файл `fileName` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ. В строке, передаваемой в качестве параметра `method`, указывается, как именно следует записывать информацию
`public byte[] UploadFile(Uri address, string method, string fileName)`	Записывает информацию в файл `fileName` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ. В строке, передаваемой в качестве параметра `method`, указывается, как именно следует записывать информацию
`public string UploadString(string address, string data)`	Записывает строку `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ
`public string UploadString(Uri address, string data)`	Записывает строку `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ
`public string UploadString(string address, string method, string data)`	Записывает строку `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ. В строке, передаваемой в качестве параметра `method`, указывается, как именно следует записывать информацию
`public string UploadString(Uri address, string method, string data)`	Записывает строку `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ. В строке, передаваемой в качестве параметра `method`, указывается, как именно следует записывать информацию
`public byte[] UploadValues(string address, NameValueCollection data)`	Записывает значения из коллекции `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ
`public byte[] UploadValues(Uri address, NameValueCollection data)`	Записывает значения из коллекции `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ
`public byte[] UploadValues(string address, string method, NameValueCollection data)`	Записывает значения из коллекции `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ. В строке, передаваемой в качестве параметра `method`, указывается, как именно следует записывать информацию
`public byte[] UploadValues(Uri address, string method, NameValueCollection data)`	Записывает значения из коллекции `data` по адресу URI, обозначаемому параметром `address`. В итоге возвращается ответ. В строке, передаваемой в качестве параметра `method`, указывается, как именно следует записывать информацию

В приведенном ниже примере программы демонстрируется применение класса `WebClient` для загрузки данных в файл по указанному сетевому адресу.

```
// Использовать класс WebClient для загрузки данных
// в файл по указанному сетевому адресу.

using System;
using System.Net;
using System.IO;

class WebClientDemo {
  static void Main() {
    WebClient user = new WebClient();
    string uri = "http://www.McGraw-Hill.com";
    string fname = "data.txt";

    try {
      Console.WriteLine("Загрузка данных по адресу " +
                        uri + " в файл " + fname);
      user.DownloadFile(uri, fname);
    } catch (WebException exc) {
      Console.WriteLine(exc);
    }

    Console.WriteLine("Загрузка завершена.");
  }
}
```

Эта программа загружает информацию по адресу `www.McGrawHill.com` и помещает ее в файл `data.txt`. Обратите внимание на строки кода этой программы, в которых осуществляется загрузка информации. Изменив символьную строку `uri`, можно загрузить информацию по любому адресу URI, включая и конкретные файлы, доступные по указываемому URI.

Несмотря на то что классы `WebRequest` и `WebResponse` предоставляют больше возможностей для управления и доступа к более обширной информации, для многих приложений оказывается достаточно и средств класса `WebClient`. Этим классом особенно удобно пользоваться в тех случаях, когда требуется только загрузка информации из веб-ресурса. Так, с помощью средств класса `WebClient` можно получить из Интернета обновленную документацию на приложение.

ПРИЛОЖЕНИЕ

Краткий справочник по составлению документирующих комментариев

В языке C# предусмотрено три вида комментариев. К двум первым относятся комментарии // и /* */, а третий основан на дескрипторах языка XML и называется *документирующим комментарием*. (Иногда его еще называют XML-комментарием.) Однострочный документирующий комментарий начинается с символов ///, а многострочный начинается с символов /** и оканчивается символами */. Строки после символов /** могут начинаться с одного символа *, хотя это и не обязательно. Если все последующие строки многострочного комментария начинаются с символа *, то этот символ игнорируется.

Документирующие комментарии вводятся перед объявлением таких элементов языка C#, как классы, пространства имен, методы, свойства и события. С помощью документирующих комментариев можно вводить в исходный текст программы сведения о самой программе. При компиляции программы документирующие комментарии к ней могут быть помещены в отдельный XML-файл. Кроме того, документирующие комментарии можно использовать в средстве IntelliSense интегрированной среды разработки Visual Studio.

Дескрипторы XML-комментариев

В C# поддерживаются дескрипторы документации в формате XML, сведенные в табл. 1. Большинство дескрипторов XML-комментариев не требует особых пояснений

и действуют подобно всем остальным дескрипторам XML, знакомым многим программистам. Тем не менее дескриптор <list> — сложнее других. Он состоит из двух частей: заголовка и элементов списка. Ниже приведена общая форма дескриптора <list>:

```
<listheader>
  <term> имя </term>
 .<description> текст </description>
</listheader>
```

где текст описывает имя. Для описания таблиц текст не используется. Ниже приведена общая форма элемента списка:

```
<item>
  <term> имя_элемента </term>
  <description> текст </description>
</item>
```

где текст описывает имя_элемента. Для описания маркированных и нумерованных списков, а также таблиц имя_элемента не используется. Допускается применение нескольких элементов списка <item>.

Таблица 1. Дескрипторы XML-комментариев

Дескриптор	Описание
<c> код </c>	Определяет текст, на который указывает код, как программный код
<code> код </code>	Определяет несколько строк текста, на который указывает код, как программный код
<example> пояснение </example>	Определяет текст, на который указывает пояснение, как описание примера кода
<exception cref = "имя"> пояснение </exception>	Описывает исключительную ситуацию, на которую указывает имя
<include file = 'fname' path = 'path[@tagName = "tagID "]' />	Определяет файл, содержащий XML-комментарии для текущего исходного файла. При этом fname обозначает имя файла; path — путь к файлу; tagName — имя дескриптора; tagID — идентификатор дескриптора
<list type = "тип""> заголовок_списка элементы_списка </list>	Определяет список. При этом тип обозначает тип списка, который может быть маркированным, нумерованным или таблицей
<para> текст </para>	Определяет абзац текста в другом дескрипторе
<param name = 'имя_параметра'> пояснение </param>	Документирует параметр, на который указывает имя_параметра. Текст, обозначаемый как пояснение, описывает параметр
<paramref name = "имя_параметра" />	Обозначает имя_параметра как имя конкретного параметра
<permission cref = "идентификатор"> пояснение </permission>	Описывает параметр разрешения, связанный с членами класса, на которые указывает идентификатор. Текст, обозначаемый как пояснение, описывает параметры разрешения

Окончание табл. 1

Дескриптор	Описание
`<remarks> пояснение </remarks>`	Текст, обозначаемый как `пояснение`, представляет собой общие комментарии, которые часто используются для описания класса или структуры
`<returns> пояснение </returns>`	Текст, обозначаемый как `пояснение`, описывает значение, возвращаемое методом
`<see cref = "идентификатор" />`	Объявляет ссылку на другой элемент, обозначаемый как `идентификатор`
`<seealso cref = "идентификатор" />`	Объявляет ссылку типа "см. также" на `идентификатор`
`<summary> пояснение </summary>`	Текст, обозначаемый как `пояснение`, представляет собой общие комментарии, которые часто используются для описания метода или другого члена класса
`<typeparam name = "имя_параметра"> пояснение </typeparam>`	Документирует параметр типа, на который указывает `имя_параметра`. Текст, обозначаемый как `пояснение`, описывает параметр типа
`<typeparamref name = "имя_параметра"/>`	Обозначает `имя_параметра` как имя параметра типа

Компилирование документирующих комментариев

Для получения XML-файла, содержащего документирующие комментарии, достаточно указать параметр `/doc` в командной строке компилятора. Например, для компилирования файла `DocTest.cs`, содержащего XML-комментарии, в командной строке необходимо ввести следующее.

```
csc DocTest.cs /doc:DocTest.xml
```

Для вывода результата в XML-файл из интегрированной среды разработки Visual Studio необходимо активизировать окно **Свойства** (Properties) для текущего проекта. Затем следует выбрать свойство **Построение** (Build), установить флажок **XML-файл документации** (XML Documentation File) и указать имя выходного XML-файла.

Пример составления документации в формате XML

В приведенном ниже примере демонстрируется применение нескольких документирующих комментариев: как однострочных, так и многострочных. Любопытно, что многие программисты пользуются последовательным рядом однострочных документирующих комментариев вместо многострочных, даже если комментарий занимает насколько строк. Такой подход применяется и в ряде комментариев из данного примера. Его преимущество заключается в том, что он позволяет ясно обозначить каждую строку как часть длинного документирующего комментария. Но это все же, скорее, дело стиля, чем общепринятая практика составления документирующих комментариев.

```
// Пример составления документирующих комментариев.

using System;

/** <remark>
  Это пример многострочного документирования в формате XML.
  В классе Test демонстрируется ряд дескрипторов.
</remark>
*/

class Test {
  /// <summary>
  /// Выполнение программы начинается с метода Main().
  /// </summary>
static void Main() {
  int sum;

  sum = Summation(5);
  Console.WriteLine("Сумма последовательных чисел " +
                    5 + " равна " + sum);
  }

  /// <summary>
  /// Метод Summation() возвращает сумму его аргументов.
  /// <param name = "val">
  /// Суммируемое значение передается в качестве параметра val.
  /// </param>
  /// <see cref="int"> </see>
  /// <returns>
  /// Сумма возвращается в виде значения типа int.
  /// </returns>
  /// </summary>
  static int Summation(int val) {
    int result = 0;

    for(int i=1; i <= val; i++)
      result += i;

    return result;
  }
}
```

Если текст приведенной выше программы содержится в файле `XmlTest.cs`, то по следующей команде будет скомпилирована программа и получен файл `XmlTest.xml`, содержащий комментарии к ней.

```
csc XmlTest.cs /doc:XmlTest.xml
```

После компилирования получается XML-файл, содержимое которого приведено ниже.

```
<?xml version="1.0"?>
<doc>
    <assembly>
       <name>DocTest</name>
```

```
    </assembly>
    <members>
        <member name="T:Test">
            <remark>
            Это пример многострочного документирования в формате XML.
            В классе Test демонстрируется ряд дескрипторов.
            </remark>
        </member>
        <member name="M:Test.Main">
            <summary>
            Выполнение программы начинается с метода Main().
            </summary>
        </member>
        <member name="M:Test.Summation(System.Int32)">
            <summary>
            Метод Summation() возвращает сумму его аргументов.
            <param name="val">
            Суммируемое значение передается в качестве параметра val.
            </param>
            <see cref="T:System.Int32"> </see>
            <returns>
            Сумма возвращается в виде значения типа int.
            </returns>
            </summary>
        </member>
    </members>
</doc>
```

Следует заметить, что каждому документируемому элементу присваивается уникальный идентификатор. Такие идентификаторы применяются в других программах, которые документируются в формате XML.

Предметный указатель

А

Аксессоры
  вызов *304*
  модификаторы доступа
    ограничения *323*
    применение *320*
  назначение *304*
  разновидности *304*
  событий *500*
Анонимные функции
  назначение *483*
  преимущество *483*
  разновидности *483*
Аргументы
  именованные
    назначение *252*
    применение *252*
  командной строки *255*
  метода *162*
  назначение *52*
  необязательные
    назначение *247*
    и неоднозначность *250*
    и перегрузка методов *249*
    порядок объявления *249*
  способы передачи методу *220*
  типа *579*
Атрибуты
  AttributeUsage *570*
  Conditional *571*
  MethodImplAttribute, применение *860*
  Obsolete *572*
  встроенные *570*
  извлечение *564*
  именованные параметры *566*
  назначение *562*
  позиционные параметры *566*
  присоединение *564*
  создание *563*
  указание *563*

Б

Байт-код *34*
Библиотека TPL
  возврат значения из задачи *899*
  задачи, создание и исполнение *887*
  идентификаторы задач, назначение и применение *890*
  классы
    Parallel, назначение и применение *906*
    TaskFactory, назначение и применение *895*
    Task, назначение и применение *887*
  лямбда-выражения, в качестве задачи, применение *896*
  методы
    Dispose(), назначение и применение *895*
    ForEach(), назначение и применение *915*
    For(), назначение и применение *909*
    Invoke(), назначение и применение *906*
    ожидания, назначение и применение *892*
  назначение *886*
  особенности *885*
  отмена задачи *901*
  признак отмены *901*
  продолжение задачи, создание *897*
Библиотеки классов
  C#. *717*
  для среды .NET Framework *66*
  организация *717*
  пространство имен System
    члены *719*
    структуры встроенных типов данных *727*
Буферы фиксированного размера
  назначение *693*
  создание *694*

В

Ввод-вывод
  в файл
    байтовый *441*
    символьный *449*
    последовательный *461*
    с произвольным доступом *461*
  данных в массив *463*
  двоичных данных *436, 454*
  консольный *436*
  основанный на потоках *431*
  отдельными байтами *432*
  отдельными символами *432*
  переадресация *453*
  с запоминанием *465*
Виртуальная машина Java *34*
Возможность взаимодействия, межъязыковая *35*
Выводимость типов *609*
Вызов
  перегружаемого конструктора *245*

по значению *220*
по ссылке *220*

Г

Групповая адресация, определение *478*

Д

Делегаты
Action
формы *769*
применение *769*
вызов
методов экземпляра *477*
любых методов *474*
главное преимущество *474*
групповая адресация *478*
ковариантность *481*
контравариантность *481*
назначение *483*
обобщенные
EventHandler<TEventArgs>, применение *508*
вариантные *633*
объявление *610*
общая форма объявления *474*
определение *473*
применение *474*
типа EventHandler, применение *508*
Деструкторы, назначение и применение *172*
Десятичная система счисления *80*
Динамическая диспетчеризация методов, принцип *356*
идентификация типов
назначение *537*
причины полезности *537*
Директивы
#define *529*
#else и #elif *531*
#error *533*
#if и #endif *529*
#line *534*
#pragma *534*
#region и #endregion *534*
#undef *533*
#warning *534*
using *518*
препроцессора *528*
Доступ к Интернету
cookie-наборы *1027*
заголовки протокола HTTP *1026*
обработка
исключений *1022*
сетевых ошибок *1021*
организация *1018*
передача данных
асинхронная *1015*
синхронная *1015*
получение дополнительной информации *1025*
по принципу запроса и ответа *1014*
пространство имен System.Net, члены *1012*
протоколы
определение *1013*
подключаемые *1014*
разработка поискового робота *1030*
сетевой ресурс, последнее обновление *1029*
универсальный идентификатор ресурса, определение *1013*

И

Идентификаторы
URI *1013*
директив препроцессора *529*
назначение *65*
применение *65*
Иерархии классов
многоуровневые *347*
обобщенных *620*
порядок вызова конструкторов *350*
простые *346*
ссылки на объекты разных классов *351*
Импликация *103*
Индексаторы
аксессоры get и set *304*
без базового массива *310*
интерфейсные *385*
многомерные *311*
назначение *303*
ограничения на применение *311*
одномерные *304*
перегружаемые *307*
преимущество *304*
Индекс массива, назначение *178*
Инициализаторы
коллекций *1009*
массивов *180*
объектов *246, 319*
проекции *666*
Инкапсуляция
как механизм программирования *42*
классы и объекты *43*
открытые и закрытые данные и код *42*
Интегрированная среда разработки Visual Studio *44, 46*
Интернет, определение *34*

Интерфейсы
ICloneable, реализация *779*
IComparable и IComparable<T>, реализация *617, 778, 990–993*
Icomparer и IComparer<T>, реализация *994–996*
Iconvertible, реализация *779*
IEnumerable, реализация *1001*
IEnumerator, реализация *1001*
IEquatable<T>, реализация *616, 778*
IFormatProvider, реализация *781*
IFormattable, реализация *781*
IObservable<T> и IObserver<T>, реализация и применение *781*
индексаторы
реализация *385*
общая форма объявления *385*
коллекций *924*
наследование *387*
обобщенные
контравариантность, применение *630*
объявление *612*
ковариантность, применение *626*
применение *612*
определение и реализация *375*
порядок и форма реализации *377*
правило выбора *391*
свойства
реализация *383*
общая форма объявления *383*
стандартные для среды .NET Framework *391*
форма объявления *376*
явная реализация *388*
Исключения
базового класса, перехват *426*
блоки try/catch, применение *404*
блок finally, применение *416*
вложение блоков try *413*
внутренние *420*
генерирование
вручную *414*
и перехват *405*
повторное *415*
классы *404*
обработчики *403*
оператор throw, применение *414*
последствия неперехвата *408*
при вводе-выводе *433, 442*
производных классов, перехват *426*
разнотипные, обработка *411*
сетевые, при доступе к Интернету *1021*
специальные, создание и применение *422*
стандартные *403, 420*
удаление после обработки *411*
универсальный перехват и обработка *412*
Исключительные ситуации
обработка
назначение *403*
главное преимущество *403*
для устранения программных ошибок *410*
ключевые слова try и catch *404*
организация обработки *404*
подсистема обработки в C# *404*
появление *403*
Итераторы
именованные
применение *1007*
создание *1006*
назначение *925*
несколько операторов yield, применение *1006*
обобщенные, создание *1008*
определение *1003*
прерывание *1005*
применение *1003*

K

Классы
Array
назначение *750*
методы *750*
свойства *750*
Assembly, члены *555*
Attribute, назначение *563*
BinaryReader
методы *456*
конструктор *455*
BinaryWriter
методы *455*
конструктор *454*
BitConverter
назначение *771*
методы *771*
Console
методы *437*
переадресация потоков, методы *453*
ConstructorInfo, члены *551*
Cookie, свойства *1028*
CookieCollection, члены *1028*
CookieContainer, члены *1028*
Exception
методы *418*
конструкторы *420*
свойства *418*
File
назначение *467*
методы *467*

FileStream
  методы *444, 446*
  конструкторы *441*
  средства копирования файлов *448*
GC
  назначение *774*
  методы *774*
  свойство *776*
HttpWebRequest, назначение *1018*
HttpWebResponse
  назначение *1018*
  свойства *1025*
Interlocked
  назначение *873*
  методы *873*
Math
  назначение *721*
  методы *721*
  поля *721*
MemberInfo
  методы *542*
  свойства *542*
MemoryStream
  конструктор *463*
  применение *463*
MethodInfo, члены *544*
Monitor
  назначение *855*
  методы управления синхронизацией *855*
Mutex
  методы *863*
  конструкторы *863*
object
  назначение *368*
  как универсальный тип данных *372*
  методы *368, 776*
  конструктор *777*
ParameterInfo, члены *544*
Process
  назначение *883*
  методы *883*
Random
  назначение *773*
  методы *773*
  конструкторы *773*
Semaphore
  методы *868*
  конструкторы *867*
Stream
  назначение *433*
  методы *433*
  свойства *433*
StreamReader
  конструкторы *451, 452*
  свойство EndOfStream *452*
  применение *451*
StreamWriter
  конструкторы *449, 450*
  применение *449*
String
  назначение *784*
  реализация интерфейсов *784*
  методы
    расширения *812*
    форматирования строк *816*
  конструкторы *784*
  перегружаемые операторы *786*
  поле, индексатор и свойство *785*
StringComparer
  свойства *766*
  применение *997*
StringReader
  конструктор *465*
  применение *465*
StringWriter
  конструктор *465*
  применение *465*
System.Delegate, члены *483*
Thread
  назначение *835*
  методы управления потоками *835*
  конструкторы *836*
  свойства
    IsBackground *846*
    Priority *847*
Tuple, назначение *777*
Type
  назначение *542*
  методы *542*
  свойства *543*
Uri
  назначение *1024*
  конструкторы *1024*
  свойства *1024*
WebClient
  назначение *1034*
  методы *1034*
  конструктор *1034*
  свойства *1034*
WebRequest
  назначение *1015*
  методы *1015*
  свойства *1015*
WebResponse
  назначение *1017*
  методы *1017*
  свойства *1017*

абстрактные
  реализация *364*
  объявление *364*
  правило выбора *391*
атрибутов
  члены *563*
  объявление *563*
базовые *329*
инициализация объектов *246*
исключений *404*
как ссылочные типы *154*
коллекций
  назначение *925*
  необобщенных *931*
  обобщенных *960*
  параллельных *983*
конструкторы *167*
назначение *147*
наследование *331*
обобщенные
  частичные *701*
  базовые *620*
  иерархии *620*
  общая форма объявления *585*
  определение *578*
  переопределение виртуальных методов *623*
  с несколькими параметрами типа *583*
  применение *579*
  получение экземпляров объектов *636*
  производные *622*
оболочки символьных потоков
  TextReader, методы ввода *434*
  TextWriter, методы вывода *435*
общая форма определения *148*
объекты как экземпляры класса *147*
оператор-точка *150*
определение *43*
порядок определения *149*
потоков
  назначение *432*
  байтовых *434*
  двоичных *436*
  символьных *436*
  специальные *434*
производные *329*
синхронизации, старые и новые *874*
статические *266*
суперклассы и подклассы *335*
функции-члены *148*
члены
  данных *148*
  доступ при наследовании *333*
  методы и другие функции-члены *43*
  защищенные *336*
  закрытый и открытый доступ *212*
  открытые и закрытые *209*
  управление доступом *209*
  статические *264*
  поля и переменные экземпляра *43*
Ключевые слова
  base *339, 343, 344*
  checked и unchecked
    общие формы *428*
    применение *428*
  const и volatile *710*
  delegate *474, 484*
  enum *397*
  event *494*
  extern *712*
  fixed *693*
  interface *376*
  lock *708, 850*
  new *343, 415*
  partial *700*
  readonly *709*
  sealed *367*
  static *260*
  this *174*
  unsafe *684*
  using *711*
  virtual *356*
  для обработки исключений *404*
  зарезервированные *64*
  контекстные *64, 1004*
Ключи, назначение *929*
Ковариантность *481, 626*
Кодовые блоки
  назначение *62*
  применение *62*
  создание *61*
Коллекции
  главное преимущество *924*
  назначение *923*
  необобщенные
    назначение *925*
    классы *931–949*
    интерфейсы *926–930*
    структура DictionaryEntry *931*
  обобщенные
    классы *960–982*
    интерфейсы *954–959*
    объявление *954*
    структура KeyValuePair<TKey, TValue> *960*
    принцип действия *954*
  параллельные
    назначение *983*

классы *983*
методы *984*
применение *984*
специальные
назначение *953*
классы *953*
с поразрядной организацией
хранение отдельных битов *950*
класс BitArray *950*
сравнение строк, порядок *997*
типы *924*
хранение объектов
встроенных типов *988*
определяемых пользователем классов *988*
Комментарии
документирующие
дескрипторы XML *1039*
многострочные *1039*
однострочные *1039*
определение *1039*
компилирование *1041*
пример составления *1041*
составление *1039*
многострочные *51*
назначение *51*
однострочные *52*
Компилирование и выполнение программ
в среде Visual Studio *46*
из командной строки *45*
Компиляция
многовариантная *532*
условная *529*
Конструкторы
базового класса, вызов *339*
вызываемые по умолчанию *167*
и наследование *337*
назначение *167*
общая форма определения *167*
параметризированные *168*
перегружаемые *241*
статические *265*
Контравариантность *481, 626*
Копии объектов, разновидности *779*
Критический раздел кода *709*

Л

Литералы
буквальные, строковые *82*
десятичные *80*
определение *79*
символьные *79*
с плавающей точкой *79*
строковые *81*
типы, указание *80*
целочисленные *79*
шестнадцатеричные *80*
Лямбда-выражения
блочные *492*
как обработчики событий *505*
лямбда-оператор => *488*
назначение *488*
одиночные *489*
разновидности *489*
этапы применения *489*
явное указание параметров *491*

М

Массивы
главное преимущество *177*
границы, соблюдение *181*
двумерные *182*
динамические
назначение *932*
в качестве коллекции *932*
обобщенные *961*
получение обычного массива *938*
сортировка и поиск *937*
доступ по индексу *178*
инициализация *180*
копирование *767*
массивов *185*
многомерные
инициализация *184*
объявление *183*
определение *182*
неявно типизированные *192*
обращение содержимого *766*
одномерные *178*
определение *177*
порядок применения *178*
присваивание ссылок *187*
прямоугольные *185*
реализация в виде объектов *177*
свойство Length, применение *189*
сортировка *763*
строк *203*
ступенчатые *185*
указателей *692*
Методы
Main()
возврат значений *254*
вызов *52*
передача аргументов *255*
абстрактные
назначение *364*

реализация *364*
общая форма *364*
анонимные
назначение *484*
как обработчики событий *505*
внешние переменные, применение *487*
возврат значения *485*
передача аргументов *484*
виртуальные
объявление *356*
предотвращение переопределения *368*
переопределение *355*
применение *360*
внешние, применение *712*
возврат
массивов *234*
значений *159*
объектов *231*
условия *158*
групповое преобразование *476*
запроса
назначение *669*
реализация *669*
назначение *43*
необязательные параметры и аргументы *248*
обобщенные
наложение ограничений *610*
объявление *609*
порядок вызова *609*
создание *607*
обращения со строками *199*
общая форма определения *155*
операторные
назначение *270*
формы *270*
определение *155*
параметризированные *164*
параметры и аргументы *155, 162*
перегружаемые *235*
передача
аргументов, способы *220*
значений по ссылке *222*
объектов по ссылке *218*
переопределение *356, 359*
расширения
назначение *678*
объявление *678*
рекурсивные *257*
синтаксического анализа *471*
сокрытие *345*
с переменным числом аргументов *229*
статические
ограничения *261*
применение *261*
условные *571*
частичные
реализация *701*
объявление *701*
ограничения *703*
Многозадачность
запуск отдельной задачи *882*
разновидности *834*
управление отдельным процессом *883*
Многопоточная обработка
блокировка *850*
взаимоблокировка *860*
главное преимущество *834*
момент окончания потока, определение *841*
новые средства .NET *882*
определение состояния потока *880*
основной поток
назначение *835*
применение *880*
отмена прерывания потока *878*
передача аргумента потоку *844*
потоки
определение *834*
приоритеты *847*
приоритетные и фоновые *835*
состояния *835*
прерывание потока *875*
приостановка и возобновление потока *880*
процессы, определение *834*
рекомендации *882*
синхронизация *835, 849*
создание нескольких потоков *839*
сообщение между потоками *856*
состояние гонки *860*
способы усовершенствования *838*
Многоязыковое программирование *34*
Множество
в качестве коллекции *980*
объектов *980*
операции *980, 982*
отсортированное *982*
Модификаторы
abstract *364*
const *710*
fixed *685*
override *356*
partial *700*
volatile *710*
доступа *155, 210*
internal *536*
private *155, 210*

protected *336*
protected internal *536*
public *149, 167, 210*
параметров
out *225, 227*
params *229*
ref *223, 227*
Мьютексы
именованные *867*
назначение *863*
получение и освобождение *863*
применение *863*

Н

Наследование
главное преимущество *332*
интерфейсов *387*
как один из основных принципов ООП *329*
классов *329*
повторное использование кода *349*
поддержка в C# *329*
предотвращение *367*
принцип иерархической классификации *44*
сокрытие
методов *345*
имен *344*
Небезопасный код
выполнение *681*
определение *681*
Недоступный код, исключение *166*
Непрямая адресация
многоуровневая *691*
одноуровневая *682*
Неуправляемый код *39, 681*

О

Области действия
вложенные *87*
определяемые
классом *86*
методом *86*
соблюдение правил *88*
Обнуляемые объекты
в выражениях отношения *699*
объявление *696*
применение в выражениях *697*
проверка на пустое значение *696*
Обобщения
аргументы типа *579*
главное преимущество *583*
контроль типов *579*
обеспечение типовой безопасности *580*
определение *576*
основная польза *583*
особая роль *575*
параметры типа
назначение и указание *578*
сравнение экземпляров *615*
присущие ограничения *636*
Общая система типов CTS *39*
Общеязыковая спецификация CLS *39*
Объектно-ориентированное программирование
инкапсуляция *42*
метод *33*
наследование *44*
основные принципы *41*
особенности *42*
полиморфизм *43*
Объекты, определение *42*
Ограничения
на базовый класс
назначение *585*
наложение, общая форма *586*
применение *586*
последствия *588*
на интерфейс
назначение *585*
наложение, общая форма *594*
применение *594*
на конструктор
new(), наложение *598*
назначение *586*
порядок наложения списком *603*
ссылочного типа
назначение *586*
наложение *599*
типа, неприкрытые
назначение *585*
наложение *602*
типа значения
назначение *586*
наложение *599*
Операторы
as *539*
break, применение *139*
continue, применение *142*
default *604*
goto
метки *143*
применение *143*
is *538*
new *153, 170*
return *143, 158*

sizeof *692*
stackalloc *692*
switch
вложенные *129*
обычные *125*
правило недопущения "провалов" *128*
typeof *540*
using *711*
yield return *1004*
арифметические *56, 97*
выбора *121*
вычисления остатка *98*
декремента *61, 98*
инкремента *61, 98*
итерационные *121*
логические
обычные *101*
укороченные *104*
нулеобъединяющие *698*
отношения *59, 101*
перегружаемые *269*
перехода *121*
поразрядные
обычные *107*
составные, присваивания *117*
предшествование *119*
преобразования
назначение *293*
явного, применение *295*
неявного, применение *295*
ограничения *296*
формы *293*
присваивания *55*
обычные *106*
укороченные *107*
составные *107*
сдвига *114*
цикла
do-while *138*
for *60, 129*
foreach *139, 194*
while *137*
Очередь
в качестве коллекции *948*
коэффициент роста *948*
применение *948*
принцип действия *947*

П

Параллелизм
данных *886*
задач *886*
Перегрузка
индексаторов *307*
конструкторов
преимущества *242*
причины *242*
методов
назначение *235*
главное преимущество *240*
операторных *277*
с несколькими параметрами типа *625*
по принципу полиморфизма *240*
операторов
унарных *273*
бинарных *270*
главное преимущество *269*
true и false *283*
ограничения *297*
логических *286*
укороченных, логических *288*
отношения *281*
определение *269*
основной принцип *297*
Переменные
внешние *486*
динамическая инициализация *84*
захваченные *486*
инициализация *83*
локальные *83*
неявно типизированные *85*
область действия *86*
обнуляемые
объявление *696*
присваивание значений *696*
проверка на пустое значение *696*
объявление типа *55*
определение *54*
ссылочного типа
назначение *153*
интерфейсного *381*
объявление *153*
присваивание *154*
статические *260*
форма и порядок объявления *83*
экземпляра, объявление *149*
Переполнение, появление *428*
Перечисления
базовый тип *399*
доступ к членам *397*
инициализация *399*
объявление *397*
определение *397*
применение *399*
Перечислители
доступ к коллекции *998*
назначение *924-925*

обычные, применение *999*
определение *998*
применение в цикле foreach *925*
типа IDictionaryEnumerator, применение *1000*
установка в исходное положение *998*
Полиморфизм
динамический *356*
основной принцип *43, 359*
Последовательности случайных чисел, генерирование *773*
Потоки
байтовые *432*
встроенные *432*
запоминающие *465*
исполнения *709, 834*
определение *431*
переадресация *441, 452*
символьные *432*
стандартные
ввода *432*
вывода *432*
сообщений об ошибках *432*
Предикаты
назначение *768*
применение *768*
Преобразование типов
в выражениях *93*
неявное, условия *90, 297*
перечислимых *397*
расширяющее *90*
сужающее *91*
явное *91*
Препроцессор, назначение *529*
Приведение типов
в выражениях *95*
как явное преобразование типов *89*
назначение *91*
Продвижение типов
неожиданные результаты *94*
правила *93*
целочисленное *94*
Проецирование *650*
Пространства имен
System *54, 719*
System.Collections *924*
System.IO *432*
System.Net *1011*
System.Reflection *541*
System.Threading *835*
System.Web *1012*
аддитивный характер *521*
вложенные *523*
глобальные *524*
назначение *51, 514*
объявление *514*
описатель псевдонима *525*
определение *513*
предотвращение конфликтов имен *516*
псевдонимы *520*
Пустая ссылка, определение *421*

P

Распаковка *370*
Рекурсия
главное преимущество *260*
определение *257*
принцип действия *258*
Рефлексия
вызов методов *548*
излечение типов данных из сборок *555*
назначение *541*
обнаружение типов, полностью автоматизированное *560*
получение списка
методов *544*
конструкторов *551*
применение *543*
принцип действия *541*

C

Сборки
автоматическое получение *535*
декларация *535*
дружественные *708*
метаданные типов *535*
назначение *535*
программные ресурсы *535*
программный код в формате MSIL *535*
составные разделы *535*
Свойства
автоматически реализуемые
общая форма *318*
ограничение доступа к аксессорам *321*
применение *319*
поддерживающее поле *319*
аксессоры get и set *313*
главное преимущество *313*
индексированные *707*
инициализаторы объектов, применение *319*
интерфейсные *383*
назначение *313*
общая форма *313*
ограничения *320*
Связный список
в качестве коллекции *965*
двунаправленный *965*

реализация *966*
узлы *966*
Семафоры
именованные *870*
назначение *867*
применение *868*
разрешение на доступ *867*
создание *867*
счетчики разрешений *867*
Сигнатуры, назначение *241*
Символы
в коде ASCII *74*
в уникоде *74, 742*
форматы UTF-16 и UTF-32 *742*
кодовая точка *742*
старший и младший суррогаты *742*
заполнители специального формата *820*
Синтаксические ошибки, обработка *53*
Система
"сборки мусора"
назначение *171*
номера поколений оперативной памяти *776*
применение *171*
ввода-вывода *431*
Скобки и пробелы, назначение *119*
Словарь
в качестве коллекции *969*
динамический характер *969*
создание *970*
События
аксессоры *500*
групповая адресация *496*
для синхронизации потоков, применение *870*
методы экземпляра как обработчики *497*
обработчики *494*
объявление *494*
порядок обработки *495*
практический пример обработки *509*
принцип действия *494*
разнообразные возможности *504*
рекомендации по обработке в .NET *506*
статические методы как обработчики *499*
управление списками вызовов обработчиков *500*
устанавливаемые
автоматически *870*
вручную *870*
хранение обработчиков *500*
Совместимость типов, принцип *351*
Сокрытие имен *89, 343, 388*
Спецификаторы
доступа *148, 210*
формата
назначение *813*
даты и времени *824*
числовых данных *814*
номера аргументов *815*
перечислений *830*
применение *813*
промежутков времени *829*
Среда .NET Framework
библиотека классов *38*
назначение *37*
общеязыковая среда выполнения CLR *37*
Среда CLR
JIT-компилятор *38*
метаданные *38*
назначение *38*
принцип действия *38*
псевдокод MSIL *38*
Стек
в качестве коллекции *945*
классический пример ООП *212*
основные операции *212*
применение *945*
принцип действия *212, 945*
Стиль оформления исходного кода *64*
Строки
в операторе switch *206*
вставка, удаление и замена *810*
заполнение и обрезка *808*
индексирование *201*
обращение *199*
операции *201*
определение *783*
поиск, методы *796*
получение подстрок *811*
постоянство *205, 784*
построение *198*
преобразование в лексемы *806*
разделение и соединение *804*
реализация в виде объектов *198*
смена регистра, методы *811*
сравнение
методы *787*
основные способы *786*
с учетом и без учета регистра *199*
с учетом культурной среды *199*
порядковое *199*
сцепление *203, 793*
числовые, преобразование *469*
Структурное программирование *32*
Структуры
Boolean, члены *748*

Char, члены *741*
Decimal, члены *735*
DictionaryEntry, члены *931*
KeyValuePair<TKey, TValue>, члены *960*
встроенных типов данных, в .NET *727*
инициализация *393*
назначение *391, 395*
обобщенные
наложение ограничений *606*
создание *606*
объявление *391*
применение *392, 395*
присваивание *393*
псевдонимы *469*
типов данных с плавающей точкой, члены *730*
целочисленных типов данных, члены *728*
числовых типов данных *470*

Т

Типы данных
анонимные *663*
десятичные *73*
динамические *703*
закрыто сконструированные *579*
закрытые *580*
значений *68*
логические *75*
наложение ограничений *585*
обнуляемые *601, 695, 697*
обобщенные *580*
ограниченные *585*
особенное значение *67*
открыто сконструированные *580*
параметризированные *576*
перечислимые *397*
полубайты, пример реализации *298*
простые *68*
символьные *74*
сконструированные *580*
соотносимые *682*
с плавающей точкой *57, 71*
ссылочные *68, 154*
строковые *198, 783*
целочисленные *55, 69*
частичные *700*
Точка с запятой, назначение *63*

У

Указатели
арифметические операции *686*
доступ к членам структуры *686*
и массивы *688*
индексирование *689*
и строки *690*
на указатели *691*
объявление *682*
оператор-стрелка *686*
операторы * и & *683*
определение *682*
сравнение *688*
файлов *461*
Упаковка *370*
Управляемый код *38, 682*
Управляющие операторы, категории *121*
Управляющие последовательности символов *81*
Условные операторы
? *117*
else *121*
if *58, 121*
вложенные, if *122*
многоступенчатая конструкция if-else-if *124*

Ф

Фабрики классов, назначение *233*
Флаг знака *69*
Форматирование
ввода-вывода *76*
даты и времени *824*
команды *77, 813*
образцы формата *78*
перечислений *830*
поставщики формата *812*
промежутков времени *829*
спецификаторы формата *77, 812*
строковое представление значения, способы получения *816*
форматирующие строки *77, 813*
Форматы
данных, специальные *820*
даты и времени, специальные *827*
изображения *820*
с обратным порядком байтов *771*
с прямым порядком байтов *771*

Х

Хеш-таблицы
в качестве коллекции *939*
коэффициент заполнения *939*
назначение *939*
применение хеш-кода *939*
Хэширование
механизм *939*
преимущество *939*

Ц

Цепочки
  вызовов *480*
  событий *497*
Циклы
  do-while *138*
  for *129*
  foreach *194*
  while *137*
  без тела *135*
  бесконечные *135*

Ш

Шестнадцатеричная система счисления *80*

Я

Язык C#
  генеалогическое дерево
    язык C *32*
    язык C++ *33*
    язык Java *33*
  история
    развития *36*
    создания *35*
  как хороший язык программирования *25*
  нововведения в версии C# 4.0 *37*
  происхождение *25*
  связь со средой .NET Framework *37*
  усовершенствование *26*
Язык LINQ
  сохранение данных во временной переменной *659*
  вложенные операторы from *653*
  группирование результатов запроса *655*
  групповое объединение, создание *666*
  деревья выражений *676*
  запросы
    связь между типами данных *642*
    выполнение *638*
    немедленное выполнение *675*
    неоднократное выполнение *641*
    обработка *642*
    общая форма *643*
    формирование *638*
    отложенное выполнение *675*
    определение *638*
  интерфейс IEnumerable, формы реализации *639*
  ключевые слова *643*
  методы
    расширения *673*
    запроса *669*
  назначение *637*
  обращение к источнику данных *639*
  объединение данных из разных источников *660*
  операторы
    from *640*
    group *655*
    into *657*
    join *661*
    let *659*
    orderby *646*
    select *643, 649*
    where *640, 644*
  отбор запрашиваемых данных *644*
  переменные
    диапазона *640*
    запроса *640*
  предикаты *640*
  продолжение запроса *657*
  сортировка результатов запроса *646*
  средства формирования запросов *637*
  формирование запросов
    методы запроса *670*
    сравнение способов *673*
    синтаксис запросов *670*
Язык PLINQ
  вопросы эффективности *922*
  другие средства *922*
  классы
    ParallelEnumerable *918*
    ParallelQuery *918*
  методы
    AsOrdered(), назначение и применение *919*
    AsParallel(), назначение и применение *918*
    WithCancellation(), назначение и применение *920*
  назначение *885*
  параллельные запросы
    отмена *920*
    формирование *918*
  применение *917*