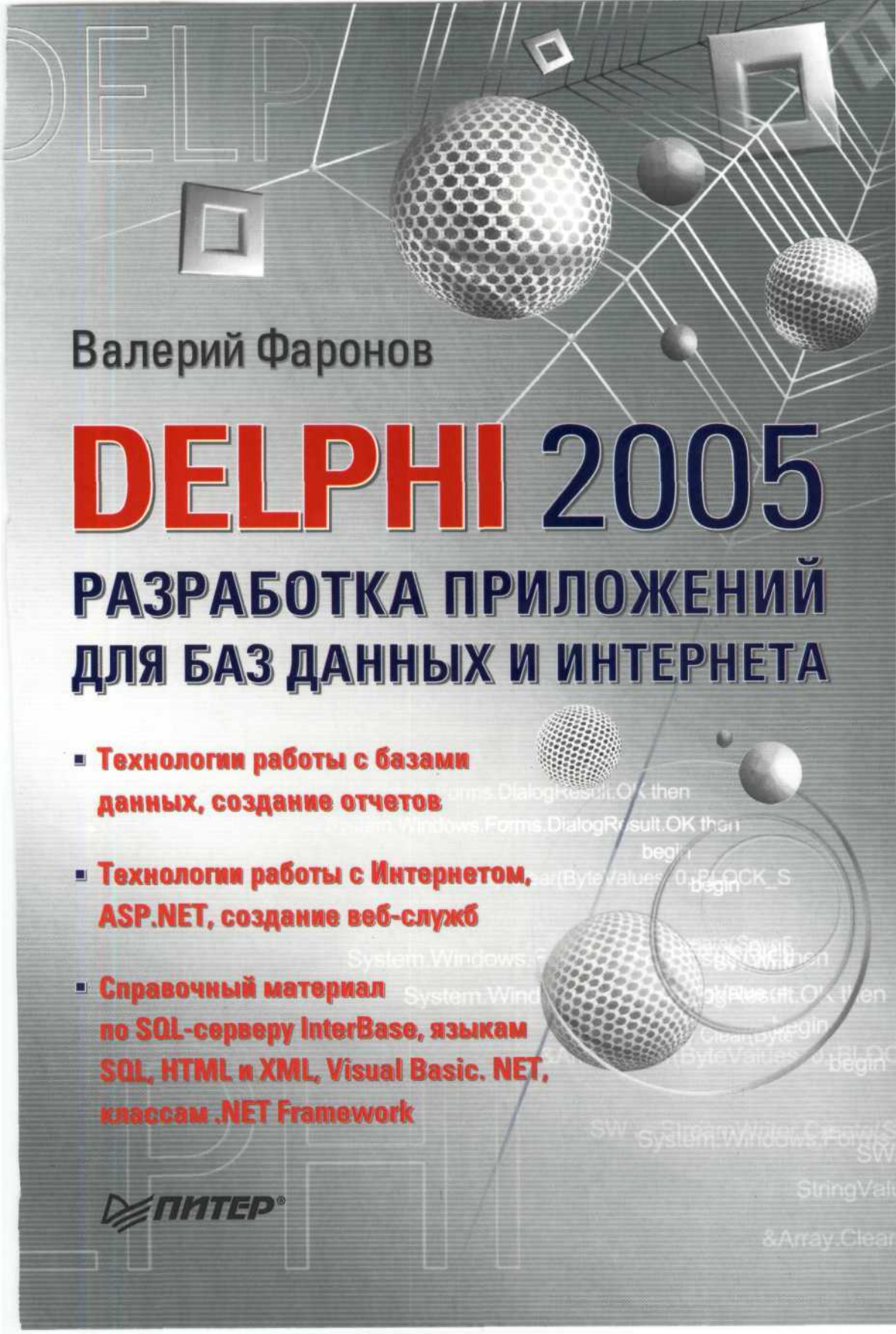


Валерий Фаронов

DELPHI 2005

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ БАЗ ДАННЫХ И ИНТЕРНЕТА

- Технологии работы с базами данных, создание отчетов
- Технологии работы с Интернетом, ASP.NET, создание веб-служб
- Справочный материал по SQL-серверу InterBase, языкам SQL, HTML и XML, Visual Basic. NET, классам .NET Framework

ПИТЕР®

Валерий Фаронов

DELPHI 2005

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ БАЗ ДАННЫХ И ИНТЕРНЕТА

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2006

ББК 32.973-018.2
УДК 004.43
Ф24

Фаронов В. В.

Ф24 Delphi 2005. Разработка приложений для баз данных и Интернета. — СПб.: Питер, 2006. — 603 с.: ил.

ISBN 5-469-01191-7

В книге самого известного в нашей стране автора по тематике Delphi на многочисленных примерах показывается применение системы программирования Delphi 2005 для создания двух наиболее важных типов приложений: для управления базами данных и для работы с Интернетом. Издание посвящено решению указанных задач на платформе .NET Framework с применением технологий ADO.NET и ASP.NET. Книга в основном рассчитана на читателя, имеющего опыт создания приложений Win32 и желающего освоить уникальные возможности платформы .NET Framework.

ББК 32.973-018.2
УДК 004.43

ISBN 5-469-01191-7

Краткое содержание

От автора 16
От издательства 18

Часть I. Создание приложений для работы с базами данных

Глава 1. Введение в базы данных 20
Глава 2. Технология ADO.NET 57
Глава 3. Технология BDE.NET 117
Глава 4. Технология dbGo.NET 211
Глава 5. Технология dbExpress.NET 242
Глава 6. Технология IBX.NET 268
Глава 7. Хранимые процедуры, триггеры и представления 291
Глава 8. Отчеты 308

Часть II. Создание приложений для работы с Интернетом

Глава 9. Программирование для Интернета и технология ASP.NET 357
Глава 10. Веб-формы и серверные элементы управления 369
Глава 11. Проверка данных 398
Глава 12. Работа с базами данных 406
Глава 13. Веб-службы 436
Глава 14. Создание пользовательских элементов управления 447
Глава 15. Защита приложений ASP.NET 455
Глава 16. Программирование приложений ASP.NET 468

Приложения

Приложение А. Сервер InterBase 484
Приложение Б. Краткая справка по языку SQL 514
Приложение В. Краткая справка по языкам HTML и XML 526
Приложение Г. Краткая справка по языку Visual Basic .NET 553
Приложение Д. Использование классов общего назначения платформы .NET Framework 559
Алфавитный указатель 594

Содержание

От автора 16

От издательства 18

Часть I. Создание приложений для работы с базами данных

Глава 1. Введение в базы данных 20

Типы СУБД 20
Таблицы БД и связи между ними 23
Первичные ключи и индексы 23
Создание БД 24
Демонстрационная БД «Книголюб» 26
Анализ информационных потоков 26
Проектирование БД 27
Имена таблиц и полей 30
Механизмы BDE и ODBC 31
Создание таблиц файл-серверных БД 31
Свойства таблиц Paradox 7 35
Типы полей 35
Контроль содержимого полей 36
Таблица подстановки 36
Вторичные индексы 37
Ссылочная целостность 38
Парольная защита 39
Выбор языкового драйвера 39
Создание таблиц клиент-серверных БД 40
Пример простой программы 43
Разработка главной формы 43
Создание псевдонима БД 44

Связь с данными . . . 47
Создание объектов-полей . . . 49
Создание объектов-столбцов . . . 52
Обработчик события OnGetText . . . 53
Бизнес-правила . . . 54
Транзакции . . . 55

Глава 2. Технология ADO.NET . . . 57

Классы ADO.NET . . . 58
Соединенные классы . . . 58
Разъединенные классы . . . 59
Провайдеры данных ADO.NET . . . 60
Провайдеры Microsoft . . . 60
Провайдер Borland . . . 61
Простой пример . . . 61
Конструирование проекта . . . 61
Настройка набора данных . . . 63
Прогон программы . . . 65
Применение объекта Connection . . . 66
Задание значения свойства ConnectionString . . . 67
Работа с транзакциями . . . 68
Получение метаданных . . . 68
События соединения . . . 69
Применение объектов Command и DataReader . . . 69
Использование метода ExecuteNonQuery . . . 70
Выполнение параметрических запросов . . . 71
Получение множества результатов запроса . . . 73
Класс DataAdapter . . . 75
Свойства, методы и события класса . . . 76
DataSet — набор данных . . . 80
DataTable — таблица НД . . . 83
DataColumn — поля таблиц . . . 86
Вычисляемые поля . . . 88
Подстановочные поля . . . 91
Изменение названия и скрытие полей . . . 92
DataRow — записи . . . 95
Визуализация данных . . . 99
DataGrid — сетка данных . . . 99
Использование свойства DataBindings визуальных компонентов . . . 104
Программный доступ к значениям таблиц . . . 112

Глава 3. Технология BDE.NET . . . 117

Суть технологии . . . 117
Поля . . . 118
Обзор свойств, методов и событий . . . 118
Использование объектов-полей . . . 125

Проверка правильности введенного в поле значения . . . 132
Формирование текстового представления поля . . . 133
Обзор полей TxxxField . . . 134
Компоненты TSession и TDatabase. Транзакции . . . 139
Компонент TSession . . . 139
Компонент TDatabase . . . 145
Транзакции . . . 146
Наборы данных . . . 149
Обзор свойств, методов и событий . . . 149
Открытие и закрытие набора данных . . . 155
Программный доступ к записям . . . 155
Навигация по набору данных . . . 157
Поиск записей в наборах данных . . . 159
Фильтрация записей . . . 163
Блокировка таблиц в многопользовательском режиме . . . 165
Обзор событий . . . 166
Таблицы . . . 167
Обзор свойств и методов . . . 167
Индексы . . . 171
Эксклюзивный доступ к таблице . . . 173
Удаление записей и таблиц . . . 173
Поиск записей в таблице . . . 173
Выборка записей . . . 174
Запросы . . . 175
Обзор свойств . . . 176
Обзор методов . . . 178
Свойство SQL . . . 179
Методы Open и ExecSQL . . . 179
Параметрические запросы . . . 180
Методы Prepare и UnPrepare . . . 181
Изменяемые запросы . . . 182
Сортировка в обратном порядке . . . 184
Визуализация данных . . . 185
Компонент TDataSource . . . 185
Компонент TDBGrid . . . 187
Компоненты для визуализации полей текущей записи . . . 201

Глава 4. Технология dbGo.NET . . . 211

Пример простой программы . . . 213
Установление связи с объектом ADO . . . 216
Структура строки связи . . . 217
Формирование строки связи . . . 217
Особенности использования компонентов dbGo.NET . . . 222
Базовые объекты ADO . . . 222
Связной компонент TADOConnection . . . 223
Работа с транзакциями . . . 225

Компонент TADOCommand 232
Свойства, методы и события компонентов-наборов 234
Компонент TADODataSet 239
Компонент TADOTable 240
Компонент TADOQuery 241

Глава 5. Технология dbExpress.NET 242

Пример простой программы 243
Компоненты для реализации технологии 247
Компонент TSQLConnection 248
Компонент TSQLDataSet 254
Компонент TSQLQuery 257
Компонент TSQLTable 257
Компонент TSQLMonitor 257
Компонент TSimpleDataSet 259

Глава 6. Технология IBX.NET 268

Пример простой программы 268
Компоненты для реализации технологии 271
Класс TIBBase 271
Компонент TIBDatabase 272
Компонент TIBTransaction 276
Компонент TIBTable 280
Компонент TIBQuery 281
Компонент TIBDataSet 281
Компонент TIBSQL 285
Компонент TIBDatabaseInfo 287
Компонент TIBSQLMonitor 289
Компонент TIBExtract 289

Глава 7. Хранимые процедуры, триггеры и представления 291

Создание хранимых процедур 291
Алгоритмический язык процедур и триггеров 293
Локальные переменные 293
Операторные скобки 294
Оператор SELECT 294
Условный оператор 294
Операторы FOR и SUSPEND 294
Оператор присваивания 295
Операторы WHILE и EXIT 295
Оператор EXECUTE PROCEDURE 295
Исключения 296
Компоненты доступа к хранимым процедурам 297
Компонент TStoredProc 297
Компонент TADOStoredProc 300
Компонент TSQLStoredProc 301
Компонент TIBStoredProc 301

Создание триггеров . . . 302
Реализация бизнес-правил с помощью триггеров . . . 302
Изменение и удаление процедур и триггеров . . . 306
Представления . . . 306

Глава 8. Отчеты . . . 308

Основы технологии Crystal Reports . . . 308
Пример создания простого отчета . . . 308
Дополнительные средства эксперта создания стандартного отчета . . . 315
Экспорт отчета . . . 322
Основы технологии Rave Reports . . . 323
Пример создания отчета . . . 323
Привязка проекта отчета к приложению . . . 331
Визуальная среда Rave Reports Designer . . . 332
Составляющие проекта отчета . . . 333
Объекты данных . . . 336
Импорт в отчет произвольных внешних файлов . . . 339
Защита данных . . . 341
Типы отчетов . . . 342
Использование агрегатных функций . . . 351
Экспорт отчета в файл . . . 353

Часть II. Создание приложений для работы с Интернетом

Глава 9. Программирование для Интернета и технология ASP.NET 357

Основы сетевого программирования . . . 357
Средства . . . 357
Некоторые детали протокола HTTP . . . 358
Общая схема обработки запроса клиента . . . 360
Введение в технологию ASP.NET . . . 361
Назначение и архитектура технологии ASP.NET . . . 361
Возможности технологии ASP.NET . . . 365
Директива Page . . . 366
Атрибут CodeBehind . . . 367
Сценарии . . . 368

Глава 10. Веб-формы и серверные элементы управления . . . 369

Веб-формы . . . 369
Серверные элементы управления . . . 372
Компоненты категории HTML Elements . . . 373
Компоненты категории Web Controls . . . 374

Глава 11. Проверка данных . . . 398

Две формы проверки . . . 398
Настройка клиентской проверки . . . 399
Отключение клиентской проверки . . . 399

Элемент RequiredFieldValidator 399
Элемент RangeValidator 401
Элемент RegularExpressionValidator 402
Элемент CompareValidator 403
Элемент ValidationSummary 404
Элемент CustomValidator 405

Глава 12. Работа с базами данных 406

Доступ к данным 406
Компоненты доступа к MS SQL Server 406
Компоненты доступа к другим источникам данных 407
Визуализация данных 407
Простой пример 407
Стандартные элементы для работы с базами данных 409
Привязка данных 410
Элемент Repeater 412
Элемент DataList 415
Элемент DataGrid 417
Элементы категории DB Web 428
Элемент DBWebDataSource 428
Элемент DBWebNavigator 431
Элемент DBWebGrid 432
Визуализирующие элементы 433

Глава 13. Веб-службы 436

Создание веб-служб 436
Атрибут [WebService] 442
Атрибут [WebMethod] 442
Использование веб-служб 443
Создание прокси-класса 443
Использование прокси-класса 446

Глава 14. Создание пользовательских элементов управления . . 447

Пример простого пользовательского элемента управления 447
Исследование простого элемента управления 450
Элемент управления для регистрации пользователя 451

Глава 15. Защита приложений ASP.NET 455

Способы защиты приложений ASP.NET 455
Аутентификация Windows 456
Аутентификация на основе форм 457
Изменения в файле Web.config 458
Индивидуальная защита страниц 458
Создание страницы регистрации 459
Создание регистрационного удостоверения пользователя 459
Хранение аутентификационной информации в файле Web.config 463

Хранение регистрационной информации в XML-файле 465
Хранение регистрационной информации в БД 466
Аутентификация по паспорту 467

Глава 16. Программирование приложений ASP.NET 468
Объекты и классы приложений ASP.NET 468
Класс Page 468
Класс HttpRequest 471
Класс HttpResponse 475
Состояние вида 478
Поле _ViewState 479
Управление состоянием на уровне сеанса 480
Управление состоянием на уровне приложения 482

Приложения

Приложение А. Сервер InterBase 484
Назначение и возможности 484
Некоторые технические характеристики 485
Физическая организация базы данных InterBase 486
Типы данных InterBase 488
Обзор типов данных InterBase 488
Столбцы-массивы 490
Типы DECIMAL и NUMERIC 491
Тип DATE 491
Типы CHAR и VARCHAR 492
Тип BLOB 494
Денежные столбцы 495
Генераторы 495
Совместимость типов 496
Домены 496
Ограничения на значения столбцов 497
Ручное администрирование сервера 498
Программное администрирование сервера 503
Базовые классы 503
Компонент TIBConfigService 505
Компонент TIBBackupService 506
Компонент TIBRestoreService 507
Компонент TIBValidationService 507
Компонент TIBStatisticalService 508
Компонент TIBLogService 509
Компонент TIBSecurityService 509
Компонент TIBServerProperties 510
Компонент TIBLicensingService 510
Компонент TIBInstall 511
Компонент TIBUnInstall 513

Приложение Б. Краткая справка по языку SQL 514
Простая выборка данных 514
Выборка из связанных таблиц 515
Сортировка записей 517
Сложные критерии отбора 517
Псевдонимы полей, таблиц и комментарии 520
Агрегатные функции и группировка записей 520
Создание/удаление таблиц и индексов 523
Изменение таблиц 524
Вставка, удаление и редактирование записей 524

Приложение В. Краткая справка по языкам HTML и XML 526
Знакомство с языком HTML 526
Система тегов 526
Гиперссылки 528
Шрифты 529
Списки 529
Изображения 531
Уточняющие параметры и цвет 531
Комментарии 532
Диалоговые средства 532
Таблицы 535
Фреймы 537
Другие возможности 539
Знакомство с языком XML 540
Причины разработки XML 540
Структура XML-документа 541
Простой пример 544
Шаблон преобразования 545
Обработка таблицы 547
Сортировка 549
Фильтрация 551
Концепция объектной модели документа 552

Приложение Г. Краткая справка по языку Visual Basic .NET . . . 553
Синтаксис языка 553
Типы данных 554
Объявления переменных 554
Массивы 554
Преобразования типов 555
Выражения и операции 556
Операторы присваивания 556
Условный оператор 557
Оператор выбора 557
Оператор For 558
Оператор While 558
Оператор Do 558

Приложение Д. Использование классов общего назначения платформы .NET Framework 559
Коллекции 559
Интерфейсы пространства имен System.Collection 559
Классы пространства имен System.Collection 562
Обработка строк 572
Преобразование значений других типов в строку и обратно 572
Форматирование строк 572
Использование методов и свойств строк 576
Работа с файловой системой 580
Классы для работы с файловой системой 581
Запись и чтение файлов 587
Сериализация 590
Техника сериализации 590
Пример 591

Алфавитный указатель 594

От автора

Появление платформы .NET Framework является, безусловно, самым революционным событием в мире программирования за последние 10 лет. Разработчики корпорации Microsoft, создавшие эту платформу, постарались учесть накопленный опыт программирования самых разных приложений и создали уникальный инструмент, позволяющий значительно облегчить и ускорить их разработку. Пожалуй, наиболее важная составляющая новой платформы — общая система типов (Common System Type, CTS). Входящие в ее состав более 4000 типов и классов составляют «кирпичики» повторно используемого кода, решающие практически любые задачи при создании самых разных приложений.

В сегодняшнем мире, объединенном «паутиной» Интернета, наиболее востребованной является задача создания веб-приложений. Разработчики .NET Framework подчеркивают нацеленность данной платформы на решение именно этой задачи. Кроме того, программистам очень часто приходится решать задачу управления данными. В .NET Framework существует множество классов, способных обеспечить доступ к разнообразным данным, представленным в форме файловых таблиц, серверных таблиц, XML-файлов и т. д. Этим двум задачам — созданию приложений для работы с данными и приложений для Интернета — и посвящена книга, которую вы держите в своих руках.

Система программирования Delphi 2005 американской корпорации Borland приспособлена как для создания обычных приложений Win32, так и для использования платформы .NET Framework. Предполагается, что вы знакомы с этой системой (средой и языком программирования) по другим книгам. (В качестве необязательной ссылки сошлюсь на мою книгу «Delphi 2005. Язык, среда, разработка приложений», выпущенную в конце мая 2005 года в издательстве «Питер».) И хотя Delphi 2005 может применяться для создания приложений Win32, в настоящей книге эта возможность не описывается: книга полностью посвящена решению

указанных задач на платформе .NET Framework. Более того, технология ASP.NET, которая рассматривается во второй части книги, может работать только на этой платформе.

В заключение скажу, что сжатые сроки создания книги и сложность рассматриваемых в ней технологий не позволили написать ее в форме доступного учебника — судя по всему, это задача будущего. Книга во многом рассчитана на искушенного читателя, имеющего опыт создания приложений Win32 и желающего освоить уникальные возможности платформы .NET Framework. Буду искренне рад, если она поможет вам в этом.

Москва, 20 августа 2005 г.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу: http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: http://www.piter.com.

Создание приложений для работы с базами данных

Delphi имеет развитые средства создания систем управления базами данных. Речь идет прежде всего о технологии ADO.NET, которая поддерживается в CLR (Common Language Run-time — общеязыковая среда исполнения в технологии .NET). Но возможности Delphi не исчерпываются только этой технологией. В предыдущих версиях Delphi созданы различные альтернативные технологии, подчас более эффективные, чем ADO.NET. Эти технологии перенесены в Delphi 2005 и, как и ADO.NET, описываются в главах этой части.

1 Введение в базы данных

Базами данных (БД) называют электронные хранилища информации, доступ к которым осуществляется с одного или нескольких компьютеров. Обычно БД создается для хранения данных, содержащих сведения о некоторой предметной области, то есть некоторой области человеческой деятельности или области реального мира, и доступа к ним.

Типы СУБД

Системы управления базами данных (СУБД) — это программные средства, предназначенные для создания, наполнения, обновления и удаления баз данных. Различают три основных вида СУБД: промышленные универсального назначения, промышленные специального назначения и разрабатываемые для конкретного заказчика. Специализированные СУБД создаются для управления базами данных конкретного назначения — бухгалтерскими, складскими, банковскими и т. д. Универсальные СУБД не имеют четко очерченных рамок применения, они рассчитаны «на все случаи жизни» и, как следствие, достаточно сложны и требуют от пользователя специальных знаний. Как специализированные, так и универсальные промышленные СУБД относительно дешевы, довольно надежны (отлажены) и готовы к немедленной работе, в то время как заказные СУБД требуют существенных затрат, а их подготовка к работе и отладка занимают значительный период времени (от нескольких месяцев до нескольких лет). Однако, в отличие от промышленных, заказные СУБД в максимальной степени учитывают специфику работы заказчика (того или иного предприятия), их интерфейс обычно интуитивно понятен пользователям и не требует от них специальных знаний.

По своей *архитектуре* СУБД делятся на одно-, двух- и трехзвенные (рис. 1.1). В однозвенной архитектуре используется единственное звено (клиент), обеспечивающее необходимую логику управления данными и их визуализацию. В двухзвенной

архитектуре значительную часть логики управления данными берет на себя сервер БД, в то время как клиент в основном занят отображением данных в удобном для пользователя виде. В трехзвенных СУБД имеется промежуточное звено — сервер приложений, являющееся посредником между клиентом и сервером БД. Сервер приложений призван полностью избавить клиента от каких бы то ни было забот по управлению данными и обеспечению связи с сервером БД.

Рис. 1.1. Архитектура СУБД: слева — однозвенная; в центре — двухзвенная; справа — трехзвенная

В зависимости от местоположения отдельных частей СУБД различают *локальные* и *сетевые* СУБД.

Все части локальной СУБД размещаются на компьютере пользователя базы данных. Чтобы с одной и той же БД одновременно могли работать несколько пользователей, каждый пользовательский компьютер должен иметь свою копию локальной БД. Существенной проблемой СУБД такого типа является синхронизация копий данных, именно поэтому для решения задач, требующих совместной работы нескольких пользователей, локальные СУБД практически не применяются.

К сетевым относятся *файл-серверные*, *клиент-серверные* и *распределенные* СУБД. Непременным атрибутом этих систем является сеть, обеспечивающая аппаратную связь компьютеров и делающая возможной корпоративную работу множества пользователей с одними и теми же данными.

В файл-серверных СУБД все данные обычно размещаются в одном или нескольких каталогах достаточно мощной машины, специально выделенной для этих целей и постоянно подключенной к сети. Такой компьютер называется *файл-сервером* — отсюда название СУБД. Безусловным достоинством СУБД этого типа является относительная простота ее создания и обслуживания — фактически все сводится лишь к развертыванию локальной сети и установке на подключенных к ней компьютерах сетевых операционных систем. По счастью, Delphi «умеет»

использовать сетевые средства самой популярной в мире ОС — Windows для создания соответствующих *клиентских мест*, то есть специального программного обеспечения компьютеров пользователей. Нетрудно заметить, что между локальными и файл-серверными вариантами СУБД нет особых различий, так как в них все части собственно СУБД (кроме данных) находятся на компьютере клиента. По архитектуре они обычно являются однозвенными, но в некоторых случаях могут использовать сервер приложений. Недостатком файл-серверных систем является значительная нагрузка на сеть. Если, например, клиенту нужно отыскать сведения об одной из фирм-партнеров, по сети вначале передается весь файл, содержащий сведения о многих сотнях партнеров, и лишь затем в созданной таким образом локальной копии данных отыскивается нужная запись. Ясно, что при интенсивной работе с данными уже нескольких десятков клиентов пропускная способность сети может оказаться недостаточной, и пользователя будут раздражать значительные задержки в реакции СУБД на его запросы. Файл-серверные СУБД могут успешно использоваться в относительно небольших фирмах с количеством клиентских мест до нескольких десятков.

Клиент-серверные (двухзвенные) системы значительно снижают нагрузку на сеть, так как клиент общается с данными через специализированного посредника — *сервер базы данных*, который размещается на машине с данными. Сервер БД принимает запрос от клиента, отыскивает в данных нужную запись и передает ее клиенту. Таким образом, по сети передаются относительно короткий запрос и единственная нужная запись, даже если соответствующий файл с данными содержит сотни тысяч записей. Запрос к серверу формируется на специальном языке структурированных запросов (Structured Query Language, SQL, произносится «*сиквэл*»), поэтому часто серверы БД называются SQL-серверами. Серверы БД представляют собой относительно сложные программы, разрабатываемые различными фирмами. К таким программам относятся, например, Microsoft SQL Server производства корпорации Microsoft, Sybase SQL Server корпорации Sybase, Oracle производства одноименной корпорации[1], DB2 корпорации IBM и т. д. SQL-сервером является также и сервер InterBase корпорации Borland, который поставляется вместе с Delphi. Клиент-серверные СУБД масштабируются до сотен и тысяч клиентских мест.

Распределенные СУБД могут содержать несколько десятков и сотен серверов БД. Количество клиентских мест в них может достигать десятков и сотен тысяч. Обычно такие СУБД работают на предприятиях государственного масштаба, отдельные подразделения которых разнесены на значительной территории. К таковым, например, относятся подразделения Министерства обороны и Министерства внутренних дел. В распределенных СУБД некоторые серверы могут дублировать друг друга с целью достижения предельно малой вероятности отказов и сбоев, которые могут исказить жизненно важную информацию. Они используют собственные региональные средства связи. Интерес к распределенным СУБД возрос в связи со стремительным развитием Интернета. Опираясь на его возможности, распределенные системы строят не только предприятия государственного масштаба, но

[1] Кстати, Oracle является второй по объемам доходов корпорацией, занимающейся исключительно созданием программного обеспечения (первой является, конечно, Microsoft).

и относительно небольшие коммерческие предприятия, обеспечивая своим сотрудникам работу с корпоративными данными на дому и в командировках.

Таблицы БД и связи между ними

Единицей хранящейся в БД информации является *таблица*. Каждая таблица представляет собой совокупность строк и столбцов, где строки соответствуют экземпляру объекта, конкретному событию или явлению, а столбцы — атрибутам (признакам, характеристикам, параметрам) этого объекта, события, явления. Ниже приведен пример таблицы, в которой содержатся сведения о продаже книг со склада. Столбцы описывают такие параметры, как дата продажи, название проданной книги, наименование покупателя, количество проданных ему книг. Каждая строка содержит сведения о конкретном событии — продаже книги покупателю. В терминах БД столбцы таблицы называются *полями*, а ее строки — *записями*.

Дата	Название книги	Покупатель	Отпущено
10.12.99	Borland C++ Builder 4	Магазин № 1	100
10.12.99	Delphi 5. Учебный курс	Магазин № 1	100
12.12.99	В сетях всемирной паутины	Дом книги	2000

Между отдельными таблицами БД могут существовать связи. Например, информация о покупателе в предыдущей таблице может дополняться в другой таблице.

Покупатель	Адрес	Телефон
Магазин № 1	107005, Москва, 2-я Бауманская ул., 12	273-00-14
Дом книги	105066, Москва, Измайловский б-р, 18/11	165-18-99

Базы данных, между отдельными таблицами которых существуют связи, называются *реляционными* (от *relation* — связь, отношение).

Связанные отношениями таблицы взаимодействуют по принципу *главная* (master) — *детальная* (detail)[1]. В нашем примере таблица покупателей — главная, а таблица отпуска товаров — детальная. Главную таблицу часто называют *родительской*, а детальную — *дочерней*. Одна и та же таблица может быть главной по отношению к одной таблице БД и дочерней — по отношению к другой.

Первичные ключи и индексы

В каждой таблице БД может существовать *первичный ключ* — поле или набор полей, однозначно идентифицирующий запись. Значение первичного ключа в таблице БД должно быть уникальным, то есть в таблице не должно существовать двух или более записей с одинаковым значением первичного ключа.

[1] Отношение «главная—детальная» иногда называют еще отношением «главная—подчиненная».

Первичные ключи облегчают установление связи между таблицами. В таблице покупателей таким ключом может быть одноименное поле. Установив связь по первичному ключу, мы можем выяснить, что, например, 10.12.99 со склада было отпущено 100 единиц книг «Borland C++ Builder 4» покупателю «Магазин № 1», который расположен по адресу: 107005, Москва, 2-я Бауманская ул., 12 (телефон для связи 273-00-14).

Поскольку первичный ключ должен быть уникальным, для него могут использоваться не все поля таблицы. В приведенном примере название покупателя вряд ли может быть уникальным («Магазин № 1» может существовать не только в Москве, но и в любом другом городе), поэтому поле `Покупатель` не может использоваться в качестве первичного ключа. Значительно более редким является совпадение телефонов у двух разных покупателей, поэтому поле `Телефон` в бóльшей степени подходит на роль первичного ключа. Если в таблице нет полей с уникальными значениями, для создания первичного ключа в нее обычно вводят дополнительное числовое поле, значениями которого СУБД может распоряжаться по своему усмотрению. Если, например, в таблицу покупателей добавить поле №, то она могла бы выглядеть так.

№	Покупатель	Адрес	Телефон
1	Магазин №1	107005, Москва, 2-я Бауманская ул., 12	273-00-14
2	Дом книги	105066, Москва, Измайловский б-р, 18/11	165-18-99

Соответственно изменилась бы и связанная с ней таблица отпуска товаров.

Дата	Название книги	Покупатель	Отпущено
10.12.99	Borland C++ Builder 4	1	100
10.12.99	Delphi 5. Учебный курс	1	100
12.12.99	В сетях всемирной паутины	2	2000

Теперь в таблице отпуска товаров в поле `Покупатель` указывается значение первичного ключа, построенного по полю № таблицы покупателей, что позволяет установить однозначную связь между таблицами.

Индексы отличаются от первичных ключей тем, что не требуют непременной уникальности значений входящих в их состав полей. Они устанавливаются по полям, которые часто используются при поиске и сортировке данных: индексы помогут системе значительно быстрее найти нужные данные или отсортировать их в нужной последовательности.

Создание БД

Создать саму БД (не СУБД) средствами Delphi очень сложно, даже невозможно. Действительно, если вы выбрали двух- или трехзвенную архитектуру, вам не обойтись без сервера БД, который создать собственными силами очень трудно

(да и ни к чему, если на рынке предлагаются десятки таких программ, а в Интернете при желании можно найти и бесплатный, но вполне приличный сервер MySQL). Если речь идет о файл-серверной БД, то и здесь понадобятся специальные средства. В Delphi для этих целей обычно используется утилита Database Desktop. Могут также применяться промышленные СУБД «Paradox», «dBASE» (корпорации Borland), «FoxPro», «Access» (корпорации Microsoft) и др. В рамках этой книги я буду описывать средства Database Desktop и сервер InterBase, так как они входят в комплект поставки Delphi и являются собственными продуктами корпорации Borland.

Создание БД начинается с ее проектирования. На этом этапе анализируются информационные потоки с целью определить набор таблиц и их полей. Процесс проектирования в немалой степени зависит от опыта и интуиции разработчика, то есть является творческим, однако некоторые его моменты можно формализовать. Одной из таких формализаций является требование, согласно которому реляционная база данных должна быть *нормализована*.

Нормализованной называется БД, в которой выполняется как минимум 3 условия. Выполнение условия называется приведением БД к соответствующей *нормальной форме*.

Первая нормальная форма (1НФ) требует, чтобы каждое поле таблицы было неделимым и не содержало повторяющихся групп. Пусть в таблице хранится информация о книгах. Эту информацию можно держать в единственном поле, но это противоречит 1НФ. В соответствии с ней в таблице должны быть поля `Название`, `Автор`, `Издательство` и др., совокупность которых дает исчерпывающую информацию о книге. Это облегчает реализацию таких запросов к БД, которые позволят узнать количество разных авторов, количество книг одного автора, количество книг, выпущенных определенным издательством, и т. п. Повторяющимися являются поля, содержащие сходную информацию. Если, например, таблица содержит информацию о покупателе и купленных им книгах, то, в соответствии с 1НФ, список книг должен выделяться в отдельную таблицу. В этом случае таблица покупателей будет главной, а таблица купленных ими книг — детальной. Между такими таблицами должна устанавливаться реляционная связь *один ко многим*.

Вторая нормальная форма (2НФ) требует, чтобы все поля одной таблицы зависели от первичного ключа — только в этом случае первичный ключ, как и требуется, будет определять уникальную информацию. Допустим, что мы в таблице отпуска книг составили первичный ключ по полям `Номер накладной` и `Покупатель`. Нетрудно заметить, что такой ключ будет избыточным: поле `Покупатель` однозначно определяется номером накладной, поэтому, в соответствии с 2НФ, оно не может входить в первичный ключ.

Третья нормальная форма (3НФ) требует, чтобы значение любого поля, не входящего в первичный ключ, никак не зависело от значения других полей. Если в таблице накладных содержатся поля `Книга` и `Отпускная цена`, второе поле явно зависит от первого и по требованию 3НФ должно быть вынесено в отдельную таблицу. Общее правило выполнения 3НФ таково: в таблице должны быть только не зависящие друг от друга поля, однозначно определяющие некоторый факт, вся

дополнительная информация о значении полей должна выноситься в отдельные таблицы. Например, в той же накладной на продажу книг должно быть поле Покупатель, но не должно быть характеризующих его полей Город, Адрес, Телефон и т. п. — эти данные должны быть в отдельной таблице Покупатели.

Как уже говорилось, проектирование БД — творческий процесс, а описанные выше требования носят лишь рекомендательный характер. На практике безусловное выполнение этих требований (в особенности 3НФ) приводит к появлению множества небольших по объему таблиц и может существенно замедлять работу с СУБД. Для повышения скорости ее работы проектировщики часто идут на сознательное нарушение описанных требований.

Демонстрационная БД «Книголюб»

Большинство примеров в этой книге относятся к демонстрационной БД «Книголюб», система управления которой предназначена для автоматизации работы крупного оптового поставщика книг. В этом разделе я попытаюсь описать основные функции оптового поставщика и создать проект соответствующей БД. Среди файлов к книге, которые можно загрузить с сайта издательства «Питер» (http://www.piter.com), вы найдете два варианта этой БД (для файл-серверной СУБД и для СУБД с сервером InterBase). Все хранящиеся в ней данные взяты из реально существующей БД, однако информация о партнерах (поставщиках и покупателях) намеренно искажена.

Анализ информационных потоков

Оптовый поставщик является промежуточным звеном между издательствами (поставщики книг) и магазинами (покупатели книг). Наличие этого звена выгодно тем и другим: издательство, выпустив книгу, передает весь ее тираж или значительную его часть оптовому поставщику и, таким образом, не заботится об отслеживании многочисленных связей с магазинами; магазины, в свою очередь, находят у оптового поставщика огромный ассортимент книг, выпущенных многочисленными издательствами не только в России, но и в ближнем зарубежье.

На рис. 1.2 отображены взаимосвязи между оптовым поставщиком и его партнерами. Характерной особенностью является двусторонний обмен книгами как с поставщиками, так и с покупателями. Это связано с тем, что большинство покупателей (магазинов) берут книги без предварительной оплаты, обязуясь реализовать их в определенный срок. По истечении этого срока магазин обязан оплатить взятые им книги и, возможно, вернуть не проданные книги оптовому поставщику. На таких же условиях оптовый поставщик берет книги у издательств.

Итак, существует два вида документов, которыми обменивается оптовый поставщик со своими партнерами, — это накладные на отпуск, покупку или возврат книг и платежные извещения. В накладных указывается, кому, сколько и каких книг продано (или куплено), в платежных извещениях — суммы платежей и наименование партнера. Характерно, что в них обычно не указывается, за какие конкретные книги осуществляется платеж, — СУБД должна автоматически перераспределить

сумму платежа на книги, указанные в накладных, в соответствии с обычным правилом: оплачиваются самые ранние накладные по мере их поступления.

Рис. 1.2. Взаимосвязи оптового поставщика книг с партнерами

Проектирование БД

После анализа особенностей автоматизируемой области деятельности следует приступить к, возможно, самому важному этапу — проектированию будущей БД, которое заключается в определении состава полей ее таблиц и связей между таблицами. От того, насколько тщательно проведен анализ и насколько грамотно спроектирована БД, в существеннейшей мере зависят эффективность будущей СУБД и ее полезность для пользователя.

В нашем случае анализ показывает, что в БД должно быть как минимум 5 таблиц. В таблице `FIRMS` будут храниться все нужные сведения о партнерах — с указанием юридического адреса, контактных лиц, телефонов и, разумеется, полного названия каждого партнера. В этой же таблице будем хранить суммарный долг каждого покупателя (или каждому поставщику), который обычно называется *сальдо*. В таблице `BOOKS` разместим полные сведения о каждой книге, хотя бы раз купленной у какого-либо поставщика или полученной по обмену от другого оптового поставщика (практика обмена книгами между оптовыми поставщиками широко распространена; за книги, полученные или переданные по обмену, платежи не осуществляются). Таблица `NAKLS` будет предназначена для хранения сведений о накладных. В ней будут поля, в которых СУБД поместит дату отгрузки или получения партии книг, тип накладной (на покупку или продажу, с предоплатой или в рассрочку, с возвратом ранее проданных/купленных книг или с передачей их по обмену), наименование партнера, общую сумму накладной и т. п. Тут же возникает вопрос: а каким образом СУБД будет хранить сведения о собственно получаемых или передаваемых по накладной книгах? Ведь их состав и количество могут быть какими угодно: если осуществляется покупка книг у поставщика, в накладной обычно указываются лишь одна-две книги, которые покупаются крупными партиями; если книги покупает магазин, он может закупить до сотни наименований книг относительно небольшими партиями; не исключена и розничная продажа одной-единственной книги случайному покупателю (я сам много раз

покупал одну-две книги по относительно небольшой цене у того или иного оптового поставщика). Ясно, что в таблице NAKLS, как и в любой другой таблице БД, не может быть переменного количества полей, поэтому, в соответствии с 1НФ, сведения о связанных с накладной книгах будем хранить в отдельной таблице MOVES: в ее полях укажем ссылку на соответствующую накладную, наименование и количество переданной/полученной книги. Наконец, в таблицу PAYMENTS поместим сведения о платежах: кто, кому и сколько платит.

Таким образом, таблица NAKLS оказывается центральной. Она должна иметь уникальное поле, которое будет однозначно определять каждую накладную. В дальнейшем по этому полю мы создадим первичный ключ, чтобы СУБД могла быстро найти нужную накладную. Каждой записи в NAKLS будет соответствовать произвольное количество записей в таблице MOVES (такая связь в терминологии БД называется связью *один ко многим*). В этой таблице ссылка на главную таблицу NAKLS определяется тем, что одно из ее полей будет содержать уникальный идентификатор накладной. По этому полю следует создать индекс, чтобы СУБД смогла быстро отыскать все книги, связанные с той или иной накладной. В таблице NAKLS будет также ссылка на уникальный идентификатор партнера из таблицы FIRMS, а в таблице MOVES — ссылка на уникальный идентификатор переданной/полученной книги (две последние связи называются связью *один к одному*). Таблица PAYMENTS имеет единственную связь — с таблицей FIRMS. Однако при появлении в ней очередной записи должны соответствующим образом изменяться суммы платежей в таблице NAKLS и сальдо в таблице FIRMS.

Ниже приводится описание пяти основных таблиц БД «Книголюб».

Таблица NAKLS

Имя поля	Назначение
NaklID	Уникальный идентификатор накладной. По этому полю нужно создать первичный ключ
NDate	Дата составления накладной. По этому полю нужно создать индекс для сортировки накладных по мере их поступления
NRetDate	Срок возврата нереализованных книг. По истечении этого срока возврат не принимается или возвращаемые книги уцениваются. Не используется в накладных обмена
NType	Тип накладной: 0 — покупка у поставщика; 1 — продажа покупателю; 2 — возврат поставщику; 3 — возврат от покупателя; 4 — книги получаются по обмену; 5 — книги передаются по обмену; 6 — покупка с предоплатой; 7 — продажа с предоплатой
NFirm	Уникальный идентификатор партнера (поле FirmID таблицы FIRMS)
NCoeff	Величина скидки/наценки. Переносится из поля FCoeff таблицы FIRMS, так как это поле со временем может меняться
NSum	Сумма накладной с учетом значения поля NCoeff
NPayedSum	Оплаченная сумма. Не используется в накладных обмена
NRetSum	Сумма возврата. Не используется в накладных обмена

Таблица BOOKS

Имя поля	Назначение
BookId	Уникальный код книги (первичный ключ)
BName	Название книги (индексное поле)
BAuthor	Автор(ы)
BPublish	Издательство
BYear	Год выпуска
BPages	Количество страниц
BISBN	Код ISBN
BStand	Стандарт упаковки (количество книг в пачке)
BQuan	Остаток книг на складе
BPrice	Цена покупки книги
BOpt	Цена оптовой продажи
BRozn	Цена розничной продажи

Таблица FIRMS

Имя поля	Назначение
FirmId	Уникальный идентификатор партнера (первичный ключ)
FName	Наименование партнера (индексное поле)
FAddress	Адрес
FCity	Город
FPhone	Телефон(ы)
FEMail	Адрес электронной почты
FPerson	Контактное лицо (лица)
FFinDelta	Финансовое сальдо
FCngDelta	Обменное сальдо
FCoeff	Коэффициент скидки/наценки
FRetDays	Количество дней для возврата

Таблица MOVES

Имя поля	Назначение
MoveId	Уникальный идентификатор (первичный ключ)
NaklID	Код накладной из поля NaklId таблицы NAKLS (индексное поле)
MBook	Код книги из поля BookId таблицы BOOKS (индексное поле)
MQuan	Количество экземпляров книги
MPrice	Цена одного экземпляра с учетом скидки/наценки

Таблица PAYMENTS

Имя поля	Назначение
PayID	Уникальный идентификатор платежного документа (первичный ключ)
PFirm	Код партнера из поля FirmId таблицы FIRMS
POut	Направление платежа: True — партнеру; False — от партнера
PDate	Дата платежа
PSum	Сумма платежа

В таблице NAKLS в поле NType содержится код, указывающий тип накладной. Чтобы работа с таблицей стала более удобной, введем дополнительную таблицу TYPES, в которой будем хранить расшифровку типа накладной.

Таблица TYPES

Имя поля	Назначение
TypeID	Код накладной (0..7)
TName	Расшифровка кода

Имена таблиц и полей

В файл-серверных БД имя таблицы совпадает с именем файла, в котором размещаются все содержащиеся в ней данные. Поскольку 32-разрядные версии Windows разрешают длинные русскоязычные имена файлов, было бы заманчиво назвать таблицы КНИГИ, НАКЛАДНЫЕ и т. п. Я не советую вам делать этого по той простой причине, что кириллицу нельзя использовать в SQL (точнее, это можно делать, только заключив полное название таблицы в кавычки). А так как даже в файл-серверных СУБД часто приходится применять SQL-запросы, это может затруднить их формирование. По той же причине не следует использовать кириллицу в именах полей.

ПРИМЕЧАНИЕ

Поскольку .NET Framework работает с символами Unicode, в именах таблиц и полей можно использовать символы национального алфавита.

В именах полей полезно ставить префикс из одной-двух букв названия таблицы (в таблице NAKLS все имена начинать с буквы «N», в FIRMS — с «F» и т. п.). Это исключит вероятность того, что вы случайно назовете поле одним из зарезервированных в SQL слов и просто не сможете составить нужный запрос.

ПРИМЕЧАНИЕ

Исключение составляет поле NaklID в таблице MOOVES. Дело в том, что по полям NaklID между таблицами NAKLS и MOVES будет создана реляционная связь «один ко многим». В технологии ADO.NET имена такого рода полей и тип хранящихся в них данных должны быть одинаковыми.

Механизмы BDE и ODBC

Характерной особенностью программ, созданных с помощью Delphi и предназначенных для работы с файл-серверными базами данных, является их зависимость от специальной библиотеки программ, которая называется BDE (Borland Database Engine — машина баз данных корпорации Borland). BDE представляет собой набор библиотек DLL, предназначенных для низкоуровнего доступа к данным самых разных форматов. BDE автоматически устанавливается в процессе установки Delphi и регистрируется в реестре 32-разрядной версии Windows. BDE «умеет» работать с таблицами самых распространенных СУБД, причем как файл-серверных (dBase, Paradox, FoxPro, Clipper), так и клиент-серверных (InterBase, Microsoft SQL Server, Oracle и др.). В BDE имеется собственный интерпретатор языка SQL, что позволяет создавать запросы не только к серверам БД, но и к таблицам файл-сервера.

Без установки и регистрации BDE на компьютере не может работать ни одна программа БД, созданная в Delphi и использующая механизм BDE. Это обстоятельство существенно затрудняет распространение программ, так как вместе с программой должен поставляться и набор библиотек BDE.

Альтернативой BDE может служить система драйверов ODBC (Open Data Base Connection — открытое соединение с БД). Драйверы ODBC выполняют сходные с BDE функции низкоуровнего доступа к данным и должны распространяться вместе с программой.

В Delphi 2005 имеются технологии, позволяющие обойтись без BDE или ODBC: ADO.NET (см. главу 2), dbDo.NET (см. главу 4), dbExpress.NET (см. главу 5) и IBX.NET (см. главу 6).

Создание таблиц файл-серверных БД

Как уже говорилось, в файл-серверных БД (а описываемый далее пример создан на основе файл-серверной БД) все таблицы размещаются в одном каталоге (папке). Поэтому перед созданием БД создадим ее каталог. Пусть это будет каталог C:\BIBLDATA. Создание таблиц файл-серверных БД осуществляется с помощью утилиты (вспомогательной программы) Database Desktop (DBD), входящей в комплект поставки Delphi.

Утилита DBD решает целый ряд задач, связанных с таблицами файл-серверных БД. С ее помощью можно создавать или изменять структуру таблицы, строить ее первичные ключи и индексы, создавать и изменять записи, просматривать их и т. д. Запустите DBD с помощью команды Tools ▸ Database Desktop главного меню. Первое, что необходимо сделать, — это настроить рабочий каталог утилиты. Выберите команду File ▸ Working Directory и установите в появившемся окне ссылку на каталог C:\BIBLDATA (рис. 1.3).

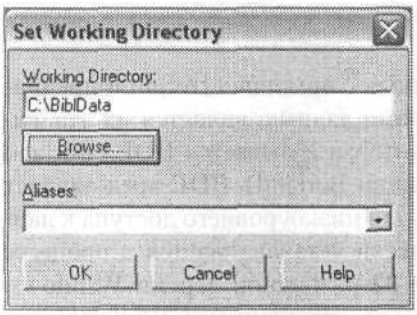


Рис. 1.3. Установка рабочего каталога

Для создания таблицы NAKLS выберите команду File ▸ New ▸ Table. DBD откроет окно Create Table, в котором можно выбрать тип таблицы. Тип таблицы определяет многие ее свойства. Тип Paradox 7 можно считать наилучшим для файл-серверных таблиц: щелчком на кнопке OK согласитесь с вариантом Paradox 7, предложенным по умолчанию. На экране появится окно (рис. 1.4), предназначенное для создания/редактирования структуры таблицы.

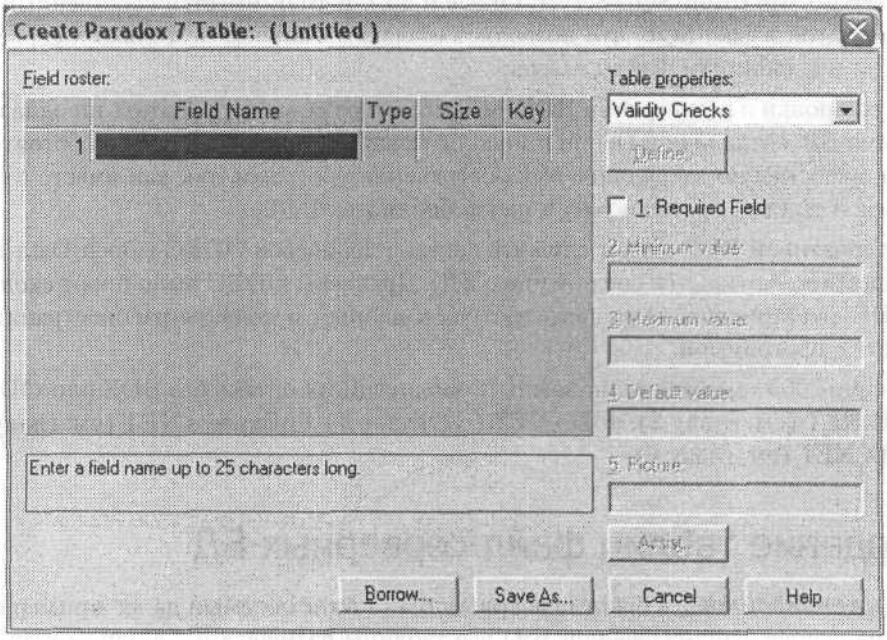


Рис. 1.4. Окно создания структуры таблицы

Каждому полю создаваемой таблицы соответствует одна запись в таблице Field roster этого окна: в колонку Field Name нужно поместить имя поля, в колонку Type — символ, определяющий тип хранимых в поле данных, в колонку Size — число, определяющее длину поля (требуется не для всех типов полей), и, наконец, в колонку Key — символ звездочки (*), если по значениям этого поля нужно построить первичный ключ.

Введите название первого поля, NaklId (первый символ названия поля для таблиц Paradox DBD всегда вводится прописным), и нажмите клавишу табуляции для перехода к следующей колонке, в которой вводится тип поля. Нажмите клавишу пробела, чтобы утилита DBD показала список возможных типов, и выберите

в нем тип Long[1]. Щелкните на колонке Key. Нажмите клавишу пробела, чтобы создать по полю первичный ключ.

Продолжите ввод полей таблицы NAKLS так, как показано на рис. 1.5. Для первых четырех полей установите флажок Required Field, означающий, что при вводе очередной записи в эти поля обязательно должны быть помещены значения, — за этим будет следить СУБД. Четыре других поля могут не определяться в момент ввода очередной записи. Имеет смысл определить для них значения по умолчанию в строке Default value: для поля NCoeff таким значением будет 1, для остальных — 0.

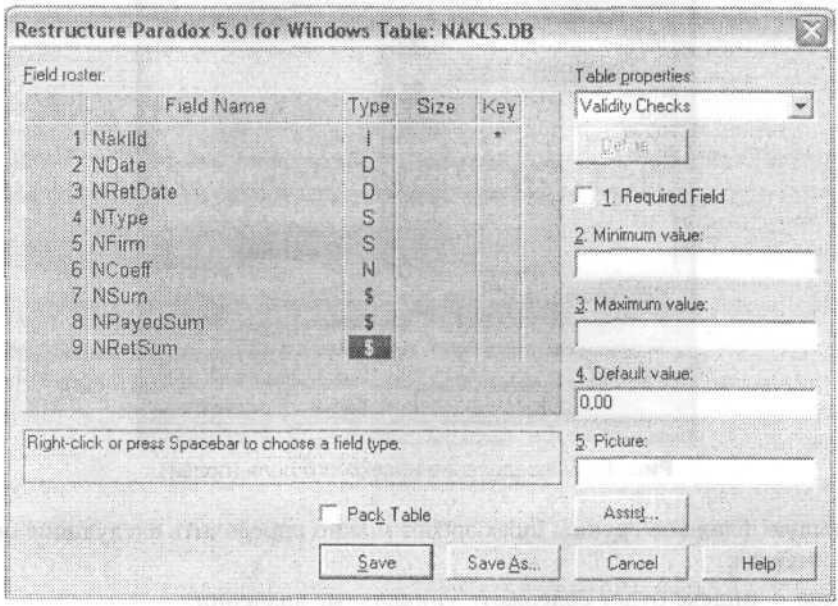


Рис. 1.5. Структура полей таблицы NAKLS

По полю NDate нужно определить индекс (в терминологии таблиц Paradox он называется *вторичным*). Как уже говорилось, в отличие от первичных ключей, индексы могут содержать неуникальные значения и служат для быстрого поиска нужной записи (группы записей). С помощью этого индекса мы сможем быстро

[1] Поле NaklID, как и все другие поля, предназначенные для создания первичных ключей, должно иметь только уникальные значения. Уникальность значений достигается тем, что такие поля обычно объявляются автоинкрементными. Автоинкрементные поля служат для создания уникального числа, однозначно определяющего запись: для первой записи в это поле будет автоматически помещено число 1, для второй — 2 и т. д. Вставка уникального числа осуществляется автоматически (BDE или ODBC) при добавлении очередной записи. При удалении какой-либо записи выделенное для нее число не используется вновь. В DBD есть тип Autoincrement, но этот тип неизвестен CLR и не может использоваться для установления реляционных связей в технологии ADO.NET. Вот почему тип поля NaklID — Long Inter (соответствует типу Int32 общей системы типов).

отыскать группу накладных, выданных такого-то числа. Для определения индекса раскройте список Table properties в правом верхнем углу окна, выберите пункт Secondary Indexes и щелкните на появившейся кнопке Define. В окне Define Secondary Index (рис. 1.6) в списке полей таблицы выделите (щелчком) поле NData и затем щелкните на кнопке со стрелкой вправо, чтобы перенести поле в список Indexed fields. Замечу, что таким способом можно перенести не одно, а несколько полей. Индекс, построенный по нескольким полям, называется *составным*.

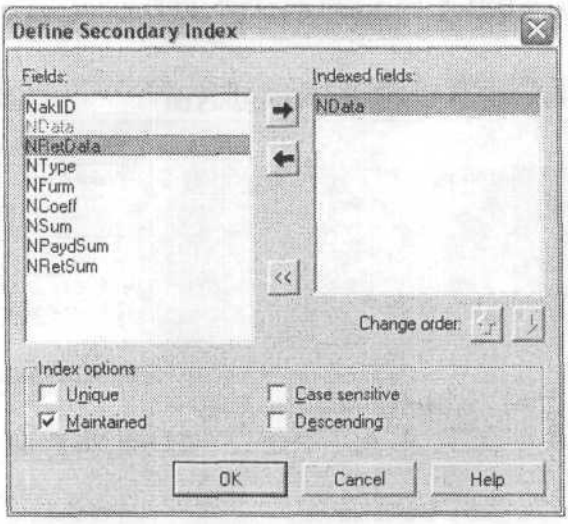


Рис. 1.6. Определение индексного поля (полей)

С помощью флажков группы Index options можно определить следующие особенности индекса:

- Unique — индекс будет содержать уникальные значения;
- Maintained — индексные поля сортируются по возрастанию значений;
- Case sensitive — индекс чувствителен к регистру букв в текстовых полях;
- Descending — индексные поля сортируются по убыванию значений.

В нашем случае (а также при определении индексов в других таблицах демонстрационной БД) оставьте эти флажки без изменений (установлен только флажок Maintained) и щелкните на кнопке OK. DBD запросит имя индекса (в таблицах Paradox, как и в большинстве серверов БД, индексы именуются) — введите имя Nakls_ date и щелкните на кнопке OK. Во вновь появившемся окне определения структуры таблицы щелкните на кнопке Save as и затем укажите имя файла — Nakls.

Мы только что создали *структуру* таблицы NAKLS. Для ее *наполнения* лучше воспользоваться ее копией, имеющейся на сайте издательства: www.piter.com. Руководствуясь описанием полей, вы можете самостоятельно создать структуры остальных таблиц демонстрационной БД, а вот реальные таблицы лучше взять на указанном сайте.

Свойства таблиц Paradox 7

Как уже говорилось, тип таблиц Paradox 7 предпочтителен при создании файл-серверных БД. Таблицы Paradox 7 по сравнению с другими поддерживают самый богатый набор разных типов полей, что позволяет автоматически следить за правильностью вводимых в поля данных (пункт Validity Checks в списке Table properties), выбирать данные из другой таблицы (Lookup Table), строить вторичные индексы, в том числе составные (Secondary Indexes), следить за ссылочной целостностью БД (Referential Integrity), защищать таблицу от несанкционированного доступа (Password Security), выбирать языковой драйвер (Table Language). Поскольку вам, возможно, понадобится создавать таблицы Paradox для реальной файл-серверной БД, в этом разделе рассматриваются их свойства.

Типы полей

В табл. 1.1 представлены типы полей, которые могут использоваться в таблицах Paradox. В этой таблице в колонках «Type» и «Size» указаны, соответственно, условный символ типа и его размер, то есть те данные, которые помещаются в одноименные колонки таблицы `Field roster` DBD.

Таблица 1.1. Типы данных в таблицах Paradox

Type	Size	Тип	Описание
A	1–255	Alpha	Текстовое поле указанной длины
N		Number	Числа с плавающей запятой в диапазоне от -10^{307} до $+10^{307}$ с 15 значащими десятичными разрядами
$		Money	Денежное поле. Содержит вещественные числа с фиксированной запятой, шестью знаками целой части и двумя знаками дробной
S		Short	Целые числа в диапазоне от –32 768 до +32 767
I		Long Integer	Целые числа в диапазоне от –2 147 483 648 до +2 147 483 647
#	0–32	BCD	Двоично-десятичные вещественные числа. `Size` — количество разрядов после запятой
D		Date	Дата в диапазоне от 1.01.0000 до 31.12.9999
T		Time	Время с точностью до миллисекунд
@		Timestamp	Дата и время
M	1–240	Memo	Мемо-поле для размещения произвольных текстовых строк неограниченной длины. Первые `Size` символов хранятся в основной таблице, остальные — в файле с расширением .MB
F	0–240	Formatted Memo	Мемо-поле для размещения форматированного текста в формате RTF

продолжение

Таблица 1.1 (продолжение)

Type	Size	Тип	Описание
G	0–240	Graphic	Графическое изображение в формате BMP. Size байтов этого поля хранится в основной таблице, остальные — в отдельном файле
O	0–240	OLE	Объект OLE
L		Logical	Логическое поле. Содержит значение True или False
+		Autoincrement	Автоинкрементное поле
B	0–240	Binary	Набор байтов произвольной длины. Первые Size байтов хранятся в основной таблице, остальные — в отдельном файле
Y	1–255	Bytes	Набор из Size байтов (целиком хранится в таблице)

Контроль содержимого полей

По умолчанию сразу после открытия окна редактирования структуры таблицы в списке Table properties выбран пункт Validity Checks (см. рис. 1.5), что позволяет контролировать содержимое полей.

С помощью флажка Required Field вы можете потребовать обязательного заполнения поля при вводе новой записи — за этим будет следить BDE. Также на BDE можно возложить контроль за минимальным и максимальным значениями числового поля (строки Minimum Value и Maximum Value). В строке Default Value можно указать значение поля по умолчанию — при вводе новой записи значение в это поле поместит BDE. С помощью строки Picture можно задать шаблон для автоматического форматирования значения поля. Например, если задан шаблон (###)###-#### и в поле введена строка 9151653939, она будет автоматически преобразована к виду (915)165-3939.

За более полной информацией о шаблонах обратитесь к встроенной справочной службе DBD.

Таблица подстановки

Для какого-либо поля таблицы иногда требуется установить однозначную связь с полем другой таблицы. В этом случае BDE будет следить за тем, чтобы значение вновь вводимой записи в поле первой таблицы было бы одним из значений указанного поля в другой таблице, которая в этом случае называется *таблицей подстановки*. Например, в нашей демонстрационной БД поле NaklID таблицы MOVES должно содержать ссылку на накладную в таблице NAKLS. Если с помощью DBD установить связь между этим полем и полем NaklID таблицы NAKLS, при вводе или редактировании таблицы MOVES BDE отвергнет любые значения, которые не совпадают с одним из значений поля NaklID таблицы подстановки NAKLS.

Для установления связи нужно выбрать пункт Table Lookup в списке Table properties и щелкнуть на кнопке Define. В появившемся окне (рис. 1.7) в списке Fields выбирается поле, за значениями которого нужно следить, и щелчком на кнопке со стрелкой вправо имя этого поля переносится в строку Field name. Затем в списке Lookup table выбирается нужная таблица и щелчком на кнопке со стрелкой влево имя первичного ключевого поля этой таблицы переносится в строку Lookup field.

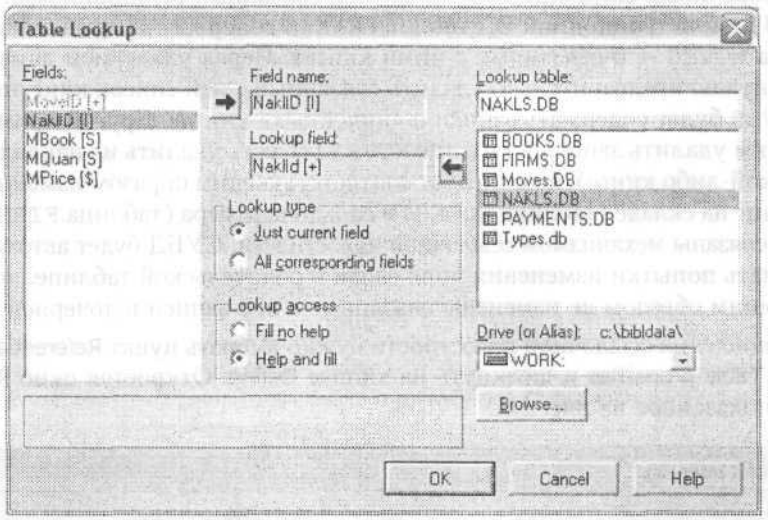


Рис. 1.7. Установление связи с таблицей подстановки

ПРИМЕЧАНИЕ

Связь будет установлена только в том случае, если поле, за значением которого нужно следить, имеет такой же тип, как и первое индексное поле в таблице подстановки. Так как в таблицах Paradox вторичный индекс можно создать только при наличии в таблице первичного ключа, который, в свою очередь, можно создать только в первом поле, подстановочное поле должно быть ключевым и первым в списке полей таблицы подстановки.

Назначение переключателей:

- Just current field — проверяется соответствие только связанных полей;
- All corresponding fields — проверяется соответствие всех полей обеих таблиц (в этом случае структуры обеих таблиц должны быть идентичными);
- Fill no help — при редактировании проверяемого поля таблица подстановки не показывается;
- Help and fill — при редактировании проверяемого поля показывается таблица подстановки.

Вторичные индексы

Процесс создания вторичных индексов описан в разделе «Создание таблиц файл-серверных БД». В таблицах Paradox вторичный индекс может быть создан только

в том случае, если предварительно в ней создан первичный ключ. В свою очередь, первичный ключ может быть создан только для одного или нескольких первых полей.

Ссылочная целостность

Ссылочная целостность — это особый механизм, способствующий поддержанию непротиворечивых сведений в таблицах БД, связанных реляционными отношениями. В демонстрационной БД таблица NAKLS содержит данные о накладных, а таблица MOVES — о связанных с ними книгах. Перед удалением данных о накладной нужно предварительно удалить связанный с ней список книг, иначе таблица MOVES будет содержать записи о «ничейных» книгах. Другой пример. Если мы захотим удалить запись в таблице MOVES (то есть удалить из накладной данные о какой-либо книге), нам следует соответствующим образом изменить количество книг на складе (таблица BOOKS) и сальдо партнера (таблица FIRMS). Если таблицы связаны механизмом ссылочной целостности, СУБД будет автоматически блокировать попытки изменения поля связи в родительской таблице, пока соответствующим образом не изменены связанные с ней записи в дочерней таблице.

Для установления ссылочной целостности нужно выбрать пункт Referential Integrity в списке Table properties и щелкнуть на кнопке Define. Откроется окно Referential Integrity, показанное на рис. 1.8.

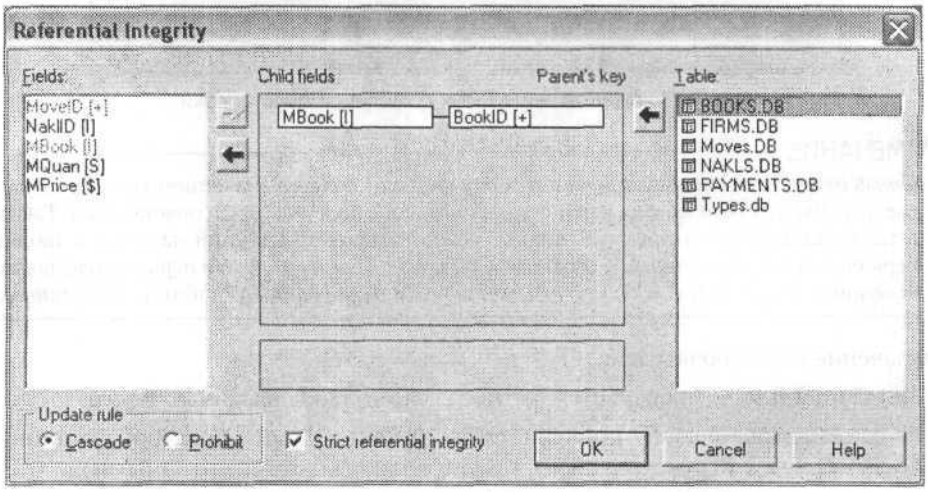


Рис. 1.8. Установление ссылочной целостности

В списке Fields выбирается поле связи редактируемой таблицы (она должна быть дочерней), а в списке Table — нужная родительская таблица. После щелчков на кнопках со стрелками имена полей связи переносятся в строки Child fields и Parent's key. Далее нужно щелкнуть на кнопке OK. DBD запросит имя вновь создаваемой ссылочной целостности и затем создаст ее. После этого попытка удаления информации о любой накладной будет отвергаться до тех пор, пока не удалены данные обо всех связанных с ней книгах.

С помощью переключателей в группе Update rule определяются правила поддержания ссылочной целостности:

- Prohibit — BDE отвергает любые изменения в связанном поле родительской таблицы без соответствующего изменения записей в дочерней таблице; удаление записи родительской таблицы блокируется до удаления связанных с ней записей в дочерней таблице;
- Cascade — BDE автоматически производит каскадные изменения в дочерней таблице при изменениях родительской таблицы[1].

Парольная защита

Любая таблица Paradox может быть полностью или частично защищена от несанкционированного доступа. Для этого в списке Table properties выбирается пункт Password Security и щелчком на кнопке Define открывается соответствующее диалоговое окно (рис. 1.9).

Пароль может содержать от 1 до 15 любых символов, в том числе пробелов. Он чувствителен к регистру букв. С помощью кнопки Auxiliary Passwords вызывается дополнительное окно, в котором можно уточнить, какие поля и как защищаются.

Рис. 1.9. Окно определения пароля

Выбор языкового драйвера

Если таблица имеет текстовые поля, для нее нужно указать языковой драйвер. Для этого в списке Table properties выбирается пункт Table Language и щелчком на кнопке Define открывается диалоговое окно Table Language (рис. 1.10). Для правильного отображения русскоязычного текста следует выбирать драйвер Pdox ANSI Cyrillic или Paradox Cyrr 866. Первый использует кодовую страницу 1251 для Windows, второй — страницу 866 для MS-DOS.

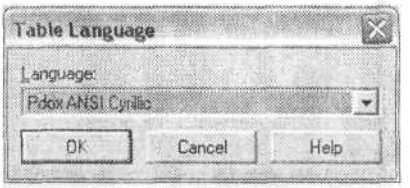


Рис. 1.10. Окно выбора языкового драйвера

ВНИМАНИЕ

Устанавливать языковой драйвер следует даже для таблиц, которые содержат одни числовые поля. Дело в том, что в программе вы будете показывать не условные числовые обозначения партнера или книги, а с помощью подстановочных (lookup) полей дополните таблицу полями из других таблиц. Если для таблицы не установлен нужный языковой драйвер, содержимое подстановочных полей окажется искаженным.

[1] Мне не удалось добиться каскадных изменений — на практике BDE работает только в режиме Prohibit.

Создание таблиц клиент-серверных БД

Таблицы и все относящееся к ним (индексы, триггеры, генераторы и т. д.) в серверных БД хранятся в одном файле — файле базы данных. Поэтому перед созданием таблиц необходимо создать этот файл. Каждый промышленный сервер поставляется с утилитами, позволяющими работать с ним и не требующими для этого каких-либо дополнительных средств. Вместе с сервером InterBase, входящим в поставку Delphi, поставляется утилита IBConsole. Разыщите одноименную команду ее запуска в программной группе InterBase (кнопка Пуск, далее Все программы и т. д.) и запустите утилиту (рис. 1.11).

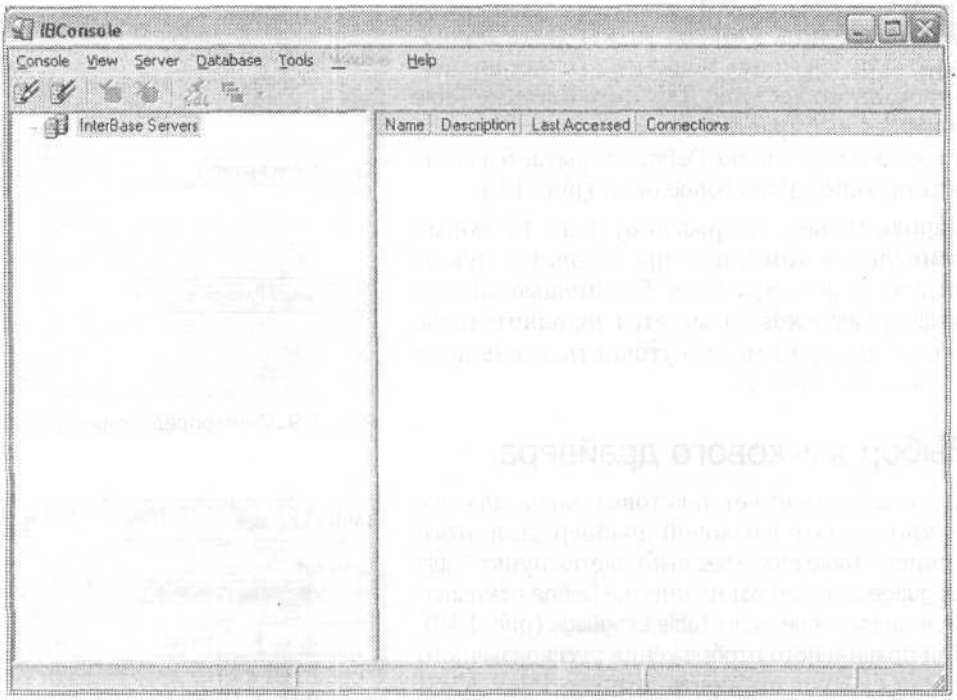


Рис. 1.11. Окно утилиты IBConsole

Сначала нужно зарегистрировать сервер (в сети могут работать несколько серверов). Выберите в меню команду Server ▸ Register, открыв окно регистрации сервера, показанное на рис. 1.12.

ПРИМЕЧАНИЕ

Расширение .GDB является стандартным для файлов БД сервера InterBase. В списке строки Default Character Set нужно выбрать кодировку текстовых полей. Если вы собираетесь работать с технологией ADO.NET, следует оставить в этой строке неопределенное значение (пункт None). Для всех других технологий, а также при создании приложений Win32 можно установить кодировку WIN1251. Однако опыт показывает, что с неопределенной кодировкой текстовые поля нормально работают и в этом случае.

Установите переключатель Local Server. Если вы уже работали с утилитой IBConsole и регистрировали локальный сервер, этот переключатель окажется недоступным, а при открытии окна утилиты в нем вы увидите перечеркнутый значок локального сервера. В этом случае щелкните на значке и в правом окне выберите Un-Register.

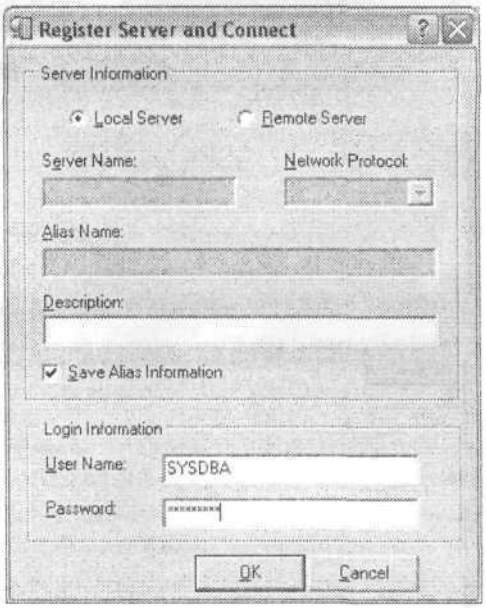


Рис. 1.12. Регистрация сервера

Введите в строке User Name имя SYSDBA, а в строке Password — пароль masterkey.

После регистрации сервера нужно создать файл БД. Выберите в меню команду Database ▸ Create Database, открыв окно создания файла БД (рис. 1.13).

В таблице File(s) введите маршрут доступа и имя файла, например C:\BIBLDATA\IB_BIBL.GDB, а в строке Alias — псевдоним БД, например BIBL, после чего закройте окно щелчком на кнопке OK.

Создание таблиц и всех остальных сущностей серверной БД осуществляется с помощью команд на языке SQL, точнее, на его подмножестве — языке DDL (Data Definition Language — язык определения данных). Выберите в меню Tools ▸ Interactive SQL, открыв окно Interactive SQL (рис. 1.14).

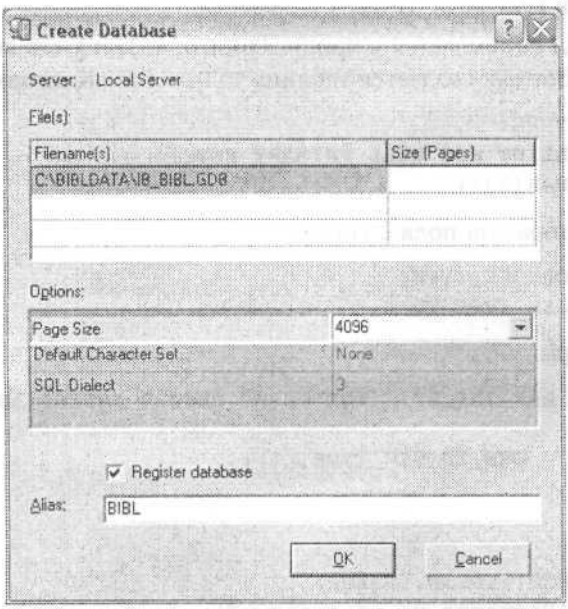


Рис. 1.13. Создание файла БД

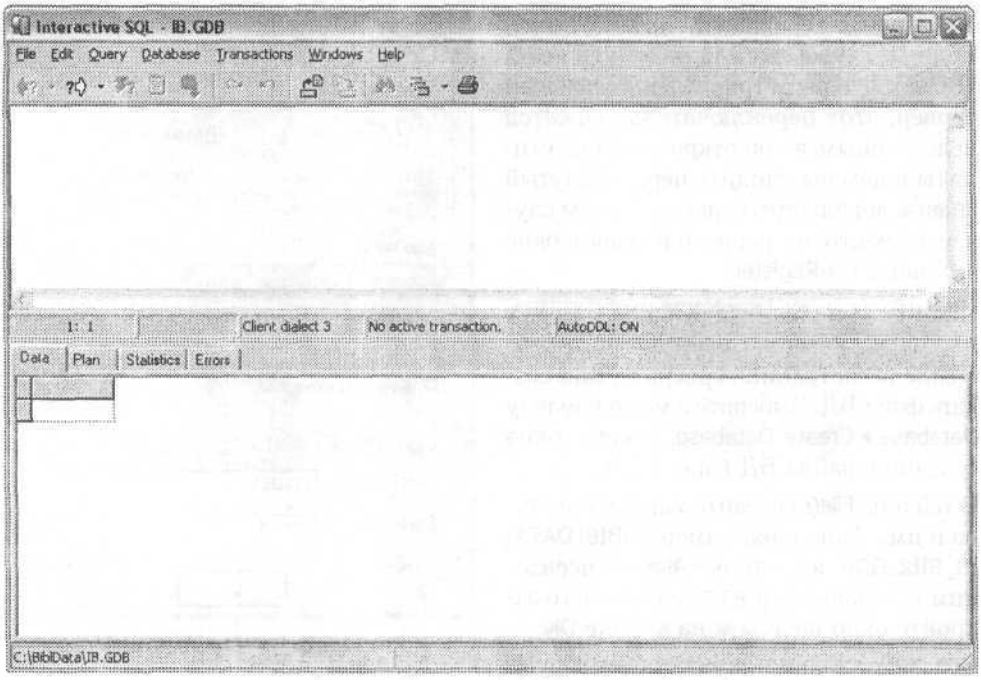


Рис. 1.14. Окно Interactive SQL

В этом окне верхняя панель предназначена для ввода команд или запросов, в нижней отображается результат запроса.

Язык SQL (DDL) описывается в приложении Б, поэтому его конструкции здесь не поясняются. Вот как создается таблица TYPES БД «Книголюб»:

```
CREATE TABLE TYPES(
    TYPEID SMALLINT NOT NULL PRIMARY KEY,
    TNAME VARCHAR(23))
```

Создание генератора для поля TYPEID:

```
CREATE GENERATOR GEN_TYPE;
SET GENERATOR GEN_TYPE TO 0
```

Создание триггера:

```
CREATE TRIGGER BEF_INS_TYPE FOR TYPES BEFORE INSERT AS
BEGIN
    NEW.TYPEID = GEN_ID(GEN_TYPE, 1);
END
```

Ввод записи:

```
INSERT INTO TYPES(TYPEID, TNAME)
    VALUES(0, "Поставка на реализацию")
```

ВНИМАНИЕ

Ввод записи с русскоязычными полями в БД, для которых в строке Default Character Set окна Create Database указано значение None, средствами утилиты IBConsole невозможен. Используйте для этого утилиту SQL Explorer.

Пример простой программы

В этом разделе на примере создания несложной программы я постараюсь пояснить основные приемы программирования для БД. Окно работающей программы показано на рис. 1.15.

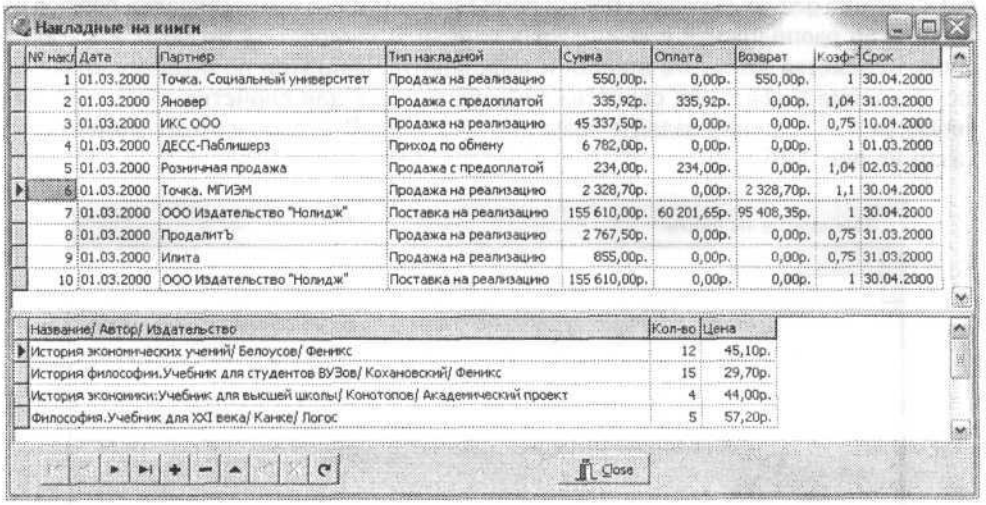


Рис. 1.15. Окно программы на этапе прогона

Как видно из рисунка, программа позволяет просматривать накладные и связанные с ними списки книг. Напомню, что все необходимые для программы файлы БД вы найдете на сайте издательства «Питер» (http://www.piter.com).

Разработка главной формы

Начните новый проект VCL Forms и измените следующие свойства пустой формы, заданные по умолчанию:

```
Caption = 'Накладные на книги'
Name = 'fmNakls'
```

После изменения свойств сразу сохраните модуль в специально отведенной для этого папке под именем `fmNaklsU`, а проект — под именем `Nakls`.

Поместите на форму две панели `TPanel`, задайте для их свойств `Align` значения `alBottom`: нижняя из них предназначена для размещения кнопок навигатора и кнопки закрытия программы, а расположенная над ней — для размещения сетки

DBGrid, в которой будет отображаться список книг. Чтобы пользователь программы мог менять высоту этой панели, поместите на пустую часть формы вешку разбивки TSplitter (категория Additional палитры компонентов), для ее свойства Align задайте значение alBottom, для свойства Beveled — значение True и для свойства Height — значение 5. Поместите на форму еще одну панель и задайте для ее свойства Align значение alClient — на этой панели будет расположена сетка DBGrid с данными о накладных.

Поместите на верхнюю и среднюю панели по компоненту TDBGrid (вкладка Data Controls) и для их свойства Align задайте значение alClient.

Очистите свойство Caption самой нижней панели и поместите на нее навигатор БД TDBNavigator (вкладка Data Controls) и кнопку TBitBtn (вкладка Additional). Навигатор расположите у левого края панели, а кнопку — у правого. Раскройте список свойства Anchors кнопки и выровняйте навигатор по правому краю: задав значение False для свойства akLeft и True для свойства akRight. Для свойства Kind кнопки задайте значение bkClose. Вид окна к этому моменту показан на рис. 1.16.

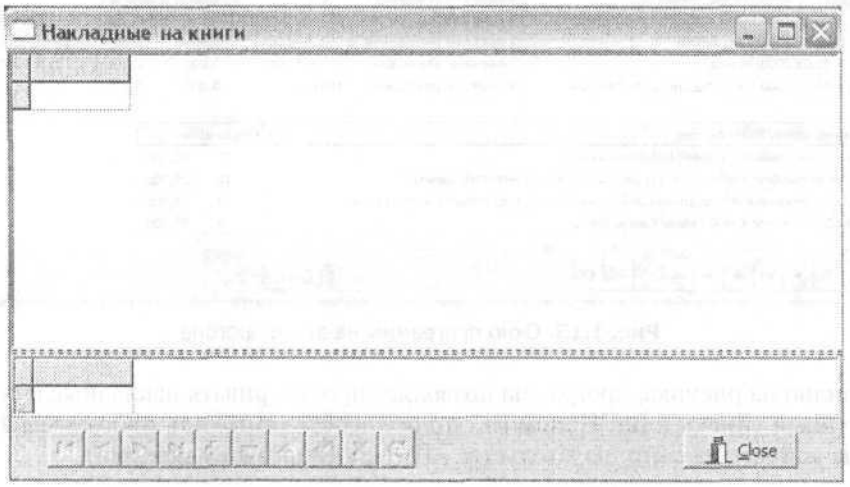


Рис. 1.16. Главное окно программы на этапе разработки

Сохраните все сделанные вами изменения модуля главной формы на диске.

Создание псевдонима БД

Перед тем как двигаться дальше, сделаем очень важный шаг: создадим *псевдоним* для нашей учебной БД. Псевдоним БД — это просто имя БД. Для файл-серверных БД псевдоним определяет путь доступа к файлам базы данных. В дальнейшем мы, возможно, захотим изменить его. В этом случае нам не придется исправлять этот путь в многочисленных компонентах доступа к данным — достаточно изменить его в псевдониме, и все ссылающиеся на псевдоним компоненты будут связаны с новым местом размещения данных. Роль псевдонима особенно велика

в клиент-серверных БД, в которых он содержит многочисленные дополнительные свойства, управляющие доступом к серверу.

Создать псевдоним можно либо с помощью Администратора BDE (Пуск ▸ Панель управления ▸ BDE Administrator), либо с помощью очень полезной утилиты Database Explorer (SQL Explorer). Так как в дальнейшем нам придется много раз прибегать к ее услугам, настроим пункт Tools главного меню Delphi. Щелкните на этом пункте и выберите команду Configure Tools. В открывшемся окне щелкните на кнопке Add, а во вновь появившемся — на кнопке Browse. С помощью диалогового окна разыщите в папке BIN каталога размещения Delphi файл dbexplor.exe утилиты SQL Explorer (рис. 1.17).

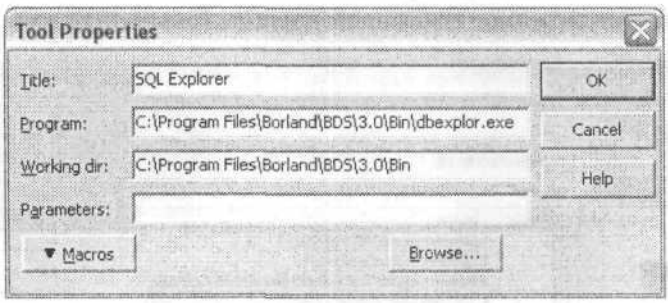


Рис. 1.17. Настройка вызова утилиты SQL Explorer из меню Tools Delphi

Запустите утилиту SQL Explorer, на вкладке Databases открывшегося окна щелкните правой кнопкой мыши на узле Database и выберите команду New в контекстном меню. Утилита предложит выбрать тип вновь создаваемого псевдонима. Согласитесь с вариантом Standard, предлагаемым по умолчанию, — псевдонимы именно этого типа предназначены для обслуживания файл-серверных БД с таблицами Paradox. Сразу после этого появится имя псевдонима STANDARD1, предлагаемое по умолчанию, и на вкладке Definition будут перечислены его свойства (рис. 1.18).

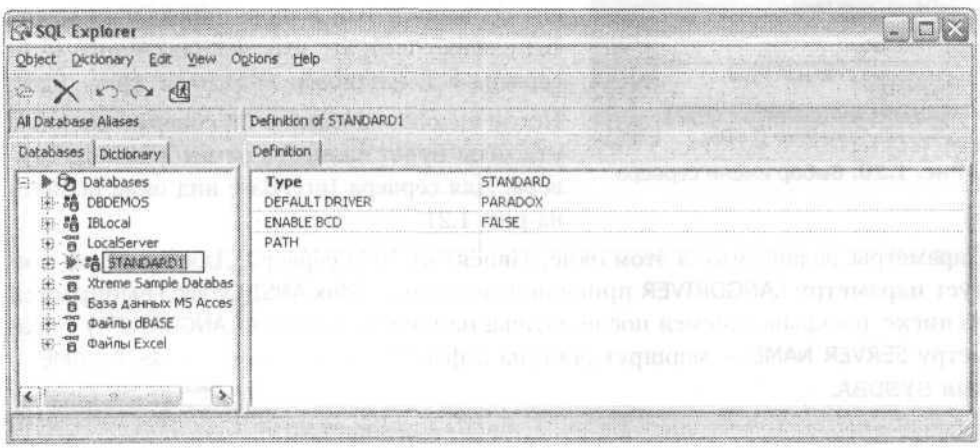


Рис. 1.18. Определение псевдонима

Воспользуемся тем, что выделено имя STANDARD1, чтобы сразу заменить его именем BIBLDATA. После этого перейдите на вкладку Definition и в пустом поле справа от свойства PATH введите маршрут доступа к файлам БД: C:\BIBLDATA. На вкладке Databases щелкните на вновь созданном псевдониме правой кнопкой мыши и выберите команду Apply в контекстном меню. В появившемся после этого диалоговом окне подтвердите необходимость запомнить вновь созданный псевдоним.

Если теперь на вкладке Databases щелкнуть на значке свернутого узла слева от имени псевдонима, а затем — на значке свернутого узла Tables, вы увидите все таблицы БД «Книголюб». Щелкнув на любой из них и открыв вкладку Data, вы сможете увидеть содержимое выбранной таблицы (рис. 1.19).

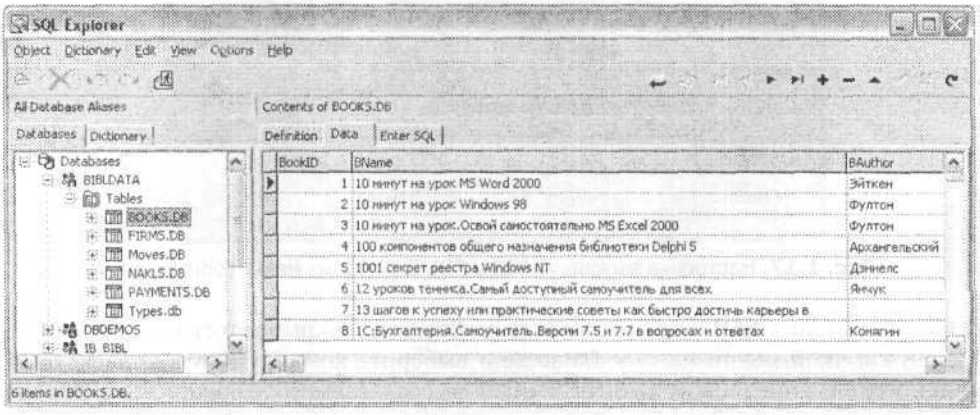


Рис. 1.19. Отображение содержимого таблицы в окне SQL Explorer

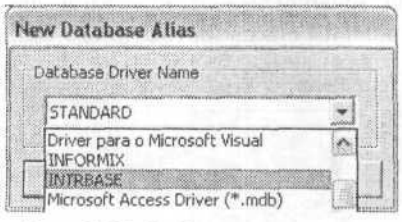


Рис. 1.20. Выбор имени сервера

Псевдоним BIBLDATA открывает доступ к файл-серверной БД. Чтобы получить доступ к клиент-серверной БД, в окне определения типа БД следует выбрать имя используемого вами сервера БД, например INTRBASE (рис. 1.20).

После щелчка на кнопке OK содержимое окна утилиты будет зависеть от выбранного сервера. Для сервера InterBase вид окна показан на рис. 1.21.

Параметры, задаваемые в этом окне, зависят от типа сервера. Для InterBase следует параметру LANGDRIVER присвоить значение[1] Pdox ANSI Cyrillic (выбирается в списке, раскрывающемся после щелчка на кнопке в строке LANGDRIVER), параметру SERVER NAME — маршрут доступа к файлу БД, а параметру USER NAME — имя SYSDBA.

[1] Как уже отмечалось, установка языкового драйвера не обязательна.

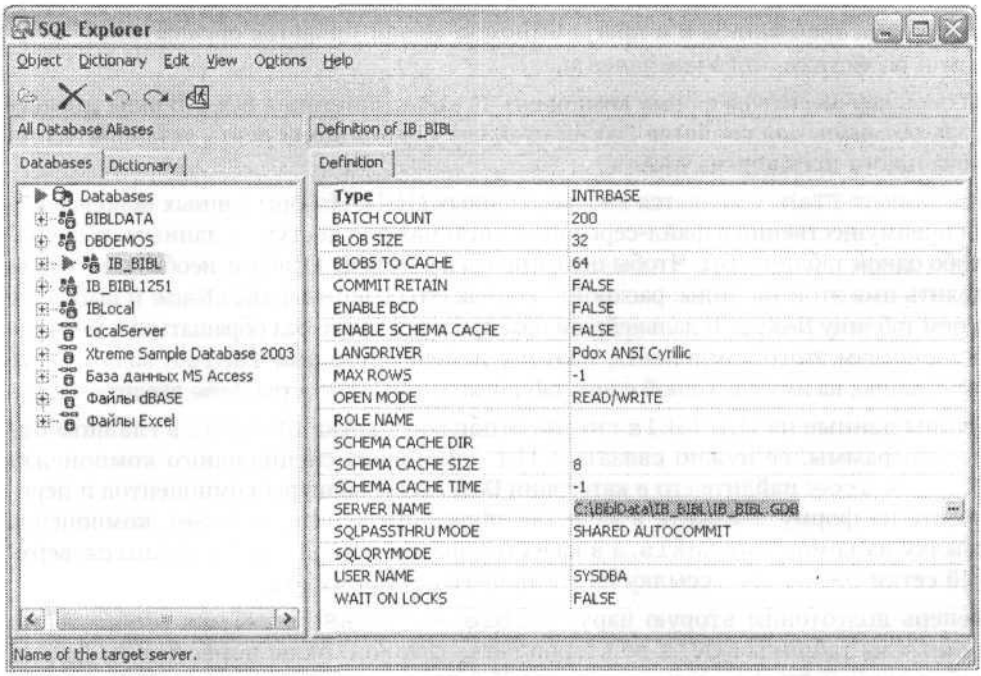


Рис. 1.21. Определение псевдонима БД для сервера InterBase

Связь с данными

Для связи таблиц с БД воспользуемся компонентом `TDatabase` категории BDE палитры компонентов Delphi: перенесите его на форму. В окне инспектора объектов раскройте список свойства `AliasName` и выберите псевдоним `BIBLDATA`. В строке свойства `DatabaseName` введите произвольное имя (например, `AAA`) так называемого *локального* псевдонима, который создает компонент `TDatabase`.

Локальный псевдоним доступен только в той программе, в которой используется компонент `TDatabase`. Этот компонент выполняет множество полезных функций, обеспечивающих связь программы с БД. Для файл-серверных систем только с его помощью можно реализовать *транзакции* — специальный механизм доступа к данным, повышающий их достоверность и непротиворечивость. В клиент-серверных системах он, кроме того, способен передать серверу БД имя пользователя, его пароль и ряд других параметров, оптимизирующих связь с сервером и избавляющих пользователя программы от обязательной регистрации на сервере.

После определения псевдонима БД и создания локального псевдонима исчезнет красный знак вопроса слева от компонента в окне структуры, что свидетельствует о готовности компонента к работе. Поскольку в нашей простой программе мы не будем непосредственно обращаться к компоненту, можно оставить для него имя `Database1`, заданное по умолчанию, но я все-таки рекомендую изменить его на `DB`: в дальнейшем на примере этой программы я продемонстрирую работу

механизма транзакций, и в программном коде нам придется обращаться к методам и свойствам этого компонента.

Теперь перенесите на форму компонент TTable (категория BDE). В окне инспектора объектов для свойства DatabaseName нового компонента установите имя локального псевдонима AAA.

Компонент TTable является набором данных (НД). Наборы данных используются преимущественно в файл-серверных системах для доступа к данным из какой-либо одной таблицы БД. Чтобы подготовить компонент к работе, необходимо определить имя этой таблицы: раскройте список его свойства TableName и выберите в нем таблицу NAKLS. В дальнейшем нам придется много раз обращаться к методам и свойствам этого компонента, поэтому измените его имя Table1, заданное по умолчанию, на имя связанной с ним таблицы: в строке свойства Name введите Nakls.

Чтобы данные из НД Nakls смогла отобразить сетка DBGrid1 в главном окне программы, ее нужно связать с НД с помощью специального компонента TDataSource: найдите его в категории Data Access палитры компонентов и перенесите на форму. Укажите в качестве значения свойства DataSet компонента ссылку на компонент Nakls, а в качестве значения свойства DataSource верхней сетки DBGrid1 — ссылку на компонент DataSource1.

Теперь подготовим вторую пару, TTable — TDataSource, для отображения данных из таблицы MOVES во второй сетке главного окна: поместите на форму набор данных TTable и источник данных TDataSource; свяжите НД с таблицей MOVES и дайте ему имя Moves. Свяжите источник данных с таблицей MOVES, а сетку DBGrid2 — с источником DataSource2. Вид формы к этому моменту показан на рис. 1.22.

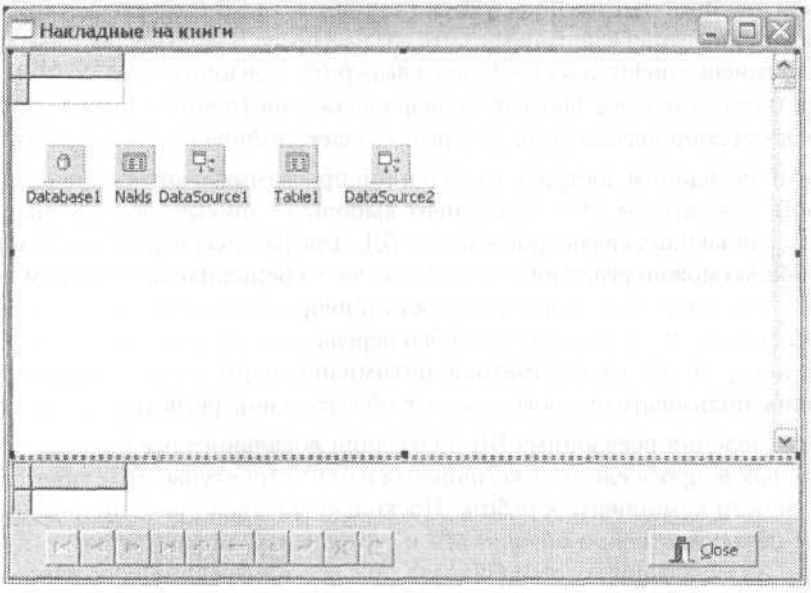


Рис. 1.22. Окно формы с двумя наборами данных

Наборы данных Nakls и Moves связаны реляционным отношением *один ко многим*: единственной записи (накладной) в первом НД может соответствовать произвольное количество записей (книг) во втором. Чтобы НД «знали» об этом и согласованно отображали данные, их нужно предварительно подготовить. Для этого в свойство MasterSource таблицы MOVES поместите ссылку на компонент DataSource1 и щелчком на кнопке раскройте окно редактора его свойства MasterFields (рис. 1.23).

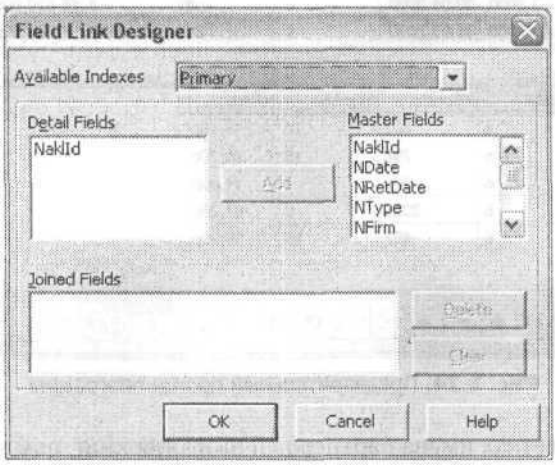


Рис. 1.23. Редактор свойства MasterFields

Раскройте список Available Indexes и выберите индекс Move_Nakl. Щелкните на единственном поле NaklID в списке Details Fields и на поле NaklID в списке Master Fields. После этого станет доступна кнопка Add — щелкните на ней. В области Joined Fields появится нужная связь NaklID → NaklID. Закройте окно редактора щелчком на кнопке OK. Раскройте список свойства IndexName таблицы MOVES и выберите индекс Move_Nakl.

Щелкните на таблице MOVES и откройте НД: в окне инспектора объектов поместите в его свойство Active значение True. Точно так же откройте таблицу NAKLS.

Свяжите навигатор баз данных DBNavigator1 в нижней части окна с источником DataSource1.

Создание объектов-полей

Если вы на этом этапе выполните прогон программы, то увидите (рис. 1.24), что сетки связаны друг с другом: при перемещении указателя текущей записи в верхней сетке автоматически меняется содержимое нижней сетки.

Однако сами данные использовать практически невозможно, так как вместо имен партнеров или наименований книг в соответствующих колонках сеток видны лишь их идентификаторы. Кроме того, вас, очевидно, не удовлетворили бы и заголовки колонок, в которых по умолчанию выводятся названия соответствующих полей.

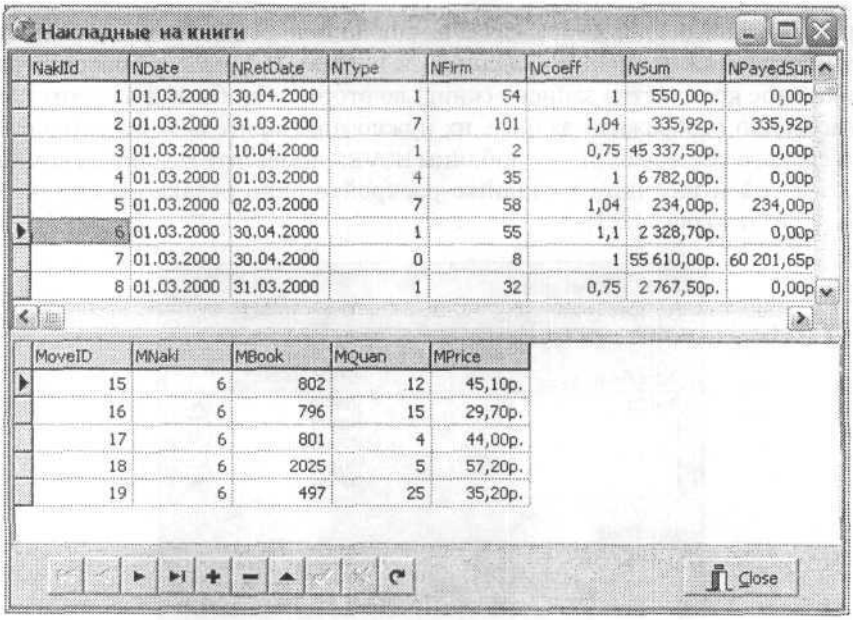


Рис. 1.24. Предварительный прогон программы

Чтобы показать в сетках имена партнеров и названия книг, нам нужно расширить исходные НД, добавив к ним подстановочные (*lookup*) поля из других таблиц. Для этого завершите прогон программы и вернитесь к редактору форм.

Двойным щелчком на компоненте `Nakls` откройте окно *редактора полей*, щелкните в окне редактора правой кнопкой мыши и в контекстном меню выберите команду Add all fields — окно редактора заполнится списком всех полей таблицы `NAKLS`. Этот список означает, что теперь для каждого поля НД `Nakls` создан специальный объект со своими свойствами, методами и событиями. Если вы щелкнете на любом поле в окне редактора полей, то в окне инспектора объектов станут доступными свойства и события *объекта-поля*. По умолчанию Delphi присваивает объекту-полю имя, полученное сцеплением имени НД и имени поля, так что теперь в программе окажутся компоненты с именами `NaklsNaklID`, `NaklsNFirm`, `NaklsNDate` и т. д.

Создание объектов-полей для каждого используемого в программе НД можно считать признаком хорошего стиля программирования, так как они упрощают доступ к данным и дают программисту дополнительные возможности. Например, с помощью свойства `DisplayLabel` вы можете изменить заголовки соответствующих колонок в сетках отображения данных, а с помощью обработчика события `OnGetText` — формат отображения данных и т. д. При этом следует помнить, что, если для НД создан хотя бы один объект-поле, соответствующий НД будет содержать значения только тех полей, для которых созданы объекты, и наоборот, если для НД не созданы объекты-поля, он содержит данные из всех полей соответствующей таблицы.

Напомню, что мы собираемся расширить НД за счет добавления к ним подстановочных полей из других таблиц. Без объектов-полей в этом случае нам просто не обойтись — это еще один веский довод в пользу их создания.

Создайте объекты-поля и для НД `Moves`.

Чтобы присоединить к наборам данных поля из других таблиц, нам нужно вначале создать для этих таблиц соответствующие НД: поместите на форму еще три компонента `TTable`. В свойстве `DatabaseName` каждого сошлитесь на локальный псевдоним `AAA`. Свяжите первый из них с таблицей `FIRMS` и дайте ему имя таблицы, второй НД — с таблицей `BOOKS` и дайте компоненту имя `Books`, третий — с таблицей `TYPES` и дайте компоненту имя `Types` (эта таблица содержит расшифровку кода типа накладной). Вновь двойным щелчком на компоненте `Nakls` откройте окно редактора полей и щелчком правой кнопкой мыши вызовите его контекстное меню. Выберите в меню команду **New Field**, чтобы открыть окно конструктора нового поля (рис. 1.25).

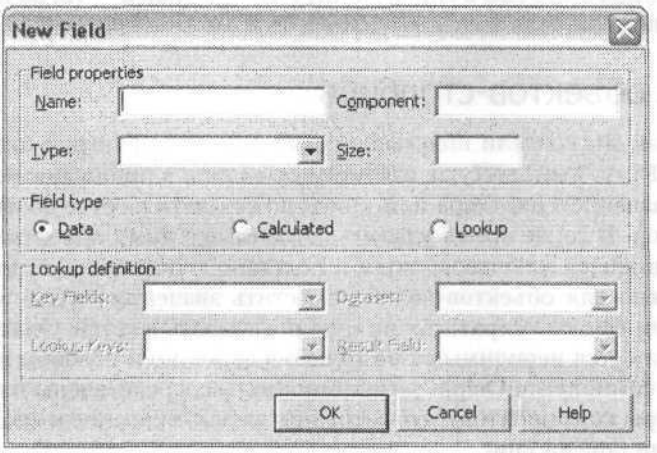


Рис. 1.25. Окно конструктора нового поля

С помощью *конструктора нового поля* можно создать в НД новые поля одного из трех типов (тип задается переключателями в группе **Field type**):

- **Data** — поля содержат произвольные данные;
- **Calculated** — поля содержат вычисляемые данные;
- **Lookup** — поля содержат данные из таблицы подстановки.

Поле первого типа будет отображаться в сетке пустой колонкой, которую можно заполнить в обработчике события `OnGetText` этого поля. Вычисляемое поле заполняется в обработчике события `OnCalcFields` набора данных. Для заполнения подстановочного поля данные берутся из нужного поля другого НД.

В строке `Name` окна конструктора поля введите имя нового поля `Firm`, раскройте список **Type** и выберите для поля тип **String**, его размер **Size** установите равным 40, установите переключатель **Lookup**, раскройте список полей **Key Fields** и выберите в нем поле `NFirm`, в списке **Dataset** выберите НД `Firms`, в списке **Lookup Fields** —

ключевое поле FirmID и, наконец, в списке Result Field — поле FName. Закройте окно конструктора щелчком на кнопке OK.

В БД есть небольшая таблица подстановки TYPES, содержащая два поля: поле TypeID типа Short (первичный ключ) и текстовое поле TName. В поле TypeID указаны возможные значения поля NType (тип накладной) в виде чисел от 0 до 7, а в поле TName — расшифровка смысла типа (покупка книг, продажа книг, обмен и т. д.). Создайте для НД Nakls подстановочное поле с именем Type текстового типа, длиной 23 символа, выбрав ключевым поле NType, свяжите его с полем TypeID набора данных Types и укажите в качестве результирующего поле TName этого НД.

Для НД Moves создайте три подстановочных поля с именами Name, Author и Publish, связав их с помощью поля MBook с НД Books (поле BookID) и указав в качестве результирующих полей BName, BAuthor, BPublish соответственно. Эти поля нам понадобятся для формирования составного поля Название книги/Автор/Издательство в списке книг. Размеры полей — 75, 30 и 40 символов соответственно.

Создание объектов-столбцов

После того как мы создали подстановочные поля, связанные с ними ключевые поля NaklsNFirm, NaklsNType и MoveMBook стали «лишними»: не имеет смысла рядом с названием партнера или книги показывать в сетке соответствующий идентификатор. В то же время удалить связанные с ними объекты-поля нельзя, так как НД лишится ключевого поля и подстановочная связь будет разрушена. Конечно, можно для объектов-полей поместить значения False в их свойства Visible и тем самым «спрятать» их и не показывать в сетке. Однако «спрятанные» поля окажутся невидимыми не только в сетке, но и в любом другом визуализирующем компоненте Delphi (эти компоненты сосредоточены на вкладке Data Controls палитры компонентов), что затруднит редактирование и ввод данных для накладных или списка книг.

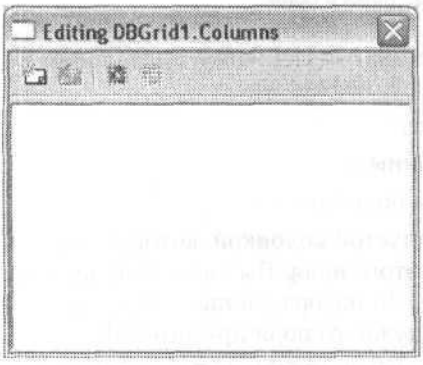


Рис. 1.26. Окно редактора столбцов

Выходом из положения является создание для сетки *объектов-столбцов* — специальных компонентов, облегчающих управление отображением данных. С их помощью можно изменять шрифт и цвет колонок, формировать их заголовки и т. д. Для создания объектов-столбцов нужно вызвать *редактор столбцов* (рис. 1.26) — для этого перейдите к форме fmNaklsU главного окна и дважды щелкните мышью на сетке DBGrid1.

Как и в случае объектов-полей, действует следующее правило: если для сетки не создан ни один объект-столбец, в ней отображаются все не «спрятанные» объекты-поля; и наоборот, если создан хотя бы один объект-столбец, сетка будет отображать данные только из объектов-столбцов.

Чтобы создать объект-столбец, нужно щелкнуть на кнопке Add New редактора столбцов или нажать клавишу Ins. Выделив (щелчком) появившийся в окне компонент TColumn, мы с помощью инспектора объектов можем менять его свойства. Раскройте список свойства FieldName и выберите поле NaklID. Раскройте список вложенных свойств сложного свойства Title и в его вложенное свойство Caption введите заголовок столбца — № накл. Поместите в свойство Width ширину колонки 40 (в пикселах экрана). Руководствуясь табл. 1.2, создайте остальные объекты-столбцы для сеток.

Таблица 1.2. Значения свойств FieldName, Caption и Size объектов-столбцов для сеток

FieldName	Caption	Width
Компонент DBGrid1		
NData	Дата	60
Firm	Партнер	150
Type	Тип накладной	130
NSum	Сумма	60
NPayedSum	Оплата	60
NRetSum	Возврат	60
NCoeff	Коэффициент	35
NRetDate	Срок	60
Компонент DBGrid2		
Name	Название книги/Автор/Издательство	500
MQuan	Количество	40
MPrice	Цена	45

Обработчик события OnGetText

Первый столбец в нижней сетке, как это видно из его заголовка, должен содержать текст, составленный из трех подстановочных полей. Чтобы реализовать такое составное поле, воспользуемся обработчиком события OnGetText объекта-поля MoveName. Перейдите к форме и вызовите редактор полей для НД Moves. Щелкните на поле Name и откройте вкладку Events в окне инспектора объектов. Дважды щелкнув в правой части строки события OnGetText, напишите такой обработчик:

```
procedure TForm1.MoveNameGetText(Sender: TField;
                                 var Text: String;
                                 DisplayText: Boolean);
begin
  Text := MoveName.Value + '/ '+MoveAuthor.Value + '/' +
                                                  MovePublish.Value
end;
```

Событие OnGetText возникает всякий раз, когда программа обращается к полю с целью отображения его содержимого в любом визуализирующем компоненте. Программист в обработчике этого события должен поместить нужную строку в переменную Text. В нашем случае строка получается объединением значений в подстановочных объектах-полях MoveName, MoveAuthor и MovePublish.

Бизнес-правила

Наша простая программа для отображения содержания накладных и связанных с ними списков книг готова. Вы можете ее запустить и просматривать накладные. Не советую вставлять и редактировать записи в накладных и списках книг или удалять их, так как в нашей программе еще не реализован комплекс *бизнес-правил*. Бизнес-правила определяют реакцию системы на добавление, изменение или удаление данных, обеспечивая непротиворечивость и ссылочную целостность БД. Если, например, вы удалите запись из списка книг какой-либо накладной, программа должна автоматически изменить сумму в соответствующей накладной, количество книг на складе и сальдо партнера, в противном случае мы получим недостоверную накладную, неверное количество книг и ошибочное сальдо. Такая реакция программы и реализует бизнес-правило удаления книг.

Бизнес-правила разрабатываются на основе тщательного изучения автоматизируемой области человеческой деятельности. Для файл-серверных систем они обычно реализуются в комплексе обработчиков событий AfterXXXX — BeforeXXXX компонентов-наборов данных TTable или TQuery. В клиент-серверных системах эти действия, как правило, реализуются на сервере БД с помощью *триггеров* — процедур, которые автоматически запускаются при вставке, изменении или удалении записей (см. главу 7).

Рассмотрим небольшой пример, реализующий часть бизнес-правил, связанных с удалением данных о накладной. Перед удалением данных о накладной надо удалить список связанных с ней книг, иначе в таблице MOVES появятся записи, которые ссылаются на несуществующую запись в таблице NAKLS. Наиболее подходящим местом для реализации этого бизнес-правила является обработчик события BeforeDelete набора данных Nakls. Это событие автоматически генерируется перед удалением записи. Перейдите к форме, щелкните на НД Nakls в правой панели модуля и дважды щелкните в строке события BeforeDelete в окне инспектора объектов. Напишите такой обработчик:

```
procedure TForm1.NaklsBeforeDelete(DataSet: TDataSet);
begin
  while not Move.EOF do // Пока в списке книг есть хотя бы
    Move.Delete         // одна книга, удаляем ее
end;
```

В обработчике учитывается то обстоятельство, что НД Moves связан с текущей записью НД Nakls отношением *один ко многим* и, следовательно, содержит данные только по тем книгам, которые относятся к удаляемой накладной.

Аналогичным образом (с помощью обработчика BeforeDelete НД Move) реализуются бизнес-правила, связанные с удалением информации о книге из списка книг.

Транзакции

Транзакциями называется группа действий, переводящая БД из одного целостного состояния в другое и выполняющаяся по принципу «либо все, либо ни одного». Необходимость в транзакциях вызвана тем, что одновременно в одной сети с одними и теми же данными могут работать несколько пользователей. Транзакции определенным образом изолируются друг от друга. Изоляция разрешает одному пользователю видеть данные, которые уже изменены другим пользователем, и эти изменения подтверждены. В результате достигается высокая степень защиты целостности и непротиворечивости данных.

В описываемых далее технологиях есть средства управления транзакциями. Управление заключается в том, что с помощью методов соответствующих компонентов серверу БД передается команда Begin (начать транзакцию), после чего в БД делаются те или иные изменения и серверу передается команда Commit (подтвердить изменения). При получении команды Begin сервер создает буфер для изменяемых данных. При получении команды Commit данные из буфера переносятся в таблицы БД. При этом сервер по всем полям либо по ключевым полям ищет записи, в которые следует внести изменения. Если в процессе изменения данных значения полей изменены в рамках конкурирующей транзакции (другим пользователем), сервер не сможет найти запись и отвергнет изменения. При этом в клиентской программе возбуждается исключение. В процессе обработки исключения программа может дать команду Rollback (откатить изменения), которая заставит сервер восстановить состояние БД, бывшее к моменту начала транзакции.

В рамках технологии BDE.NET, например, клиентская программа управляет транзакциями с помощью трех методов компонента TDatabase: StartTransaction (начинает транзакцию), Commit (подтверждает транзакцию) и Rollback (отменяет все изменения, сделанные с момента старта транзакции):

```
Database1.StartTransaction;  // Стартуем транзакцию
...                          // Любые изменения в любых таблицах
try
 Database1.Commit            // Пытаемся подтвердить изменения
except
 Database1.Rollback          // Откат, если подтвердить не удалось
end;
```

ПРИМЕЧАНИЕ

Откат транзакции ликвидирует также все последствия возможного срабатывания триггеров на сервере БД в рамках отмененной транзакции.

Сервер InterBase обеспечивает три уровня изоляции транзакций: *Dirty Read*, *Read Committed* и *Repeatable Read*. Пусть, например, пользователь А решил изменить коэффициент скидки/наценки в накладной продажи книг, а пользователь Б

мгновение спустя пытается вставить в эту же накладную данные о новой книге. Какой коэффициент будет использовать Б, если пользователь А успел его изменить, но еще не подтвердил транзакцию? Это зависит от того, насколько изолирована транзакция Б от любых других транзакций иных пользователей.

При уровне изоляции *Dirty Read* (буквальный перевод — грязное чтение) пользователь Б видит любые изменения в данных, даже если они еще не подтверждены. В этой ситуации может нарушиться непротиворечивость данных: если пользователь А по каким-либо причинам откатит сделанные им изменения, пользователь Б никак не узнает об этом, и вставка данных о книге пройдет с измененным коэффициентом, хотя в накладной коэффициент останется прежним. В результате сумма в накладной может оказаться неверной (одна из конкурирующих транзакций, как описано далее, может быть отвергнута сервером, поэтому сумма, возможно, будет верной). Уровень *Dirty Read*, фактически, не изолирует транзакции. Только этот уровень возможен для файл-серверных БД, вот почему в СУБД этого типа транзакции используются крайне редко.

Уровень *Read Committed* (чтение подтверждений) разрешает транзакции видеть только подтвержденные изменения. В описанной выше ситуации пользователь Б получит накладную с прежним коэффициентом. Однако если в ходе транзакции он еще раз прочитает накладную, в которую пользователь А успел внести и подтвердить изменение коэффициента, Б увидит эти изменения. Этот уровень рекомендуется для клиент-серверных БД.

Для клиент-серверных БД возможен также уровень *Repeatable Read* (повторяющееся чтение). На этом уровне изоляции также читаются только подтвержденные изменения, однако раз прочитанная запись помещается в локальный буфер и повторно читается из этого буфера, то есть повторное чтение не позволит увидеть изменений, внесенных другим пользователем, даже если эти изменения подтверждены.

Успех или неуспех транзакции во многом зависит от того, по каким столбцам сервер ищет в таблице запись, чтобы внести в нее изменения. Если в нашем примере пользователь А первым подтвердит изменения, а у пользователя Б транзакция настроена на поиск записи по всем полям, его транзакция будет отвергнута, так как в этом случае сервер ищет запись, в которой все столбцы имеют те значения, которые были к моменту старта транзакции Б. Если транзакция настроена на поиск записи, у которой ключевые столбцы и столбцы с изменениями остались прежними, транзакция Б пройдет успешно, так как оба пользователя изменяли разные столбцы, и может быть нарушена непротиворечивость данных. Еще хуже защищаются данные в случае, если транзакция требует от сервера искать запись только по ключевым столбцам.

Таким образом, уровень изоляции *Read Committed* в сочетании с настройкой транзакций на поиск записей по всем полям в максимальной степени защищает данные от искажения в условиях многопользовательской работы.

2 Технология ADO.NET

Технология ADO.NET является базовой технологией для работы с БД на платформе .NET Framework. Она обслуживает серверы MS SQL Server и Oracle, а также серверные и файл-серверные БД, доступные с помощью драйверов ODBC.

На рис. 2.1 представлены основные компоненты технологии ADO.NET.

Здесь *база данных* — реальная серверная БД или XML-файл; *связь* — компоненты, обеспечивающие связь с данными; *адаптеры* — компоненты, формирующие SQL-запросы к серверу БД или драйверу ODBC для получения и/или модификации данных; *набор данных* — компонент, обеспечивающий виртуальное хранилище для полученных и, возможно, измененных данных; *визуализация* — компоненты, обеспечивающие визуальное представление данных.

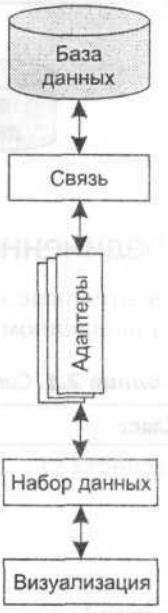


Рис. 2.1. Основные компоненты технологии ADO.NET

Характерным для технологии является наличие промежуточного буфера — набора данных. Набор данных (класс `DataSet`) предоставляет удобный механизм чтения и обновления данных и инкапсулирует в себе, в общем случае, множество таблиц и связей между ними. Набор данных (НД) не связан напрямую с источником данных. Это — виртуальный объект. Таблицы, с которыми он работает, могут быть загружены из источника данных или созданы динамически. После загрузки данных в `DataSet` эти данные могут быть изменены без внесения изменений в источник данных. Зафиксировать изменения в БД можно только после обращения к методу `Update` набора данных.

Для визуализации данных в Delphi разработан единственный объект класса `DataGrid` (категория `Data Controls`).

Он способен предоставлять данные в табличном виде (в виде сетки). Замечательной особенностью базового класса `System.Control` является предусмотренное в нем свойство `DataBindings`, предназначенное для ссылки на связанные с объектом данные. Таким образом, *любой* визуальный компонент WinForms способен визуализировать данные, получаемые из `DataSet`.

Классы ADO.NET

Все классы ADO.NET входят в систему общих типов CTS платформы .NET Framework и условно делятся на требующие соединения с БД (*соединенные классы*) и не требующие соединения (*разъединенные классы*) (рис. 2.2).

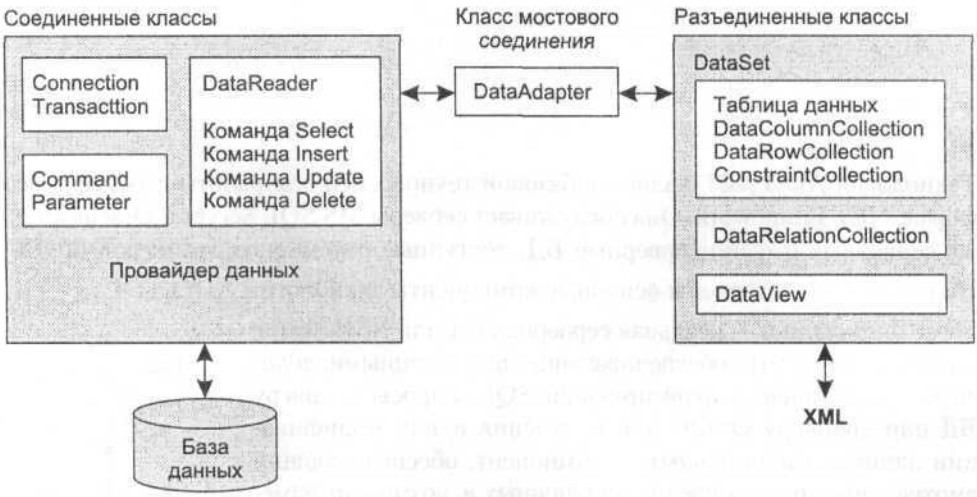


Рис. 2.2. Классы ADO.NET

Соединенные классы

Соединенные классы могут реализовать свою функциональность только при установленном соединении с БД. Некоторые из них перечислены в табл. 2.1.

Таблица 2.1. Соединенные классы ADO.NET

Класс	Описание
`Connection`	Содержит информацию об источнике данных, включая его расположение, имя и другие параметры, специфические для определенного хранилища данных. Класс `Connection` — это канал, через который другие классы получают доступ к данным. Он также способен управлять транзакциями
`Command`	Позволяет применить к данным нужную SQL-команду, в том числе **`SELECT`**, все команды языка DLL, обращение к хранимой процедуре и т. д.

Класс	Описание
`DataReader`	Обеспечивает однонаправленное (только вперед) чтение данных. Обычно используется совместно с классом `DataAdapter`
`DataAdapter`	Мостовой класс (*bridging class*) — реализует связь между разъединенными и соединенными классами. На первом этапе он переносит данные из соединенных в разъединенные классы, предоставляя пользователю возможность манипулировать ими. На втором этапе, наоборот, переносит измененные данные из разъединенных в соединенные классы, внося необходимые изменения в реальные таблицы баз данных (ТБД)
`Parameter`	Содержит параметры для реализации параметрического запроса к БД
`Transaction`	Позволяет объединить группу последовательных обращений к БД в единый блок, выполняемый по принципу «либо все, либо ни одного». Если в ходе выполнения какой-либо команды возникнет ошибка, позволяет откатить (*rollback*) все ранее сделанные в пределах текущей транзакции изменения. Если ошибка не возникает, подтверждает (*commit*) все изменения, сделанные данной транзакцией

Разъединенные классы

В табл. 2.2 представлены разъединенные классы ADO.NET. Функциональность этих классов не зависит от текущего соединения с источником данных.

Таблица 2.2. Разъединенные классы ADO.NET

Класс	Описание
`DataSet`	Класс `DataSet` реализует реляционную БД в оперативной памяти, которой можно управлять, полностью отключив ее от физического хранилища данных
`DataTableCollection`	Коллекция таблиц данных в наборе данных. Позволяет получить доступ к конкретной таблице
`DataTable`	Содержит данные в виде таблицы. Столбцы (*columns*) таблицы есть аналог полей (*fields*) ТБД, а строки (*rows*) — аналог записей (*records*) ТБД
`DataRelation`	Инкапсулирует реляционные связи (*relation*) между таблицами данных `DataTable`. Класс `DataSet` имеет свойство `DataRelationCollection`, содержащее коллекцию объектов `DataRelation`
`DataColumn`	Хранит информацию о структуре столбца таблицы данных. Сюда относится информация о типе данных, доступности, способе вычисления значений и т. п. Этот класс доступен в свойстве `DataColumnCoollection` класса `DataSet`

продолжение

Таблица 2.2 (продолжение)

Класс	Описание
`DataRow`	Предоставляет свойства и методы для редактирования данных в отдельных столбцах таблицы. Класс поддерживается коллекцией `DataRowCollection`
`Constraint`	Содержит ограничения на возможность изменения данных в разъединенной БД (ограничения внешних ключей, реализация каскадных изменений и т. п.). Поддерживается коллекцией `ConstraintCollection`
`DataView`	Позволяет предоставлять данные в `DataSet` нужным способом за счет их фильтрации, сортировки и подобных действий

Провайдеры данных ADO.NET

Провайдеры данных ADO.NET предоставляют соединенным классам конкретные механизмы доступа к тем или иным хранилищам данных. Обычно они оперируют низкоуровневыми функциями (API-функциями) промышленных серверов БД или аналогичными средствами других источников данных.

Провайдеры Microsoft

Корпорация Microsoft укомплектовала платформу .NET Framework версии 1.1 четырьмя провайдерами, описанными в табл. 2.3.

Таблица 2.3. Провайдеры данных платформы .NET Framework версии 1.1

Провайдер	Описание
SQL Server	Обеспечивает доступ к серверу MS SQL Server версии 7.0 и выше. Классы провайдера данных расположены в пространстве имен `System.SqlClient`
OLE DL	Обеспечивает доступ к источникам данных, поддерживающим технологию OLE DB. Классы провайдера расположены в пространстве имен `System.Data.OleDb`
ODBC	Обеспечивает доступ к источникам данных с помощью технологии ODBC (Open Database Connection — открытое соединение с базой данных). Классы провайдера расположены в пространстве имен `System.Data.Odbc`
Oracle	Обеспечивает доступ к промышленному серверу Oracle версий 8.i и 8.1.7. Классы провайдера расположены в пространстве имен `System.Data.OracleClient`

ПРИМЕЧАНИЕ

Провайдеры OLE DB, Oracle и ODBC не устанавливаются по умолчанию, но входят в комплект поставки Delphi 2005. Для их установки следует выбрать команду Component ▸ Installed .NET Components и в открывшемся окне установить флажки рядом с классами `Odbc`*XXXX*, `OleDb`*XXXX* и `Oracle`*XXXX*. Эти компоненты, как и компоненты провайдера SQL Server, появятся в категории Data Components палитры компонентов Delphi.

Провайдер Borland

В состав инструментальных средств Delphi 2005 включен провайдер BDP (Borland Data Provider — провайдер данных корпорации Borland). Это — единственный провайдер, открывающий доступ к нескольким типам источников данных, в том числе к базам данных промышленных серверов InterBase, Oracle, IBM DB2, MS SQL Server и Sybase, а также к СУБД Access, входящей в состав MS Office Professional. Компоненты провайдера расположены в категории Borland Data Provider, а классы провайдера определены в пространстве имен `Borland.Data.Provider`.

Описанные ранее соединенные классы `Connection`, `Command` и др. в реализации провайдеров имеют соответствующий префикс. Для примера в табл. 2.4 показано соответствие имен классов для разных провайдеров.

Таблица 2.4. Соответствие имен классов провайдеров

Базовый класс	SQL Server	OLE DB	BDP
`Connection`	`SqlConnection`	`OleDbConnection`	`BdpConnection`
`Command`	`SqlCommand`	`OleDbCommand`	`BdpCommand`
`DataAdapter`	`SqlDataAdapter`	`OleDbDataAdapter`	`BdpDataAdapter`
`DataReader`	`SqlDataReader`	`OleDbDataReader`	`BdpDataReader`
`Transaction`	`SqlTransaction`	`OleDbTransaction`	`BdpTransaction`

Все соединенные классы реализуют общие базовые интерфейсы, такие как `IDbConnection`, `IDbCommand` и т. д., поэтому в функциональном плане, а также в большинстве своих свойств, методов и событий они идентичны. Далее в разделах этой главы описываются детали работы с провайдером BDP, так как он единственный, поддерживающий «родные» для Delphi базы данных сервера InterBase (этот сервер так же, как и Delphi, разработан корпорацией Borland). Однако этот материал в полной мере применим и к любому другому провайдеру.

Простой пример

Прежде чем начать детальный анализ многочисленных объектов технологии ADO.NET, выполним простой пример, который проиллюстрирует основные приемы работы с указанными ранее объектами. В примере (проект Source\Ch02\Nakls\Nakls.dpr) будет отображаться реляционная связь между таблицами `NAKLS` и `MOVES` БД «Книголюб». В отличие от примера, приведенного в главе 1, в этом примере используется серверный вариант БД.

Конструирование проекта

Начните новый проект WinForms (File ▸ New ▸ Windows Forms Application — Delphi for .NET) и подготовьте установление связи с БД «Книголюб». Для этого в окне менеджера проектов щелкните на вкладке, помеченной значком  , чтобы вызвать окно Data Explorer (рис. 2.3).

Рис. 2.3. Окно Data Explorer

В этом окне демонстрируются соединения со всеми шестью возможными источниками данных. Раскройте узел Interbase. По умолчанию в нем определено соединение IBConn1 с демонстрационной БД, поставляемой вместе с сервером InterBase. Добавим новое соединение. Щелкните на узле правой кнопкой мыши и в контекстном меню выберите команду Add New Connection. В новом окне задайте имя соединения `IBConn2` и щелкните на кнопке OK. В окне редактора соединения (рис. 2.4) все свойства можно оставить по умолчанию, кроме свойства Database: щелкните на кнопке в строке этого свойства и установите ссылку на файл C:\BiblData\IB_Bibl.gdb (разумеется, этот файл следует предварительно загрузить с сайта www.piter.com). Закройте окно редактора соединения щелчком на кнопке OK.

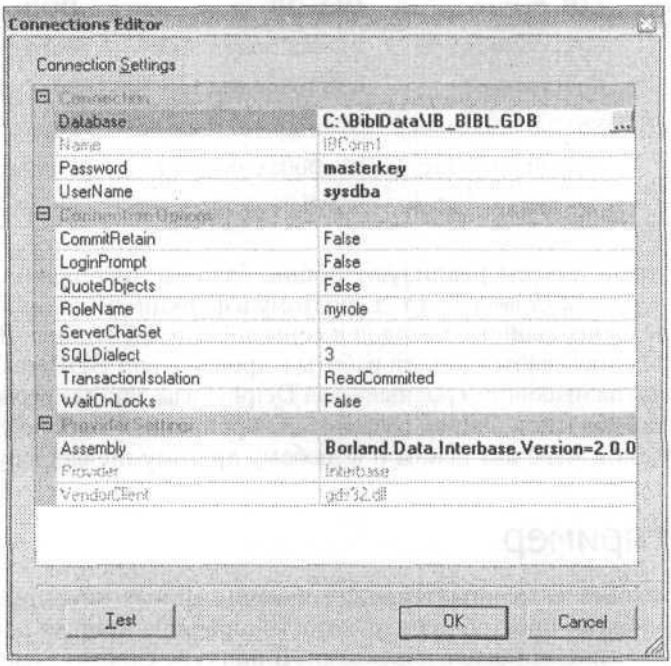


Рис. 2.4. Окно редактора соединения

ПРИМЕЧАНИЕ

В технологии ASP.NET, обсуждаемой в части II этой книги, при работе с БД InterBase в соединении следует указывать правильное DNS-имя[1] узла сети, на котором запущен сервер InterBase. Если сервер запущен на вашем компьютере, ссылку следует отредактировать следующим образом: `localhost:C:\BiblData\IB_BIBL.gdb`.

[1] DNS (Domain Name System — система доменных имен) — это, по существу, распределенная база данных, позволяющая по имени определить местонахождение узла сети.

Теперь раскройте новый узел соединения IBConn2, затем — узел Tables. Разыщите в списке узла таблицу NAKLS и перетащите (Drag&Drop) ее на пустую форму. В результате к проекту будут добавлены компоненты BdpConnection1 и BdpDataAdapter1, причем первый будет настроен на связь с файлом БД IB_BIBL.gdb, а второй — на отображение и модификацию главной таблицы NAKLS этой БД.

Добавьте к проекту еще один адаптер BdpDataAdapter2 (категория Borland Data Provider) и щелчком правой кнопкой мыши вызовите его контекстное меню, в котором выберите команду Configure Data Adapter. В окне конфигурирования (рис. 2.5) выберите таблицу MOVES и щелкните на кнопке Generate SQL.

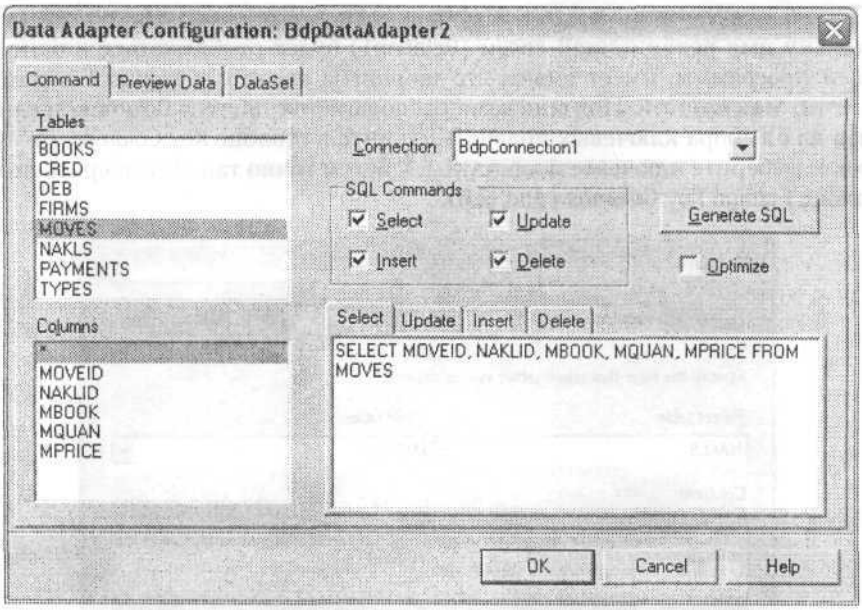


Рис. 2.5. Конфигурирование адаптера для таблицы MOVES

В результате адаптер BdpDataAdapter2 будет настроен на отображение и модификацию детальной таблицы MOVES.

Добавьте к проекту компонент DataSet (категория Data Controls). Поместите на форму сетку DataGrid (категория Data Controls или Crystal Reports) со значением Client свойства Dock.

Настройка набора данных

Настройка набора данных заключается в указании хранящихся в нем данных и связей между ними. Для этого в компоненте определены два свойства-коллекции: Tables и Relations. Первое задает таблицы (точнее, результаты SQL-запросов) и их поля, второе — реляционные связи между ними.

Для связи НД с нужными таблицами поместите в свойства DataSet адаптеров ссылку на компонент DataSet1 и откройте их (поместите значение True в их свойства Active)[1].

Теперь необходимо определить реляционную связь между таблицами. Раскройте редактор свойства Relation компонента DataSet1 и щелкните на кнопке Add. Новое окно содержит имя реляционной связи (Relation1), главной таблицы NAKLS (список Parent table) и детальной таблицы MOVES (список Child table).

ПРИМЕЧАНИЕ

Эти имена определяются порядком открытия адаптеров. Если первым вы открыли адаптер BdpDataAdapter2, имена поменяются местами. В этом случае с помощью списков установите правильные имена.

Поскольку имя реляционной связи (Relation1) будет отображаться в окне работающей программы, имеет смысл его изменить: введите в верхней строке имя Книги по накладной. Под списками располагается таблица Columns, предназначенная для выбора ключевых полей. Щелкните в столбце Key Columns, раскройте список и выберите ключевое поле NAKLID. Затем точно так же выберите это поле в столбце Foreign Key Columns (рис. 2.6).

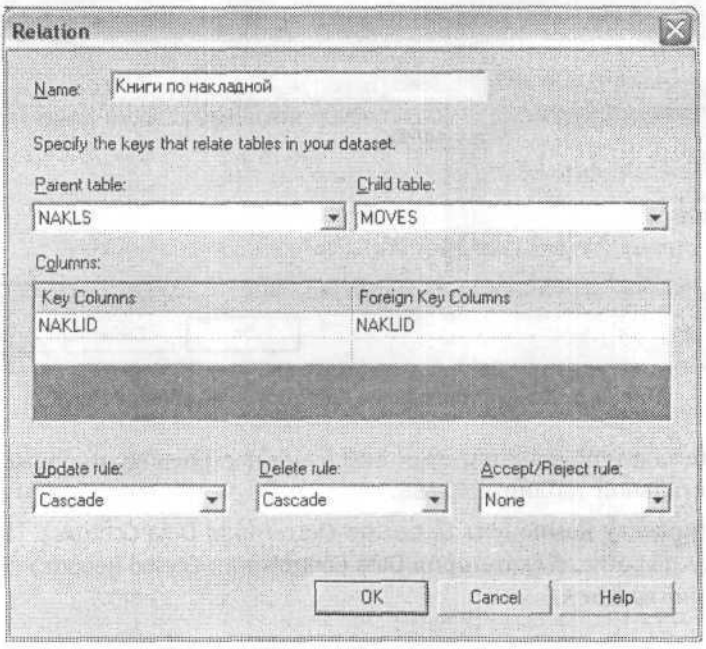


Рис. 2.6. Окно установления реляционной связи

[1] Чтобы избежать ненужной путаницы, сначала откройте адаптер BdpDataAdapter1 и уж затем — BdpDataAdapter2. Ссылки на таблицы появляются в коллекции Tables набора данных по мере открытия адаптеров. При установлении реляционной связи НД считает первую таблицу в этой коллекции главной, а вторую — детальной.

ПРИМЕЧАНИЕ

В технологии ADO.NET первичный ключ главной таблицы и вторичный ключ детальной таблицы должны содержать данные строго одинакового типа и иметь одинаковые названия.

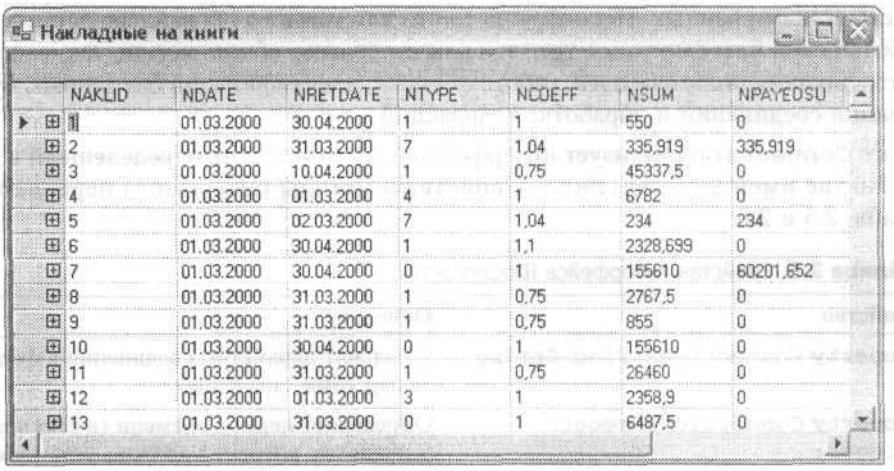
Прогон программы

Осталось настроить компонент DataGrid на отображение данных из главной таблицы. Раскройте список свойства DataSource компонента GataGrid и выберите в нем главную таблицу DataTable1.

Особенностью технологии является то обстоятельство, что обычно набор данных заполняется по требованию программы. Для этого в интерфейсе формы можно предусмотреть специальную кнопку или использовать обработчик события Load формы. Я предпочитаю конструктор формы. Сразу за обращением к процедуре InitializeComponent напишите две строки для заполнения двух таблиц:

```
constructor TWinForm.Create;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;
  //
  // TODO: Add any constructor code after InitializeComponent call
  //
  BdpDataAdapter1.Fill(DataTable1);   // Заполняем 1-ю таблицу
  BdpDataAdapter2.Fill(DataTable2);   // Заполняем 2-ю таблицу
  BdpConnection1.Close;
end;
```

После запуска программы появится окно, показанное на рис. 2.7.



	NAKLID	NDATE	NRETDATE	NTYPE	NCOEFF	NSUM	NPAYEDSU
▶ ⊞	1	01.03.2000	30.04.2000	1	1	550	0
⊞	2	01.03.2000	31.03.2000	7	1,04	335,919	335,919
⊞	3	01.03.2000	10.04.2000	1	0,75	45337,5	0
⊞	4	01.03.2000	01.03.2000	4	1	6782	0
⊞	5	01.03.2000	02.03.2000	7	1,04	234	234
⊞	6	01.03.2000	30.04.2000	1	1,1	2328,699	0
⊞	7	01.03.2000	30.04.2000	0	1	155610	60201,652
⊞	8	01.03.2000	31.03.2000	1	0,75	2767,5	0
⊞	9	01.03.2000	31.03.2000	1	0,75	855	0
⊞	10	01.03.2000	30.04.2000	0	1	155610	0
⊞	11	01.03.2000	31.03.2000	1	0,75	26460	0
⊞	12	01.03.2000	01.03.2000	3	1	2358,9	0
⊞	13	01.03.2000	31.03.2000	0	1	6487,5	0

Рис. 2.7. Окно без отображения детальных данных

В этом окне не отображаются детальные данные, но каждая запись в таблице снабжается значком со знаком + (плюс), щелчок на котором подготавливает окно к показу детальных записей: правей и ниже значка появляется надпись Книги по накладной (название реляционной связи); после щелчка на надписи в окне демонстрируются главная запись и все связанные с ней детальные записи (рис. 2.8).

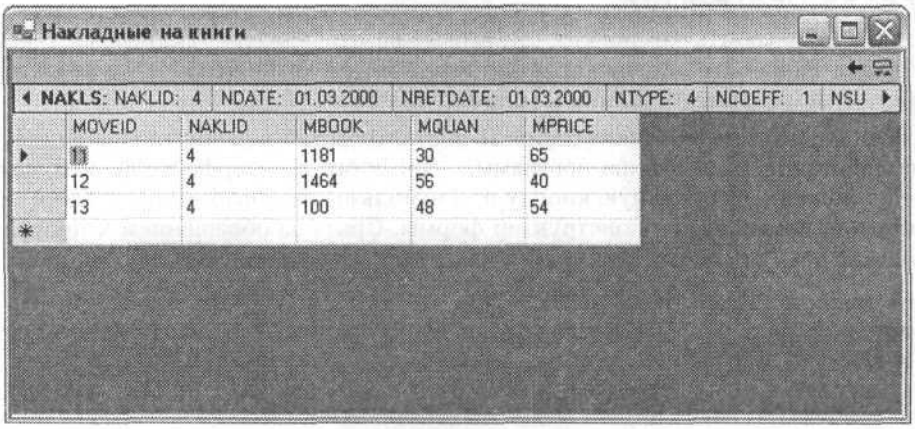


Рис. 2.8. Отображение детальных данных

Чтобы вернуть окно к первоначальному виду, нужно щелкнуть на кнопке с направленной влево маленькой стрелкой в правом верхнем углу окна и закрыть раскрытый узел.

Применение объекта Connection

Объекты класса `Connection` осуществляют соединение приложения ADO.NET с источниками данных. Несмотря на то что для каждого провайдера данных разработан свой класс `Connection`, эти классы решают общие задачи, такие как аутентификация пользователей, работа с сетями, идентификация баз данных, буферизация соединений и обработка транзакций.

Класс `Connection` реализует интерфейс `IDbConnection`, определенный в пространстве имен `System.Data`. Свойства и методы интерфейса перечислены в табл. 2.5 и 2.6.

Таблица 2.5. Свойства интерфейса IDbConnection

Свойство	Описание
`property ConnectionString: String;`	Содержит параметры соединения в формате пар «имя—значение»
`property ConnectionTimeout: Integer;`	Определяет период времени (в секундах), в пределах которого должно установиться соединение. По умолчанию — 15 с

Свойство	Описание
`property Database: String;`	Содержит имя текущей базы данных
`property State: ConnectionState;`	Содержит состояние соединения: `Close` или `Open`

Таблица 2.6. Методы интерфейса IDbConnection

Метод	Описание
`function BeginTransaction: IDbTransaction;`	Инициализирует начало очередной транзакции
`procedure ChangeDatabase(const DatabaseName: String);`	Изменяет текущее имя базы данных
`procedure Close;`	Закрывает соединение с источником данных
`function CreateCommand: IDbCommand;`	Создает объект команды (то есть объект, реализующий интерфейс `IDbCommand`), связанный с текущим соединением
`procedure Open;`	Открывает соединение с источником данных

Задание значения свойства ConnectionString

Свойство `ConnectionString` имеют все провайдеры данных, однако у каждого из них свой набор параметров, которые должно содержать это свойство. Обычно это свойство устанавливается с помощью редактора соединения, в котором, как правило, необходимо указать базу данных, имя пользователя и пароль. Все прочие параметры следует оставить без изменения.

Если по каким-либо причинам необходимо программное указание значения свойства `ConnectionString`, рекомендую на этапе конструирования создать нужное соединение и запомнить строку `ConnectionString`. Например:

```
constructor TWinForm1.Create;
// Этот констуктор создает и тестирует соединение с БД "Книголюб"
var
  Conn: BdpConnection;
const
  ConnStr = 'assembly=Borland.Data.Interbase, Version=2.0.0.0, ' +
            'Culture=neutral, PublicKeyToken=91d62ebb5b0d1b1b;' +
            'database=C:\BiblData\IB_BIBL1251.gdb;username=sysdba;' +
            'password=masterkey';  // Строка соединения
begin
  inherited Create;
  InitializeComponent;
  Conn := BdpConnection.Create(ConnStr);      // Создаем соединение
  try
    Conn.Open;                                // Открываем его...
    if Conn.State = ConnectionState.Open then // и тестируем
      Text := 'OK';
```

```
  finally
    Conn.Close;                          // Закрываем соединение
  end;
end;
```

В приведенном примере строка ConnectionString устанавливает связь с БД «Книголюб» (сервер InterBase). В ней помимо маршрута доступа к файлу БД, имени пользователя и пароля следует указать параметры сборки (параметр Assembly). В примере удалены некоторые параметры (vendorclient, provider), которые присутствуют в строке, создаваемой средствами IDE, но которые, как показывает практика, излишни. При подготовке примера я создал соединение средствами IDE, перенес его строку ConnectionString в константу ConnStr конструктора формы и методом проб и ошибок выяснил, что указанные параметры можно опустить. При работе с другими источниками данных провайдера BDP или при работе с другими провайдерами конкретный состав строки соединения следует подбирать подобным образом (а проще — ничего не менять в этой строке).

Работа с транзакциями

Следующий метод стартует транзакцию:

```
function BeginTransaction(Level: IsolationLevel;
  TransName: String): BdpTransaction
```

Параметр Level задает уровень изоляции транзакций и может принимать одно из следующих значений:

- ReadCommited — данные блокируются на момент их чтения, но могут изменяться другими пользователями до окончания текущей транзакции;
- DirtyRead — нет изоляции транзакций;
- RepeatableRead — данные блокируются и не могут изменяться другими пользователями до окончания текущей транзакции.

Параметр TransName определяет имя транзакции. Функция возвращает объект класса BdpTransaction, у которого есть методы Commit (подтверждает транзакцию) и Rollback (откатывает транзакцию).

ПРИМЕЧАНИЕ

В заключительном разделе этой главы описывается пример использования транзакций.

Получение метаданных

Получить метаданные текущего соединения можно с помощью такого метода:

```
function GetMetedata: ISQLMetadata;
```

Метод возвращает интерфейс ISQLMetadata, с помощью которого можно получить такие данные, как доступные соединению таблицы, их индексы и поля. Интерфейс не имеет свойств. Его методы представлены в табл. 2.7.

Таблица 2.7. Методы интерфейса ISQLMetadata

Метод	Описание
`function GetColumns(szTableName, saColumnName: String; eCol: ColumnType) DataTable;`	Возвращает объект DataTable, содержащий поля таблицы
`function GetIndices(szTableName: String; uIndexType: IndexType): DataTable;`	Возвращает объект DataTable, содержащий индексы таблицы
`function GetProcedureParams (scTableName, szParamName: String): DataTable;`	Возвращает объект DataTable, содержащий параметры хранимой процедуры
`function GetProcedures( szProcName: String; eProc: ProcType): DataTable;`	Возвращает объект DataTable, содержащий перечень хранимых процедур
`procedure GetProperty(eMetaProp: MetaDataProps; out Value: Object);`	Возвращает требуемый параметр метаданных
`function GetTables(szTableName: String; eTable: TableType) DataTable;`	Возвращает объект DataTable, содержащий список всех таблиц заданного типа или имени
`procedure SetProperty(eMetaProp: MetaDataProps; Value: Object);`	Устанавливает нужное свойство метаданных

Компонент имеет единственное событие StateChange, обработчик которого получает два параметра: текущее состояние компонента и его исходное состояние.

События соединения

Класс BdpConnection имеет единственное событие StateChange, которое возникает при изменении его свойства State. Обработчик события в свойствах CurrentState и OriginalState объекта e получает новое и исходное состояния соединения соответственно. Другие провайдеры имеют свой набор событий. Например, класс SqlConnection помимо события StateChange имеет еще событие InfoMessage, которое возникает, когда сервер БД передает информационное сообщение или предупреждение.

Применение объектов Command и DataReader

Объект BdpCommand используется для передачи серверу БД какой-либо команды языка SQL. Результат может не содержать данных вообще, содержать агрегатное значение или содержать множество данных, возвращаемых командой **SELECT**.

Класс BdpCommand реализует интерфейс IDbCommand, свойства и методы которого перечислены в табл. 2.8 и 2.9.

Таблица 2.8. Свойства интерфейса IDbCommand

Свойство	Описание
`property CommandText: String;`	Содержит текст команды
`property CommandTimeOut: Integer;`	Определяет максимальный период выполнения команды в секундах
`property CommandType: CommandType;`	Тип команды: `StoredProcedure` — обращение к хранимой процедуре; `TableDirect` — обращение к таблице; `Text` — SQL-команда
`property Connection: IDbConnection;`	Ссылка на объект `Connection`
`property Parameters: IDataParameterCollection;`	Коллекция параметров для параметрического запроса
`property Transaction: IDbTransatcion;`	Транзакция, в рамках которой будет выполняться команда
`property UpdateRowSource: UpdateRowSource;`	Определяет, как именно результат выполнения команды применяется к редактируемой записи

Таблица 2.9. Методы интерфейса IDbCommand

Метод	Описание
`procedure Cancel;`	Пытается прекратить выполнение команды
`function CreateParameter;`	Создает экземпляр класса `IDbParametr`
`function ExecuteNonQuery: Integer;`	Выполняет команду и возвращает количество затронутых ею записей
`function ExequteReader: IDataReader;`	Выполняет команду и возвращает экземпляр объекта `IDataReader`
`function ExecuteScalar: Object;`	Выполняет запрос и возвращает первый столбец первой записи
`procedure Prepare;`	Подготавливает обращение к хранимой процедуре

Объект, реализующий интерфейс `IDbCommand`, выполняет команды при помощи одного из трех методов: `ExecuteNonQuery`, `ExecuteReader` или `ExecuteScalar`. Специальные реализации интерфейса могут быть дополнены другими методами. Например, класс `SqlCommand` обладает методом `ExecuteXMLReader`, позволяющим обратиться к данным, хранящимся в XML-файле.

Использование метода ExecuteNonQuery

В результате выполнения некоторых SQL-операторов никакие данные не возвращаются. Такими операторами являются **`INSERT`**, **`DELETE`**, **`UPDATE`**, а также обращение к хранимым процедурам, не возвращающим никаких данных.

Для реализации подобных команд интерфейс IDbCommand имеет метод ExecuteNonQuery, который выполняет команду и возвращает количество затронутых ею записей.

В следующем примере (проект Ch02\CommExecNonQuery\ExecNonQueryDemo.dpr) демонстрируется использование метода. К проекту присоединены связной компонент BdpConnection1, настроенный на связь с БД «Книголюб», и компонент-команда BdpCommand1. В поле TextBox1 вводится текст SQL-команды, не возвращающей данные (рис. 2.9).

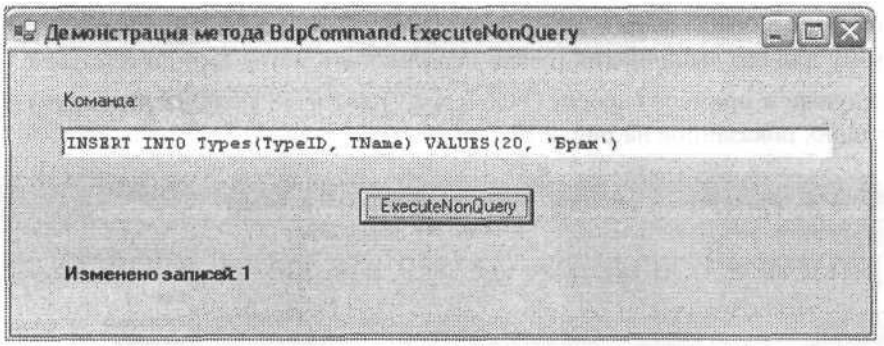


Рис. 2.9. Демонстрация метода ExecuteNonQuery

Ниже показан обработчик щелчка на кнопке ExecuteNonQuery:

```
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  RowsChanged: Integer;
begin
  if TextBox1.Text <> '' then              // Есть текст команды?
  begin                                    //-Да. Пытаемся выполнить
    BdpCommand1.CommandText := TextBox1.Text; //Переносим команду
    BdpConnection1.Open;                     //Открываем соединение
    try
     RowsChanged := BdpCommand1.ExecuteNonQuery; //Выполняем команду
    finally
     BdpConnection1.Close;                  // Закрываем соединение
    end;
    // Сообщаем о количестве измененных записей:
    Label2.Text := 'Изменено записей: ' + RowsChanged.ToString;
  end;
end;
```

Выполнение параметрических запросов

В большинстве случаев запрос имеет изменяемые параметры, что позволяет гибко реагировать на запросы пользователя. В этом случае в текст запроса на месте

изменяемых параметров вставляются символы-заменители. Для провайдера BDP такими заменителями являются символы вопроса (?). Например:

```
SELECT COUNT(*) FROM Books WHERE BPublish = ?
```

Этот запрос возвращает количество наименований книг какого-либо издательства, причем символ вопроса заменяет конкретное название издательства. Чтобы вставить в запрос нужное значение параметра (имя издательства), используется объект класса `BdpParameter`. При его создании в конструкторе `BdpParameter.Create` указываются название параметра, тип и размер данных.

Приведенный выше запрос возвращает единственное значение класса `&Object`, поэтому для его выполнения нужно использовать метод `ExecuteScalar`.

В следующем примере (проект Ch02\CommParamQuery\ParamQueryDemo.dpr) создается окно, показанное на рис. 2.10.

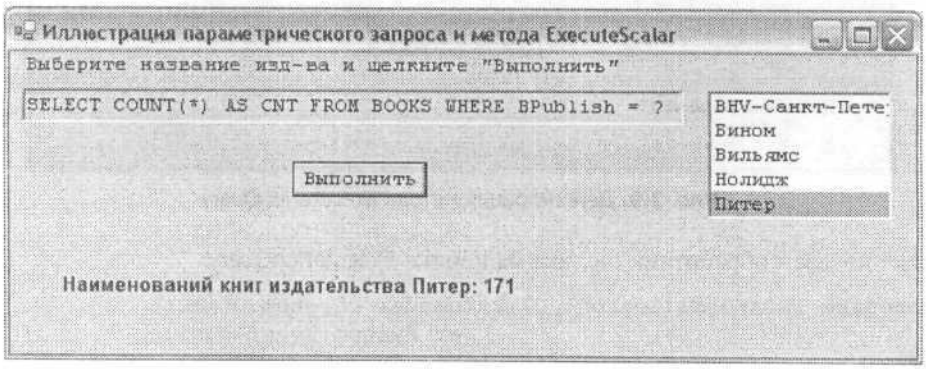


Рис. 2.10. Окно работающей программы

К проекту присоединен объект `BdpConnection1`, настроенный на связь с БД «Книголюб». Ниже показан обработчик щелчка на кнопке Выполнить:

```
procedure TWinForm2.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  Cmd: BdpCommand;
  Par: BdpParameter;
  Obj: &Object;
begin
  if ListBox1.SelectedItem <> NIL then // Выбрано название изд-ва?
  begin                                //-Да. Готовим и выполняем запрос
    // Создаем команду:
    Cmd := BdpCommand.Create(TextBox1.Text, BdpConnection1);
    // Создаем параметр и вставляем его в коллекцию Parameters:
    Par := Cmd.Parameters.Add('PUB', BdpType.&String, 25);
    // Значение параметра берем из списка:
    Par.Value := ListBox1.SelectedItem;
```

```
    BdpConnection1.Open;          // Открываем соединение с БД
    try
      Obj:= Cmd.ExecuteScalar;    // Выполняем параметрическмй запрос
      // Отображаем результат:
      Label1.Text := System.&String.Format(
         'Наименований книг издательства {0}: {1}', Par.Value, Obj);
    finally
      BdpConnection1.Close;       // Закрываем соединение
    end;
  end;
end;
```

Для форматирования результата задействован метод `Format` класса `System.String`. Подробности преобразования строк и использования других классов широкого назначения платформы .NET приведены в приложении Д.

Пример свидетельствует о том, что в провайдере BDP имена параметров никак не используются. Если в запросе имеется несколько символов-заменителей, то конкретные значения параметров берутся из коллекции `BdpCommand.Parameters` последовательно: первый символ вопроса заменяется значением первого элемента коллекции, второй — второго и т. д.

Провайдеры данных Microsoft используют именованные символы-заменители вида @Имя. Например:

```
SELECT COUNT(*) FROM Customers WHERE Country = @Country
```

При создании параметра первым указывается его имя, затем тип и размер:

```
Par := SqlCommand1.Parameters.Add('@Country',SqlDBType.NVarChar,15);
Par.Value := 'USA';
```

Соответствие параметров и их значений устанавливается не порядком определения параметра в коллекции `Parameters`, а их именами в строке запроса и коллекции.

Получение множества результатов запроса

Для получения множества результатов запроса используется метод `ExecuteReader` класса `BdpCommand`. Этот метод возвращает объект класса `BdpDataReader`, исполняющий интерфейс `IDataReader`. Свойства и методы интерфейса представлены в табл. 2.10 и 2.11 соответственно.

Таблица 2.10. Свойства интерфейса IDataReader

Свойство	Описание
`property Depth: Integer;`	Указывает глубину вложенности записей
`property IsClossed: Boolean;`	Содержит `True`, если объект закрыт
`property RecordsAffected: Integer;`	Содержит количество измененных записей

Таблица 2.11. Методы интерфейса IDataReader

Метод	Описание
`procedure Close;`	Закрывает объект
`function GetSchemaTabl: DataTable;`	Возвращает схему данных
`function NextResult: Boolean;`	Переводит курсор к следующей записи и возвращает `True`, если данные не исчерпаны
`function Read: Boolean;`	Читает очередную запись и переводит курсор к следующей. Возвращает `False`, если данных для чтения больше нет

Класс `BdpDataReader` имеет два важных свойства:

- `Item: &Object` — индексированное свойство для доступа к нужному полю записи;
- `FieldCount: Integer` — количество полей в записи.

В качестве индекса свойства `Item` можно указывать имя поля или его индекс (индексация начинается с нуля). Это свойство — умалчиваемое, поэтому два следующих обращения идентичны:

```
var
  DataRdr: BdpDataReader;
  Res: String;
...
  Res := DataRdr[0].ToString;
  Res := DataRdr.Item[0].ToString;
```

Для демонстрации метода `ExecuteReader` подготовлен проект Ch02\CommExecReader\ExecReaderDemo.dpr, который создает окно, показанное на рис. 2.11.

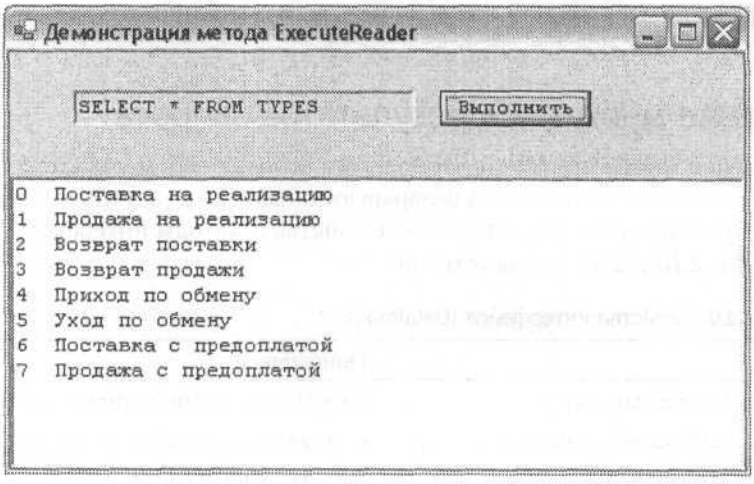


Рис. 2.11. Окно примера

Обработчик щелчка на кнопке Выполнить показан ниже:

```
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  Cmd: BdpCommand;
  Rdr: BdpDataReader;
begin
  Cmd := BdpCommand.Create('SELECT * FROM TYPES', BdpConnection1);
  BdpConnection1.Open;
  try
    Rdr := Cmd.ExecuteReader;
    while Rdr.Read do
      TextBox1.AppendText(
          System.&String.Format('{0}  {1}'#13#10, Rdr[0], Rdr[1]));
  finally
    BdpConnection1.Close;
  end;
end;
```

Символы #13#10 обеспечивают перенос отображаемого текста на следующую строку.

К проекту добавлен компонент BdpConnection1, настроенный на связь с БД «Книголюб».

Класс DataAdapter

Класс DataAdapter представляет собой мост между источником данных и разъединенными объектами, которые позволяют визуализировать данные и управлять ими. Адаптер получает данные из источника данных и помещает их в виртуальное хранилище информации — объект класса DataSet. Кроме того, он кэширует изменения, предоставляя пользователю возможность отказаться от сделанных изменений или внести их в источник данных.

Класс исполняет интерфейс IDBDataAdapter, в котором предусмотрены четыре строковых свойства: SelectCommand, InsertCommand, DeleteCommand, UpdateCommand. В этих свойствах хранятся соответствующие SQL-команды для получения и изменения данных. Каждая команда сопровождается своей коллекцией параметров, обеспечивающей параметрический запрос.

Набор данных DataSet полностью разъединен с исходными данными, поэтому он не имеет никакой информации о соотношении хранящихся в нем данных с исходными данными. Для решения этой задачи в класс DataAdapter включена коллекция TableMappings, содержащая объекты DataTableMapping. Каждый объект DataTableMapping соотносит хранящийся в НД DataSet объект класса DataTable с реальным источником информации (таблицей БД, представлением, XML-файлом и т. п.). Имя таблицы в НД заносится в свойство DataSetTable объекта DataTableMapping, а имя источника информации — в его свойство SourceTable. Для детализации связей между полями

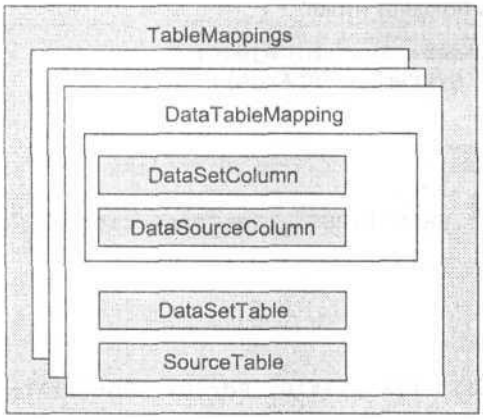


Рис. 2.12. Схема коллекции TableMappings

каждый DataTableMapping имеет коллекцию DataColumnMapping, в которой каждому полю DataSetColumn набора данных ставится в соответствие поле DataSourceColumn источника данных (рис. 2.12).

Свойства, методы и события класса

В табл. 2.12 представлены свойства класса BdpDataAdapter.

Весьма удобным является свойство Active. Как уже отмечалось, разъединенный НД класса DataSet не имеет соотношения хранимых им данных с реальными данными БД или другого источника данных. Если в свойство DataSet адаптера поместить ссылку на НД и в свойство Active установить True, произойдет автоматическое сопоставление данных. В результате сложные свойства TableMappings в адаптере и Tables в НД окажутся нужным образом сформированы. Единственное, что, возможно, придется сделать, — заменить безликие имена таблиц Table, Table2 и т. д. в DataSet.Tables более осмысленными. Замечу, что коллекция DataSet.Relations, определяющая реляционные связи между таблицами, автоматически не формируется.

Таблица 2.12. Свойства класса BdpDataAdapter

Свойство	Описание
`property AcceptShangesDuringFill: Boolean;`	Определяет, будет ли вызываться метод AcceptChages после добавления DataRow к DataTable во время операции наполнения
`property Active: Boolean;`	Если содержит True, НД, указанный в свойстве DataSet, заполняется схемой данных, возвращаемых командой **SELECT**
`property ContinueApdateOnError: Boolean;`	Запрещает/разрешает генерацию исключения при возникновении ошибки в момент подтверждения изменений
`property DataSet: DataSet;`	Содержит ссылку на связанный с компонентом НД класса DataSet
`property DeleteCommand: String;`	Содержит команду **DELETE**
`property InsertCommand: String;`	Содержит команду **INSERT**
`property MaxRecords: Integer;`	Определяет максимальное количество записей, которые будут переданы в НД

Свойство	Описание
`property MissingMappingAction: MissingMappingAction;`	Определяет реакцию на добавление данных, когда не установлено соответствие исходных данных и данных в НД
`property MissingSchemaAction: MissingSchemaActiong;`	Определяет реакцию на добавление данных, когда схема данных в НД ошибочна
`property SelectCommand: String;`	Содержит команду **SELECT**
`property StartRecord: Integer;`	Определяет номер записи, с которой начинается наполнение НД
`property TableMappings: DataTableMappingCollection;`	Коллекция объектов `DataTableMapping`
`property UpdateCommand: String;`	Содержит команду **UPDATE**

С помощью редактора свойства `TableMappings` программист может соотнести данные нужным ему образом. Например, заменить названия полей осмысленными (рис. 2.13).

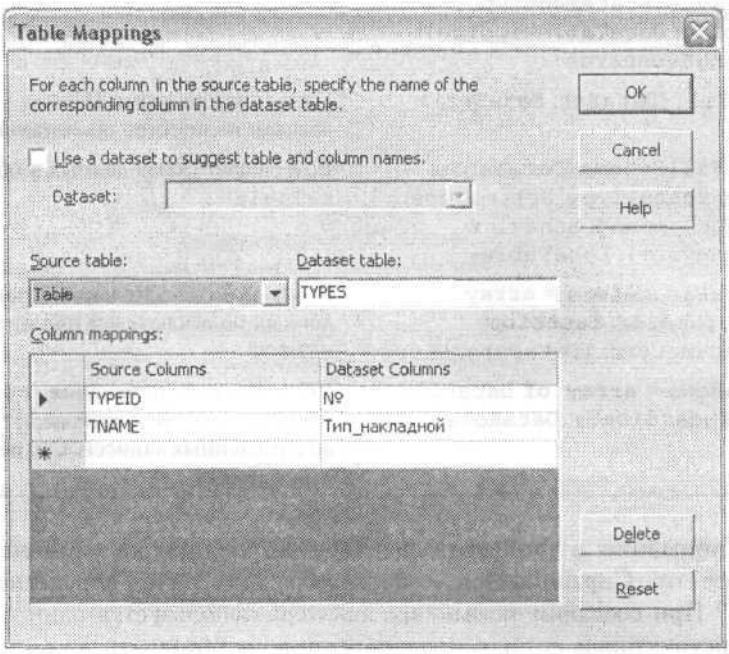


Рис. 2.13. Изменение названий полей в НД

Свойство `MaxRecords` определяет количество записей, считываемых из источника за одно обращение к методу `Fill`. Оно по умолчанию равно 100, то есть из источника считывается не более 100 записей. Если количество просматриваемых/изменяемых записей больше, нужно скорректировать это свойство.

Свойство StartRecord определяет порядковый номер (нумерация начинается с нуля) первой считываемой записи. Если в таблице определен первичный ключ, считываемые записи заменяют собой уже имеющиеся в НД записи с таким же первичным ключом, в противном случае — добавляются в конец.

Методы компонента перечислены в табл. 2.13.

Таблица 2.13. Методы компонента BdpDataAdapter

Метод	Описание
function CreateTableMappings: DataTableMappingCollection;	Создает объект класса DataTableMappingCollection
function CreateRowUpdatedEvent (DatRow: DataRow; Command: IDbCommand; StatementType: StatementType; TableMapping: DataTableMapping): RowUpdetedEventArgs;	Создает аргументы для события RowUpdated
function CreateRowUpdatingEvent (DatRow: DataRow; Command: IDbCommand; StatementType: StatementType; TableMapping: DataTableMapping): RowUpdetingEventArgs;	Создает аргументы для события RowUpdating
function Fill(DataSet: DataSet): Integer;	Наполняет данными НД DataSet и возвращает количество прочитанных записей
function FillSchema(DataTable: DataTable; SchemaType: SchemaType; Command: IDbCommand; Behavior: CommandBehavior): DataTable;	Возвращает схему данных в объекте DataTable
type IDataParameters = **array of** IdataParameter; **function** GetFillParameterd: IDataParameters;	Возвращает массив параметров, установленных пользователем для предложения **SELECT**
type DataRows = **array of** DataRow; **function** Update(Data: DataRows): Integer;	Подтверждает изменения, сделанные в массиве записей, и возвращает количество измененных записей. См. перегруженные методы

Обратите внимание: в свойствах адаптера нет ссылки на компонент класса BdpConnection. Спрашивается — каким образом адаптер определяет связной компонент? При создании экземпляра адаптера используется один из перегруженных конструкторов, которому задается команда **SELECT** и строка соединения (или компонент класса BdpConnection). Если компонент создается на этапе разработки формы, среда автоматически выбирает нужный конструктор. На этапе прогона это должен делать программист. В следующем примере (Ch02\DataTable\FillDataTable.dpr) иллюстрируются техника программного создания адаптера и использование его метода Fill для заполнения объекта DataTable. В примере создается окно, показанное на рис. 2.14.

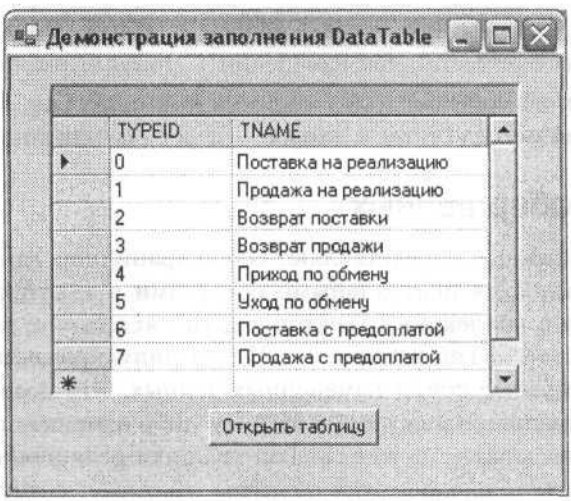


Рис. 2.14. Окно примера

К проекту присоединен компонент `BdpConnection1`, связанный с БД «Книголюб», а на пустую форму помещена сетка `DataGrid`. Вот обработчик щелчка на кнопке Открыть таблицу:

```
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  DA: BdpDataAdapter;
  DT: DataTable;
begin
  // При создании адаптера указываем строку запроса
  // и связной компонент:
  DA := BdpDataAdapter.Create(
                'SELECT * FROM TYPES', BdpConnection1);
  DT := DataTable.Create;      // Создаем таблицу DataTable
  BdpConnection1.Open;         // Открываем соединение
  try
    DA.Fill(DT);               // Наполняем таблицу
  finally
    BdpConnection1.Close;      // Закрываем соединение
  end;
  DataGrid1.DataSource := DT; // Показываем таблицу в сетке
end;
```

Метод `Fill` наполняет НД `DataSet` или, как это сделано в предыдущем примере, объект класса `DataTable` в соответствии с SQL-запросом `SelectCommand`. Если это свойство не определено, возникает исключение.

Для фиксации изменений, сделанных пользователем в НД, используется метод `Update`. Перед обращением к нему должны быть определены соответствующие

параметрические запросы: при вставке записи — запрос InsertCommand, при удалении — DeleteCommand, при изменении — UpdateCommand.

Компонент имеет два события: RowUpdated и RowUpdating. Первое возникает после выполнения метода Update, второе — перед выполнением.

DataSet — набор данных

Наборы данных DataSet представляют собой хранилища данных в оперативной памяти и являются центральными объектами в архитектуре ADO.NET. Данные хранятся в коллекции объектов класса DataTable, доступ к которой открывает свойство Tables компонента. При манипулировании данными в НД сохраняются резервные копии измененных данных. Это позволяет либо внести сделанные изменения в источник данных, либо отменить их. Вместе с таблицами в объектах класса DataRelation хранятся реляционные связи между данными. Доступ к коллекции этих объектов открывает свойство Relations. Свойства Namespace и Prefix используются для обработки данных в виде XML-файлов.

Другие свойства класса DataSet перечислены в табл. 2.14.

Таблица 2.14. Свойства класса DataSet

Свойство	Описание
`property CaseSensetive: Boolean;`	Если содержит True, операции поиска и сравнения строк учитывают высоту букв
`property DefaultViewManager: DataViewManager;`	Возвращает коллекцию наборов данных, использующихся для фильтрации, сортировки и других операций над хранимыми в НД данными
`property EnforceConstraints: Boolean;`	Включает/отключает действие ограничений при выполнении метода Update
`property HasError: Boolean;`	Возвращает True, если хотя бы один объект DataTable имеет ошибку

В табл. 2.15 указаны некоторые методы класса DataSet.

Таблица 2.15. Методы класса DataSet

Метод	Описание
`procedure AcceptChanges;`	Подтверждает все изменения, сделанные в НД
`procedure Clear;`	Удаляет из компонента все данные
`function GetChanges: DataSet;`	Возвращает все изменения в отдельном НД
`function GetXML: String;`	Возвращает данные в виде XML-таблицы
`function HasChanges: Boolean;`	Возвращает True, если НД был изменен

Метод	Описание

`procedure Merge(DataSet);`
`procedure Merge(DataRow);`
`procedure Merge(DataTable);`
	Объединяет заданный НД, `DataRow` или `DataTable` с текущим НД

`function ReadXML(ST: Stream): XMLReadMode;`	Читает XML-данные из потока (см. перекрытые методы)
`procedure RejectChanges;`	Откатывает все изменения в данных

Компонент имеет единственное специфичное событие `MergeFailed`, возникающее при ошибке в методе `Merge`. Обработчик события получает строку с описанием ошибки и ошибочный объект `DataTable`.

Отличительной особенностью технологии ADO.NET является ее тесная интеграция с файлами формата XML (eXtended Markup Language — расширенный язык разметки; подробнее о языке см. приложение В). Обмен данными между различными компонентами реализуется с помощью такого рода файлов. В компоненте `DataSet` есть соответствующие методы для чтения/записи XML.

В следующем примере (Ch02\XML\ProjectXML.dpr) демонстрируется техника работы с этими методами (рис. 2.15):

1. Начните новое приложение WinForms.

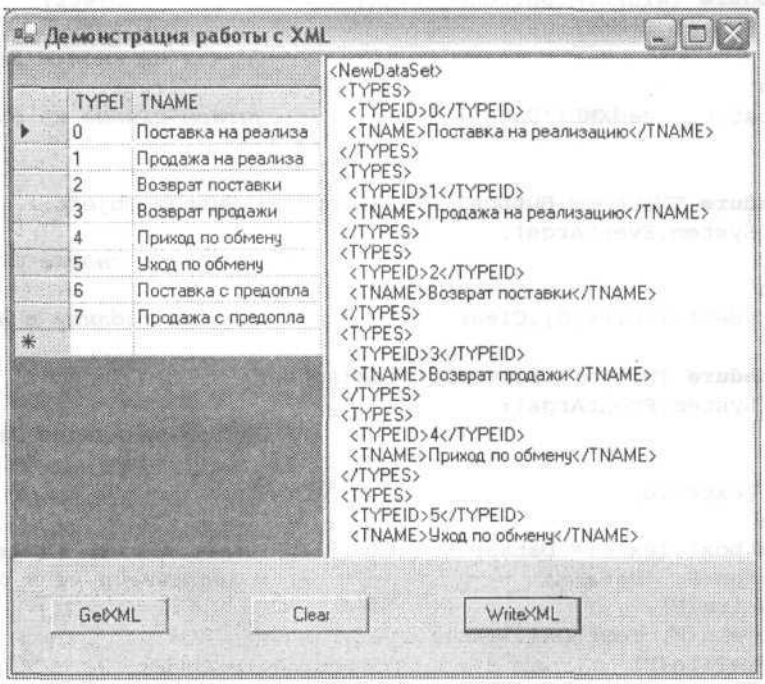


Рис. 2.15. Демонстрация работы с XML

2. На пустую форму поместите три панели. Их свойствам Dock присвойте значения Bottom, Left и Fill.
3. На нижнюю панель поместите три кнопки (GetXML, Clear, WriteXML).
4. На левой панели расположите сетку DataGrid (категория Data Controls), на правой — компонент TextBox. Для каждого из них в свойство Dock поместите значение Fill. Кроме того, в свойство Multiline компонента TextBox поместите значение True.
5. Выберите в меню команду View ▸ Data Explorer и в открывшемся окне Data Explorer раскройте узлы Interbase, IBConn2 и Tables.
6. Перетащите (Drag&Drop) таблицу Types из окна Data Explorer в любое место формы. В результате в нижней части редактора формы появятся невизуальные компоненты BdpConnect и BdpDataAdapter, причем они будут настроены на связь с таблицей Types и отображение всех ее полей.
7. Перетащите на форму компонент DataSet (категория Data components).
8. Поместите в свойство DataSet компонента BdpDataAdapter ссылку на компонент DataSet1, а в свойство Active — значение True. В свойстве DataSource сетки укажите ссылку на таблицу DataTable1. В результате сетка должна заполниться значениями записей из таблицы Types.
9. Напишите такие обработчики событий для кнопок:

```
procedure TWinForm.Button3_Click(sender: System.Object;
  e: System.EventArgs);
                                    // Щелчок на кнопке ReadXML
begin
  DataSet1.ReadXML('Data')          // Читаем данные из файла
end;

procedure TWinForm.Button2_Click(sender: System.Object;
  e: System.EventArgs);
                                    // Щелчок на кнопке Clear
begin
  DataSet1.Tables[0].Clear          // Очищаем таблицу в НД
end;
 procedure TWinForm.Button1_Click(sender: System.Object;
  e: System.EventArgs);
                                    // Щелчок на кнопке GetXML
var
  F: TextFile;
begin
  Textbox1.Text := DataSet1.GetXml; // Читаем таблицу в формате XML
  Assign(F, 'Data');                // и записываем ее в файл
  Rewrite(F);
  WriteLn(F, TextBox1.Text);
  CloseFile(F)
end;
```

DataTable — таблица НД

Объекты DataTable предназначены для хранения данных в НД. Они содержат поля (объекты DataColumn) и записи (объекты DataRow). В одном НД может храниться сколько угодно объектов DataTable.

Многие свойства DataTable дублируют соответствующе свойства DataSet. Например, свойство HasError: если хотя бы у одной таблицы это свойство имеет значение True, аналогичное свойство НД имеет то же значение.

Оригинальные свойства и методы DataTable представлены в табл. 2.16 и 2.17.

Таблица 2.16. Свойства класса DataTable

Свойство	Описание
`property ChildRelations: DataRelationCollection;`	Определяет набор связей с дочерними таблицами
`property Columns: DataColumn Collection;`	Коллекция полей таблицы
`property Constraints: ConstraintCollection;`	Коллекция ограничений, накладываемых на значения полей
`property DataSet: DataSet;`	Ссылка на компонент НД, являющийся контейнером таблицы
`property DefualtView: DataView;`	Возвращает набор данных, использующихся для фильтрации, сортировки и других операций над хранимыми в НД данными
`property DisplayExpression: String;`	Определяет выражение, значение которого будет отображаться в компонентах визуализации
`property ExtendedProperties: PropertyCollection;`	Определяет расширенный набор параметров, которые можно добавить к свойству DataColumn, DataTable или DataSet
`property MinimumCapasity: Integer;`	Определяет минимальное количество записей в таблице (по умолчанию — 25)
`property ParentRelations: DataRelationCollection;`	Определяет набор связей с родительскими таблицами
`property Rows: DataRowCollection;`	Определяет коллекцию записей
`property TableName: String;`	Содержит имя таблицы

Таблица 2.17. Методы класса DataTable

Метод	Описание
`procedure AcceptChanges;`	Подтверждает все изменения, сделанные в таблице

продолжение

Таблица 2.17 (продолжение)

Метод	Описание
`procedure BeginInit;`	Сигнализирует визуальным компонентам о начале инициализации таблицы
`procedure BeginLoadData;`	Отключает систему индексов и ограничений перед чтением данных
`procedure Clear;`	Очищает таблицу от данных
`function Compute(Expr, Filter: String): Object;`	Вычисляет результат выполнения действий, указанных в выражении `Expr`, над группой записей, заданных фильтром `Filter`
`function Copy: DataTable;`	Создает копию таблицы
`function CreateInstance: DataTable;`	Создает экземпляр таблицы
`procedure EndInit;`	Сигнализирует визуальным компонентам о завершении инициализации таблицы
`procedure EndLoadData;`	Включает систему индексов и ограничений после чтения данных
`function GetChanges: DataTable;`	Возвращает изменения в таблице
`function GetErrors: array of DataRow;`	Возвращает массив записей с ошибками
`procedure ImportRow(Row: DataRow);`	Включает в таблицу запись `Row`
`function LoadDataRow(Values: array of Object; isAcceptChanges: Boolean): DataRow;`	Обновляет указанную запись. Если такой записи нет, создает ее
`function NewRow: DataRow;`	Создает новую пустую запись, но не присоединяет ее к таблице
`function NewRowArray(Size: Ibteger): array of DataRow;`	Создает указанное количество пустых записей
`fubction NewRowFromBuilder (Bulder: DataRowBuilder): DataRow;`	Создает новую запись на основе существующей
`procedure RejectChanges;`	Откатывает все изменения таблицы
`procedure Reset;`	Устанавливает таблицу в исходное состояние
`function Select: array of DataRow;`	Возвращает массив записей (см. перегруженные методы)

Для иллюстрации приемов использования некоторых из описанных методов рассмотрим следующий пример (проект Ch02\Create Table\CreateTable.dpr). Создадим таблицу с двумя полями и несколькими записями (рис. 2.16).

Для этого на пустую форму поместите две панели со свойствами `Dock Bottom` и `Fill`. На верхнюю панель поместите компонент `DataGrid` (категория Data Controls) и установите в его свойство `Dock` значение `Fill`. На нижнюю панель

поместите кнопку Button (категория Windows Forms). Напишите такой обработчик щелчка на кнопке:

```
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  DS: DataSet;
  Table: DataTable;
  Column: DataColumn;
  Row: DataRow;
  k: Integer;
begin
  with DS do
  begin
    // Создаем набор данных
    DS := DataSet.Create('DataSet1');
    // Создаем таблицу
    Table := DataTable.Create('DataTable1');
    // Добавляем таблицу в НД
    DS.Tables.Add(Table);
    // Создаем первое поле с именем ID
    Column := DataColumn.Create('ID');
    // Устанавливаем тип поля
    Column.DataType := System.Type.GetType('System.Int32');
    // Добавляем к таблице
    Table.Columns.Add(Column);
    // Создаем второе поле
    Column := DataColumn.Create;
    // Устанавливаем имя поля
    Column.ColumnName := 'Текст';
    // и его тип
    Column.DataType := System.Type.GetType('System.String');
    // Добавляем к таблице
    Table.Columns.Add(Column);
    // Создаем 5 записей
    for k := 1 to 5 do
    begin
      // Создаем пустую запись
      Row := Table.NewRow;
      // Наполняем ее
      Row['ID'] := k;
      Row['Текст'] := 'Строка ' + k.ToString;
      // и добавляем к таблице
      Table.Rows.Add(Row);
    end;
  end;
  DataGrid1.DataSource := DS;  // Связываем сетку с НД
end;
```

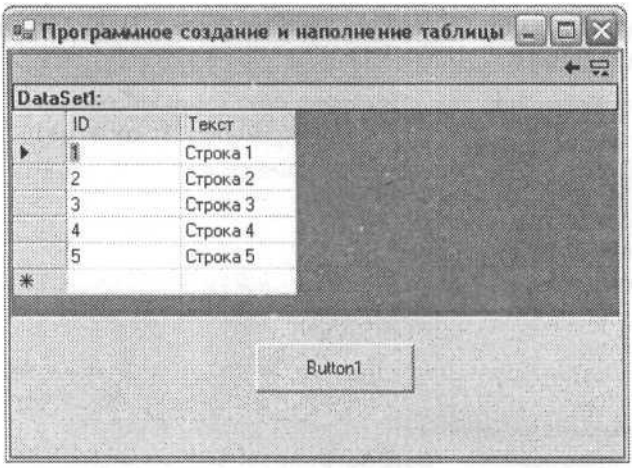


Рис. 2.16. Создание таблицы в работающей программе

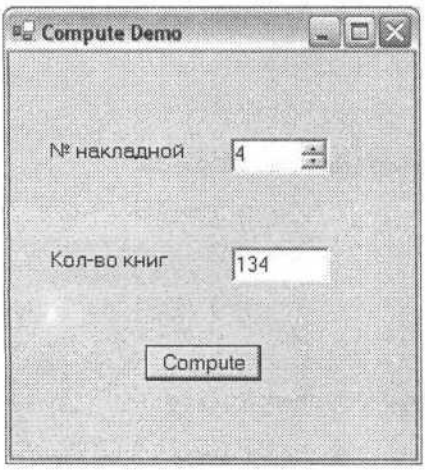


Рис. 2.17. Использование метода Compute

С помощью метода `Compute` можно получить агрегатное значение полей в диапазоне записей, выбираемых параметром `Filter`. Следующая программа (проект Ch02\Compute\ComputeDemo.dpr) подсчитывает общее количество книг, проданных или купленных по заданной накладной (рис. 2.17).

Создайте новое приложение и поместите на пустую форму компоненты `NumericUpDown`, `TextBox` и `Button` (категория Windows Forms). Свяжите с формой также компоненты `BdpConnection`, `BdpDataAdapter` (категория Borland Data Provider) и `DataSet` (категория Data Controls). Настройте их на отображение всех полей таблицы `MOVES` учебной БД. Напишите такой обработчик щелчка на кнопке:

```
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  TextBox1.Text := DataSet1.Tables[0].Compute('SUM(MQUAN)',
                'NAKLID='+NumericUpDown1.Value.ToString).ToString;
end;
```

DataColumn — поля таблиц

Свойство `Columns` таблицы `DataTable` содержит коллекцию полей. Набор полей определяет структуру таблицы. На допустимые значения полей могут

накладываться ограничения. Свойство Constraints таблицы DataTable содержит коллекцию ограничений.

В табл. 2.18 описаны свойства класса DataColumn.

Таблица 2.18. Свойства класса DataColumn

Свойство	Описание
property AllowDBNull: Boolean;	Если содержит True, поле может содержать пустое значение
property AutoIncrement: Boolean;	Если содержит True, поле может автоматически наращивать значения, то есть быть автоинкрементным
property AutoIncrementSeed: Integer;	Содержит начальное значение автоинкрементного поля
property AutoIncrementStep: Integer;	Содержит шаг наращивания автоинкрементного поля
property Caption: **String**;	Определяет заголовок поля в DataGrid
property ColumnMapping: MappingType;	Определяет, как будет представлено поле при сохранении DataSet в формате XML
property ColumnName: **String**;	Позволяет получить или установить имя поля в коллекции Columns таблицы DataTable
property DataType: &Type;	Определяет тип содержащихся в поле данных
property DefaultValue: Object;	Определяет умалчиваемое значение, которое помещается в запись, если для поля не установлено значение
property Expression: **String**;	Определяет выражение для вычисления значения поля
property ExtendedProperties: PropertyCollection;	Определяет дополнительную информацию, связанную с полем
property MaxLength: Integer;	Устанавливает максимальную длину строкового поля
property Ordinal: Integer;	Возвращает положение поля в коллекции Columns
property ReadOnly: Boolean;	Если содержит True, значения поля можно только читать
property Unique: Boolean;	Содержит True, если поле состоит из уникальных значений

Свойство Caption, как сказано в документации, содержит заголовок поля, показываемый в визуализирующих компонентах, таких как DataGrid, например. Несложный опыт показывает, что на самом деле это не так: ни в редакторе полей, ни в работающей программе изменение этого свойства никак не влияет на заголовки полей. Как реально изменить название поля, описано далее в разделе «Изменение названия и скрытие полей».

Свойство ColumnMapping управляет отображением поля в XML-документе и может иметь одно из следующих значений:

- Attribute — атрибут документа;
- Element — элемент документа;
- Hidden — внутренняя структура документа;
- SimpleContent — узел документа.

Обычно это свойство имеет значение Element. Если поместить в свойство значение Hidden, поле не будет передаваться визуализирующим компонентам.

Свойство Expression предназначено для создания так называемых вычисляемых полей, то есть полей, значения которых вычисляются на основе значений других полей той же записи. Создание вычисляемых полей рассматривается далее в разделе «Вычисляемые поля».

Свойство ExtendedProperties представляет собой коллекцию объектов, в которые программист может поместить некоторую дополнительную информацию, связанную с текущим полем, например время последнего обновления значения поля. Это свойство недоступно на этапе создания формы. Для добавления значения в коллекцию ExtendedProperties на этапе прогона программы используется метод Add, который определен следующим образом:

```
procedure Add(Key, Value: Oblect);
```

Здесь Key — строка, задающая название свойства; Value — значение свойства. Например:

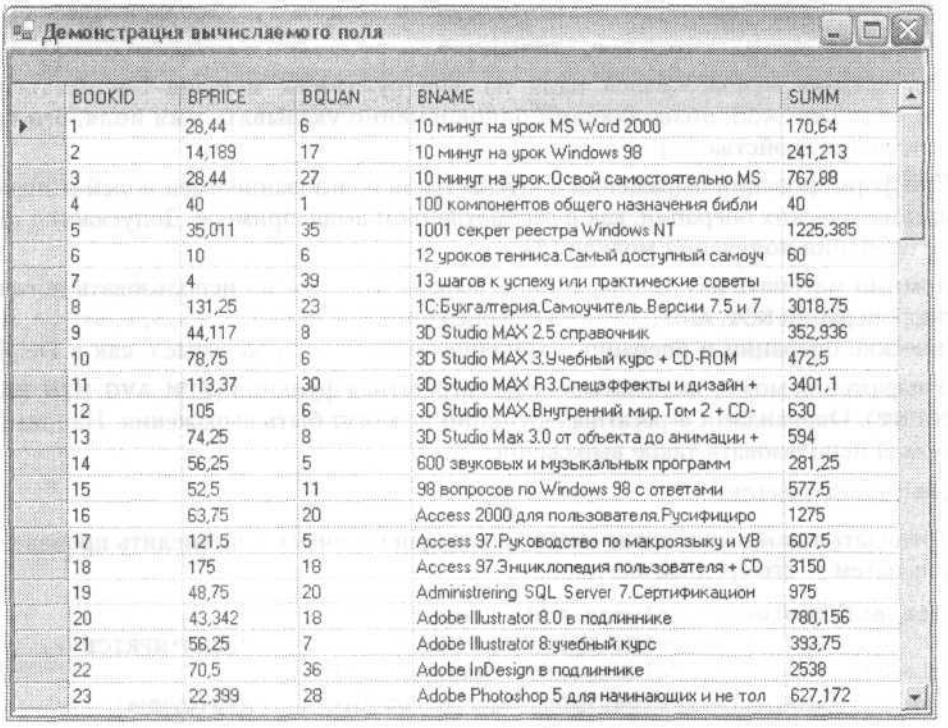
```
procedure TWinForm1.ExtPropDemo;
var
  ExtProp: PropertyCollection;  // Для сокращения длины операторов
begin
  ExtProp := DataSet1.Tables[0].Columns[0].ExtendedProperties;
  // Создаем дополнительное свойство:
  ExtProp.Add('TimeStamp', DateTime.Now);
  // Используем его:
  Console.WriteLine(ExtProp['TimeStamp'].ToString);
end;
```

Вычисляемые поля

Как уже говорилось, вычисляемые поля содержат значения, полученные путем вычисления некоторого выражения из содержимого других полей той же записи. Такие поля нельзя изменять в работающей программе. Есть два способа создания вычисляемого поля: с помощью свойства Expression или формированием сложного запроса в предложении **SELECT**. Далее рассматриваются оба способа.

Свойство Expression определяет выражение, по которому вычисляется значение поля. В выражении указываются поля и знаки математических опера-

ций. Например, в таблице BOOKS есть поля BQUAN и BPRICE. Первое определяет количество экземпляров книги, второе — цену одного экземпляра. Перемножив поля, получим стоимость всех экземпляров каждой из книг, как показано на рис. 2.18 (проект Ch02\Expression\ExpressionDemo.dpr).



BOOKID	BPRICE	BQUAN	BNAME	SUMM
1	28,44	6	10 минут на урок MS Word 2000	170,64
2	14,189	17	10 минут на урок Windows 98	241,213
3	28,44	27	10 минут на урок.Освой самостоятельно MS	767,88
4	40	1	100 компонентов общего назначения библи	40
5	35,011	35	1001 секрет реестра Windows NT	1225,385
6	10	6	12 уроков тенниса.Самый доступный самоуч	60
7	4	39	13 шагов к успеху или практические советы	156
8	131,25	23	1С:Бухгалтерия.Самоучитель.Версии 7.5 и 7	3018,75
9	44,117	8	3D Studio MAX 2.5 справочник	352,936
10	78,75	6	3D Studio MAX 3.Учебный курс + CD-ROM	472,5
11	113,37	30	3D Studio MAX R3.Спецэффекты и дизайн +	3401,1
12	105	6	3D Studio MAX.Внутренний мир.Том 2 + CD-	630
13	74,25	8	3D Studio Max 3.0 от объекта до анимации +	594
14	56,25	5	600 звуковых и музыкальных программ	281,25
15	52,5	11	98 вопросов по Windows 98 с ответами	577,5
16	63,75	20	Access 2000 для пользователя.Русифициро	1275
17	121,5	5	Access 97.Руководство по макроязыку и VB	607,5
18	175	18	Access 97.Энциклопедия пользователя + CD	3150
19	48,75	20	Administrering SQL Server 7.Сертификацион	975
20	43,342	18	Adobe Illustrator 8.0 в подлиннике	780,156
21	56,25	7	Adobe Illustrator 8:учебный курс	393,75
22	70,5	36	Adobe InDesign в подлиннике	2538
23	22,399	28	Adobe Photoshop 5 для начинающих и не тол	627,172

Рис. 2.18. Демонстрация свойства Expression

При использовании свойства мне пришлось изрядно помучиться. Дело в том, что в редакторе полей перед присваиванием значения свойству Expression нужно предварительно установить для него подходящий тип (в нашем случае — Double), однако редактор не допускает изменения типа поля! В результате в режиме конструирования формы поле нормально отображается в сетке, но в режиме прогона становится неопределенным. Пришлось вставлять вычисляемое поле на этапе прогона программы:

```
constructor TWinForm.Create;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;
  //
```

```
  // TODO: Add any constructor code after InitializeComponent call
  //
  // Добавляем новое поле с нужным именем, нужным типом
  // и значением свойства Expression:
  DataSet1.Tables[0].Columns.Add('SUMM',
          System.&Type.GetType('System.Double'), 'BQUAN*BPRICE');
end;
```

Для вставки использовался один из перегруженных методов `DataColumnCollection.Add`, позволяющий одновременно указывать имя поля, его тип и значение свойства `Expression`.

При формировании выражений допускается использование имен полей и знаков математических операций, как в рассмотренном выше примере. Допускается также операция получения модуля (`%`).

Помимо математических операций, в выражениях можно использовать логические операции **NOT**, **AND** и **OR**, а также операции сравнения: =, <>, >, <, >=, <=. Логические операции и операции сравнения имеют такой же смысл, как в Delphi.

В выражения может включаться любая агрегатная функция (**SUM**, **AVG**, **MIN**, **MAX**, **COUNT**). Операндами агрегатных функций не могут быть выражения. Например, нельзя использовать такое выражение:

```
AVG(MQUAN*MPRICE)
```

Проделать такое вычисление можно в два приема: сначала вычислить произведение, затем — его среднее значение:

```
DataSet1.Tables[0].Columns.Add('Сумма',
         System.&Type.GetType('System.Double'), 'MQUAN*MPRICE');
DataSet1.Tables[0].Columns.Add('Средняя сумма')
         System.&Type.GetType('System.Double'), 'AVG(Сумма)');
```

Таким образом, в выражении можно использовать вычисляемые поля. Разумеется, недопустима прямая или косвенная циклическая ссылка вычисляемого поля на само себя.

Следующие пары символов имеют специальное значение и не могут использоваться в именах полей:

- `/n` (перевод строки);
- `/t` (символ табуляции);
- `/r` (возврат каретки).

Помимо этих пар символов в именах полей нельзя использовать следующие специальные символы:

```
~  (  )  #  /  \  =  >  <  +  *  %  ^  |  &  '  "  [  ]
```

Если все же указанные символы входят в имя поля, такое имя нужно заключать в квадратные скобки, например:

```
[Field#1]
```

Если квадратные скобки сами по себе являются частью имени поля, им должен предшествовать символ обратного слеша (\):

```
[Number\[1\]]
```

Язык SQL позволяет создавать довольно сложные выражения. Так как адаптеры используют SQL-запросы, программист может отредактировать предложение **SELECT** так, чтобы вычислить некоторое выражение. Например:

```
SELECT
  BOOKID, BNAME, BQUAN, BPRICE, BQUAN*BPRICE AS SUMM
FROM
  BOOKS
ORDER BY
  BOOKID
```

Этот запрос создаст точно такой же НД, как и созданный ранее с помощью свойства `Expression` (см. рис. 2.18).

Подстановочные поля

Подстановочными (*lookup*) полями являются такие поля, содержимое которых не предназначено для демонстрации в визуальных компонентах; в то же время подстановочное поле содержит ссылку на некоторое поле другой таблицы, выводимое вместо подстановочного. В отличие от вычисляемых полей, значение подстановочного поля можно менять, так как оно является реальным полем реальной таблицы НД. Однако на это значение накладывается ограничение внешнего ключа: значение поля должно совпадать с единственным значением подстановочной таблицы.

Рассмотрим такой запрос.

```
SELECT
  MOVEID, BNAME, MPRICE, MQUAN, MPRICE * MQUAN AS SUMM
FROM
  MOVES, BOOKS
WHERE
  BOOKID = MBOOK
```

В этом запросе помимо полей таблицы `MOVES` и вычисляемого поля `SUMM` выбирается также поле из подстановочной таблицы `BOOKS` на основании критерия `MBOOK = BOOKID`. В данном случае поле `MBOOK` — подстановочное. Оно не может меняться произвольным образом и должно удовлетворять такому ограничению:

```
MIN(BOOKID) <= MBOOK <= MAX(BOOKID)
```

Чтобы гарантировать выполнение указанного ограничения при вводе новой записи или редактировании существующей, обычно создается список выбора, составленный из нужных полей подстановочной таблицы.

Изменение названия и скрытие полей

В ряде случаев имена полей желательно заменить более осмысленными и не отображать ненужные поля. Существует несколько вариантов решения этой проблемы. Наилучший, очевидно, использование так называемых *стилевых объектов* сетки `DataGrid` (см. далее раздел «Использование стилевых объектов»). Однако здесь мы рассмотрим два других варианта: применение свойств `TableMappings` и `Tables` объектов `BdpDataAdapter` и `DataSet`, а также формирование специального запроса в `BdpDataAdapter`.

Рассмотрим, как менять названия полей и скрывать их на примере создания краткого справочника книг из таблицы `BOOKS` (проект Ch02\Mappings\Mepp.dpr):

1. Начните новое приложение WinForms.
2. Выберите в меню команду View ▸ Data Explorer и в дереве данных последовательно раскройте узлы InterBase, IBConn2 и Tables (рис. 2.19).
3. Перетащите (Drag&Drop) таблицу `BOOKS` на форму. В результате в нижней области невизуальных компонентов появятся компоненты `BdpConnection1` и `BdpDataAdapter1`, причем они уже будут настроены на связь с нужной БД и выборку всех полей из таблицы `BOOKS`.
4. Добавьте к ним компонент `DataSet` (категория Data Components).
5. В свойство `DataSet` компонента `BdpDataAdapter` поместите ссылку на НД `DataSet1`.

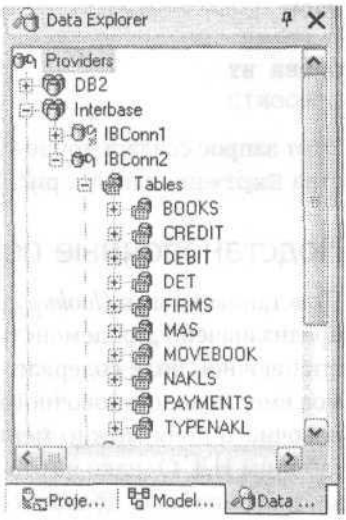


Рис. 2.19. Настройка компонента Data Explorer на работу с таблицей BOOKS

6. Щелчком на кнопке в строке свойства `TableMappings` компонента `BdpDataAdapter1` раскройте редактор этого свойства (рис. 2.20).
7. Установите флажок Use a dataset to suggest table and column names. Это заставит хранить новые имена полей в НД.
8. Измените названия полей так, как показано на рис. 2.21, и закройте редактор.
9. Поместите в свойство `Active` компонента значение `True`, чтобы перенести новые имена полей и данные в НД.
10. Раскройте редактор свойства `Tables` компонента `DataSet1`. Он уже будет содержать таблицу `BOOKS` и все ее поля.
11. Для поля `BISBN` раскройте список свойства `ColumnMapping`, как показано на рис. 2.22, и выберите значение `Hidden`. Поля, помеченные как скрытые (то есть свойство `ColumnMapping` которых равно `HiddenHidden`), не будут демонстрироваться в сетке `DataGrid`.

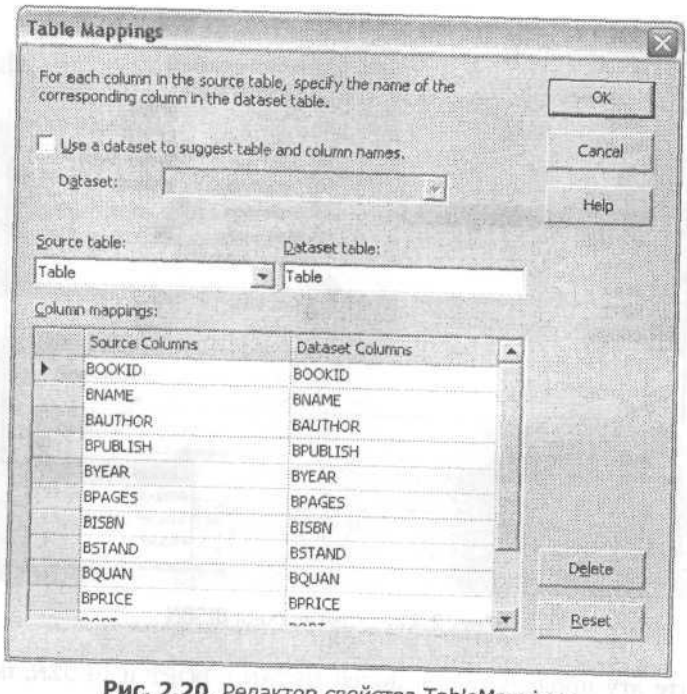


Рис. 2.20. Редактор свойства TableMappings

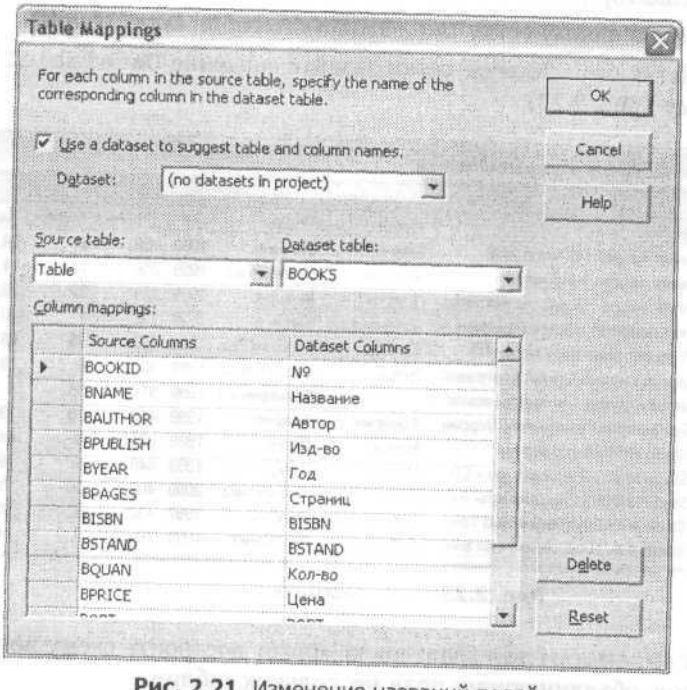


Рис. 2.21. Изменение названий полей

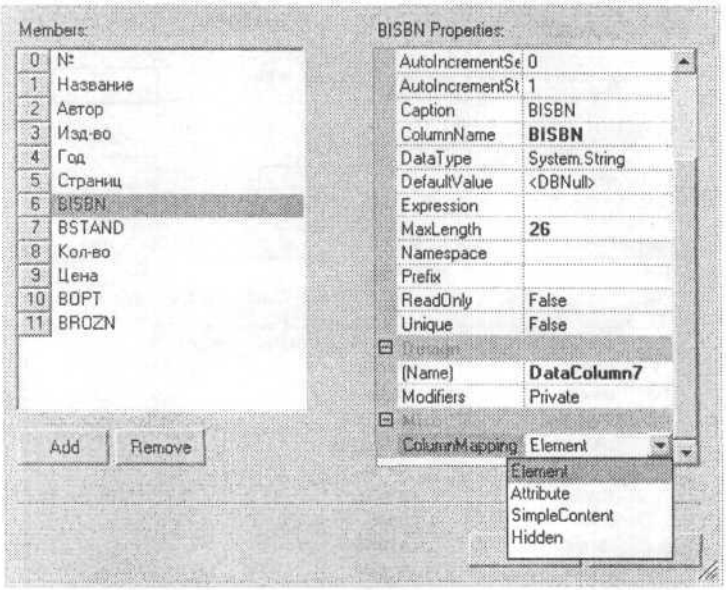


Рис. 2.22. Скрытие поля BISBN

12. Повторите эту процедуру для полей `BSTAND`, `BOPT` и `BROZN`, после чего закройте редактор полей и редактор таблиц.
13. Поместите на форму сетку `DataGrid` (категория **Data Controls**).
14. Для свойства `DataSource` сетки задайте значение `DataTable1` и запустите программу (рис. 2.23).

№	Название	Автор	Изд-во	Год	Страниц	Кол-во	Цена
1	10 минут на урок MS Word 2000	Зйткен	Вильямс	1999	208	6	28,44
2	10 минут на урок Windows 98	Фултон	Диалектика	1999	256	17	14,189
3	10 минут на урок.Освой самостоятел	Фултон	Вильямс	1999	224	27	28,44
4	100 компонентов общего назначения	Архангельск	Бином	1999	272	1	40
5	1001 секрет реестра Windows NT	Дэниелс	Русская Ред	1999	320	35	35,011
6	12 уроков тенниса.Самый доступный	Янчук	Терра-Спорт	1999	30	6	10
7	13 шагов к успеху или практические		Политехника	1998	91	39	4
8	1С:Бухгалтерия.Самоучитель.Версии	Комягин	Триумф	1999	400	23	131,25
9	3D Studio MAX 2.5 справочник	Маров	Питер	1999	672	8	44,117
10	3D Studio MAX 3.Учебный курс + CD-	Маров	Питер	1999	640	6	78,75
11	3D Studio MAX R3.Спецэффекты и д	Белл	Диалектика	2000	400	30	113,37
12	3D Studio MAX.Внутренний мир.Том	Эспиноза-Эг	ДиаСофт	1997	432	6	105
13	3D Studio Max 3.0 от объекта до ани	Кулагин	BHV-Санкт-	1999	480	8	74,25

Рис. 2.23. Окно работающей программы

Свойство `TableMappings` адаптера не может построить схему полей для сложных запросов, объединяющих поля нескольких таблиц.

Еще один способ изменения названий — формирование запросов **SELECT** в компоненте BdpDataAdapter с использованием псевдонимов полей. Для создания псевдонима сразу за названием выбираемого поля ставятся зарезервированное слово **AS** и собственно псевдоним. С помощью показанного ниже запроса создается окно краткого справочника книг (рис. 2.24):

№	Название	Автор	Изд-во	Год	Стран	Кол-во	Цена
1	10 минут на урок MS Word 2000	Зйткен	Вильямс	1999	208	6	28,44
2	10 минут на урок Windows 98	Фултон	Диалектика	1999	256	17	14,189
3	10 минут на урок.Освой самостоятел	Фултон	Вильямс	1999	224	27	28,44
4	100 компонентов общего назначения	Архангельск	Бином	1999	272	1	40
5	1001 секрет реестра Windows NT	Дэниелс	Русская Ред	1999	320	35	35,011
6	12 уроков тенниса.Самый доступный	Янчук	Терра-Спорт	1999	30	6	10
7	13 шагов к успеху или практические с	(не определе	Политехника	1998	91	39	4
8	1С:Бухгалтерия.Самоучитель.Версии	Комягин	Триумф	1999	400	23	131,25
9	3D Studio MAX 2.5 справочник	Маров	Питер	1999	672	8	44,117
10	3D Studio MAX 3.Учебный курс + CD-	Маров	Питер	1999	640	6	78,75
11	3D Studio MAX R3.Спецэффекты и ди	Белл	Диалектика	2000	400	30	113,37
12	3D Studio MAX.Внутренний мир.Том 2	Эспиноза-Зг	ДиаСофт	1997	432	6	105

Рис. 2.24. Изменение названий полей с помощью запроса

```
SELECT
  BOOKID AS "№", BNAME AS "Название", BAUTHOR AS "Автор",
  BPUBLISH AS "Изд-во", BYEAR AS "Год", BPAGES AS "Стран",
  BQUAN AS "Кол-во", BPRICE AS "Цена"
FROM
  BOOKS
ORDER BY
  BOOKID
```

Если псевдоним поля содержит символы кириллицы, он должен обрамляться двойными кавычками.

Обратите внимание на раздел **ORDER BY**. Если его убрать из запроса, ключевое поле BOOKID окажется «закрыто» псевдонимом № и не сможет участвовать в сортировке записей (в этом случае сортировка идет по первому слева строковому столбцу BNAME).

DataRow — записи

В таблице DataTable определено свойство Rows, открывающее доступ к коллекции записей таблицы (объектов класса DataRow). Эти объекты вместе с объектами DataColumn и составляют таблицу.

В табл. 2.19 описаны некоторые свойства записей.

Таблица 2.19. Свойства класса DataRow

Свойство	Описание
`property HasError: Boolean;`	Содержит `True`, если в записи есть ошибки
`property Item(Field: String): Object;`	Открывает доступ к содержимому поля `Field` записи (см. перегруженные свойства)
`property ItemArray: array of Object;`	Представляет все поля записи в виде массива
`property RowState: DataRowState;`	Определяет состояние записи

Свойство `Item` открывает доступ к нужному полю текущей записи для получения или установки его значения. Если вы взглянете на рис. 2.23, то увидите, что цифры в колонке **Цена** не соответствуют денежному представлению. Чтобы округлить цену до не более чем двух цифр после запятой, измените программу следующим образом:

```
implementation
uses
  Borland.VCL.SysUtils;
constructor TWinForm1.Create;
var
  k: DataRow;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;
  //
  // TODO: Add any constructor code after InitializeComponent call
  //
  for k in DataSet1.Tables[0].Rows do
    k.Item['Цена'] := FloatToStrF(k.Item['Цена'] as Double,
                                  ffNumber, 10, 2);
end;
```

На рис. 2.25 показано окно программы.

Обратите внимание на оператор присваивания:

```
k.Item['Цена'] := FloatToStrF(k.Item['Цена'] as Double,
                              ffNumber, 10, 2);
```

В нем левая часть имеет тип `Double` (значение в колонке), а правая — тип `String`. Тем не менее компилятор не сообщит о несовместимости типов, а обратится к методу `Double.Parse` для преобразования строки к типу `Double`.

ПРИМЕЧАНИЕ

При отображении данных путем привязки можно использовать события `Format` и `Parse` класса `Binding` для придания данным желаемого вида. Подробнее об этом см. далее раздел «Использование свойства DataBindings визуальных компонентов».

№	Название	Автор	Изд-во	Год	Стран	Кол-во	Цена
1	10 минут на	Эйткен	Вильямс	1999	208	6	28,44
2	10 минут на	Фултон	Диалектика	1999	256	17	14,19
3	10 минут на	Фултон	Вильямс	1999	224	27	28,44
4	100 компоне	Архангельск	Бином	1999	272	1	40
5	1001 секрет	Дэниелс	Русская Ред	1999	320	35	35,01
6	12 уроков те	Янчук	Терра-Спорт	1999	30	6	10
7	13 шагов к у	(не определе	Политехника	1998	91	39	4
8	1С:Бухгалте	Комягин	Триумф	1999	400	23	131,25
9	3D Studio MA	Маров	Питер	1999	672	8	44,12
10	3D Studio MA	Маров	Питер	1999	640	6	78,75
11	3D Studio MA	Белл	Диалектика	2000	400	30	113,37
12	3D Studio MA	Эспиноза-Эг	ДиаСофт	1997	432	6	105
13	3D Studio Ma	Кулагин	BHV-Санкт-	1999	480	8	74,25

Рис. 2.25. Округление поля «Цена»

Перечисление `DataRowState` представлено в табл. 2.20.

Таблица 2.20. Перечисление DataRowState

Элемент перечисления	Описание
`Added`	Запись присоединена к таблице, но метод `AcceptChanges` не вызывался
`Deleted`	Запись удалена, но метод `AcceptChanges` не вызывался
`Detached`	Запись только что создана, но не является еще частью таблицы
`Modified`	Запись изменена, но метод `AcceptChanges` не вызывался
`Unchanged`	Запись не изменялась с момента последнего вызова метода `AcceptChanges`

В табл. 2.21 представлены методы класса `DataRow`.

Таблица 2.21. Методы класса DataRow

Метод	Описание
`procedure AcceptChanges;`	Подтверждает все сделанные в записи изменения
`procedure BeginEdit;`	Переводит запись в режим редактирования
`procedure CancelEdit;`	Завершает режим редактирования записи и откатывает все сделанные в записи изменения
`procedure ClearErrors;`	Удаляет ошибки из полей записи
`procedure Delete;`	Удаляет текущую запись
`procedure EndEdit;`	Завершает режим редактирования записи

продолжение

Таблица 2.21 (продолжение)

Метод	Описание
`function GetChildRows(DR: DataRelation): array of DataRow;`	Возвращает все детальные записи по реляционной связи DR (см. перегруженные методы)
`function GetColumnError: String;`	Возвращает строку описания ошибки
`function GetColumnsInError: array of DataColumn;`	Возвращает массив полей с ошибками
`function GetParentRow(DR: DataRelation): DataRow;`	Возвращает главную запись по реляционной связи DR (см. перегруженные методы)
`function GetParentRows(DR: DataRelation): array of DataRow;`	Возвращает массив главных записей по реляционной связи DR (см. перегруженные методы)
`function HasVersion(Version: DataRowVersion): Boolean;`	Возвращает True, если для записи есть указанная версия
`function IsNull(Column: DataColumn): Boolean`	Возвращает True, если указанное поле пустое (см. перегруженные методы)
`procedure RejectChanges;`	Откатывает все изменения, сделанные в записи после последнего вызова AcceptChanges
`procedure SetColumnError(Column: DataColumn; Error: String);`	Устанавливает для поля Column описание ошибки Error (см. перегруженные методы)
`procedure SetNull(Column: DataColumn);`	Устанавливает в поле Column пустое значение
`procedure SetParentRow(Parent: DataRow);`	Назначает запись Parent в качестве главной записи

Метод BeginEdit вызывается неявно, когда пользователь начинает изменять величину какого-либо поля записи в визуализирующем компоненте. Также неявно вызывается метод EndEdit при обращении к методу AcceptChanges. Между этими обращениями создаются две версии записи: DataRowVersion.Original и DataRowVersion.Proposed. Наличие той или иной версии можно проверить методом HasVersion. При обращении к нему используется такое перечисление DataRowVersion:

- Current — запись содержит текущие значения;
- Default — запись содержит значения, зависящие от значения свойства DataRowState (если оно имеет значение Added, Modified или Current, запись содержит версию Current, если Deleted — версию Original, если Detached — версию Proposed);
- Original — запись содержит исходные значения;
- Proposed — запись содержит измененные значения.

После выполнения метода BeginEdit становятся доступны версии Current и Proposed. После вызова метода CancelEdit версия Proposed уничтожается. После вызова EndEdit версия Proposed становится версией Current. После вызова AcceptChanges версия Original становится идентичной версии Current.

Наконец, после обращения к методу RejectChanges версия Proposed уничтожается, а запись получает версию Current.

Визуализация данных

Для визуализации данных в ADO.NET используется сетка DataGrid (категория Data Controls палитры компонентов) и свойства DataBindings всех остальных визуальных компонентов.

DataGrid — сетка данных

Сетка DataGrid (категория Data Controls) позволяет визуализировать сразу множество записей, представляя их в табличной форме. В ней предусмотрен специальный механизм одновременной демонстрации главной записи и всех связанных с ней детальных записей. Данные, представленные в строковой форме, можно редактировать непосредственно в сетке.

Для редактирования щелкните на нужной ячейке и введите новый текст. Для удаления записи щелчком мыши на служебном столбце выделите ее и нажмите клавишу Delete. Для ввода новой записи переместите указатель текущей записи к самой последней (пустой) строке и начните заполнение полей.

Помните, что все изменения происходят с данными, хранящимися в DataSet. Чтобы откатить сделанные изменения, используйте метод RejectChanges нужной таблицы компонента DataSet. Например:

```
DataSet1.Tables[0].RejectChanges;
```

Чтобы подтвердить изменения и зафиксировать их в БД, используйте метод Update компонента BdpDataAdapter. Например:

```
BdpDataAdapter1.Update(DataSet1);
```

ПРИМЕЧАНИЕ

При разработке приложений ASP.NET используется одноименный компонент DataGrid, но из пространства имен System.Web.UI.WebControls. Описываемый в этом разделе класс DataGrid принадлежит пространству имен System.Windows.Forms. Несмотря на сходство названий, эти классы имеют разные свойства, методы и события.

Свойства компонента

В табл. 2.22. представлены свойства компонента.

Таблица 2.22. Свойства класса DataGrid

Свойство	Описание
property AllowNavigation: Boolean;	Разрешает или запрещает навигацию по сетке
property AllowSorting: Boolean;	Разрешает или запрещает сортировку данных по выбранному полю

продолжение

Таблица 2.22 (продолжение)

Свойство	Описание
`property AlternatingBackColor: Color;`	Определяет цвет фона всех четных записей (если считать записи, начиная с 1)
`property BackColor: Color;`	Определяет цвет фона всех нечетных записей (если считать записи, начиная с 1)
`property BackgroundColor: Color;`	Определяет цвет той части компонента, которая не занята записями
`property BackgroundImage: Image;`	Определяет фоновый рисунок
`property BorderStyle: BorderStyle;`	Определяет стиль рамки
`property CaptionBackColor: Color;`	Определяет фоновый цвет заголовка сетки
`property CaptionFont: Font;`	Определяет шрифт заголовка сетки
`property CaptionForeColor: Color;`	Определяет цвет шрифта заголовка сетки
`property CaptionText: String;`	Текст заголовка
`property CaptionVisible: Boolean;`	Разрешает или запрещает показ заголовка
`property ColumnHeadersVisible: Boolean;`	Разрешает или запрещает показ названий полей
`property CurrentCell: DataGridCell;`	Возвращает ячейку с фокусом ввода
`property CurrentRowIndex: Integer;`	Возвращает индекс записи с фокусом ввода
`property DataSource: Object;`	Содержит ссылку на источник данных
`property FirstVisibleColumn: Integer;`	Возвращает индекс первого видимого в сетке поля
`property GridLineColor: Color;`	Определяет цвет линий сетки
`property GridLineStyle: DataGridLineStyle;`	Определяет стиль линий
`property HeaderBackColor: Color;`	Определяет фоновый цвет строки с названиями полей и столбца с указателем текущей записи
`property HeaderFont: Font;`	Определяет шрифт строк с названиями полей
`property HeaderForeColor: Color;`	Определяет цвет названий полей
`property Item(CataGridCell): Object;`	Открывает доступ к содержимому указанной ячейки (см. перегруженные свойства)
`property LinkColor: Color;`	Определяет цвет надписи с названием реляционной связи, щелчок на которой показывает детальную таблицу
`property ParentRowsBackColor: Color;`	Определяет фоновый цвет записей главной таблицы
`property ParentRowsForeColor: Color;`	Определяет цвет записей главной таблицы

Свойство	Описание
`property ParentRowsLabelStyle: DataGridParentRowsLabelStyle;`	Определяет способ показа названий главной таблицы и ее поля
`property ParentRowsVisible: Boolean;`	Разрешает или запрещает показ главной записи
`property ReadOnly: Boolean;`	Разрешает или запрещает редактирование ячеек сетки
`property RowsHeaderVisible: Boolean;`	Разрешает или запрещает показ столбца с указателем текущей записи
`property RowsHeaderWidth: Integer;`	Определяет ширину столбца с указателем текущей записи
`property SelectionBackColor: Color;`	Определяет фоновый цвет ячейки с фокусом ввода
`property SelectionForeColor: Color;`	Определяет цвет текста ячейки с фокусом ввода
`property TableStyles: GridTableStyleCollection;`	Содержит коллекцию объектов `DataGridTableStyle`, определяющих стили отображения таблиц

Свойство `AllowNavigation`, как сказано в документации, должно разрешать или запрещать перемещение по записям в сетке. Однако на практике мне не удалось добиться запрета навигации.

Свойство `AllowSorting` по умолчанию имеет значение `True`. В этом случае щелчок на заголовке любого поля приводит к сортировке записей по значениям этого поля.

Свойство `BorderStyle` может принимать одно из следующих значений:

- `Fixed3D` — трехмерная рамка;
- `FixedSingle` — рамка из одной линии;
- `None` — нет рамки.

По умолчанию оно имеет значение `Fixed3D`.

Свойство `GridLineStyle` может иметь значение `Solid` (по умолчанию) или `None`.

Свойство `ParentRowsLabelStyle` определяет вид главной записи в режиме демонстрации связанных с ней детальных записей. Оно может иметь одно из следующих значений:

- `Both` — показываются имя таблицы и все поля главной записи (по умолчанию);
- `ColumnName` — показываются только поля главной записи;
- `None` — ничего не показывается;
- `TableName` — показывается только имя главной таблицы.

Главная запись выводится над детальными записями (см., например, рис. 2.8).

Использование стилевых объектов

Наиболее интересным является свойство `TableStyles`, определяющее особенности отрисовки таблицы данных в компоненте. С помощью этого свойства создаются *стилевые объекты*, содержащие индивидуальные параметры отрисовки каждой таблицы и каждого поля. Компонент может показывать множество таблиц из связанного с ним набора данных, например главную и детальную. Для каждой из них в коллекции `TableStyles` можно предусмотреть свой объект `DataGridTableStyle`, который перекрывает следующие свойства сетки:

- `AllowSorting`;
- `AlternatingBackColor`;
- `BackColor`;
- `ColumnHeadersVisible`;
- `ForeColor`;
- `GridLineColor`;
- `GridLineStyle`;
- `HeaderBackColor`;
- `HeaderFont`;
- `HeaderForeColor`;
- `LinkColor`;
- `PreferredColumnWidth`;
- `PreferredRowHeight`;
- `ReadOnly`;
- `RowHeadersVisible`;
- `RowHeaderWidth`;
- `SelectionBackColor`;
- `SelectionForeColor`.

Объект `DataGridTableStyle` имеет свойство `GridColumnStyles`, открывающее доступ к объектам `DataGridColumnStyle`. Каждый из этих объектов создается для одного из полей таблицы и содержит свойства, управляющие отображением поля в сетке, в том числе:

- `Alignment`;
- `HeaderText`;
- `MappingName`;
- `Width`.

Последовательность работы с объектами стилей такова:

1. Создается объект `DataGridTableStyle`.
2. В его свойство `MappingName` помещается имя таблицы и нужным образом настраиваются другие его свойства.

3. Создается объект DataGridColumnStyle. Поскольку этот класс — абстрактный, для его создания используются конструкторы классов DataGridBoolColumn (для логических полей) или DataGridTextBoxColumn (для остальных полей).
4. В объекте DataGridColumnStyle в свойство MappingName помещается имя отображаемого поля и при необходимости изменяются другие его свойства, после чего объект вставляется в коллекцию GridColumnStyles объекта DataGridTableStyle.
5. Шаги 3 и 4 повторяются для всех полей, которые следует отобразить в сетке. Порядок размещения объектов DataGidColumnStyle в коллекции GridColumnsStyles определяет порядок их отображения (слева направо) в сетке. Поля, для которых не созданы стилевые объекты, отображаться в сетке не будут.
6. После того как объект DataGridTableStyle полностью подготовлен, он вставляется в коллекцию TableStyles сетки DataGrid.

Все описанные действия должны выполняться на этапе прогона программы. В то же время свойство TableStyles объекта DataGrid имеет редактор, позволяющий создавать стили таблиц и полей на этапе конструирования. Однако фактически с его помощью можно создавать лишь стили таблиц: после вставки объекта DataGridTableStyle и привязки его к нужной таблице все стилевые объекты для полей таблицы создаются автоматически. Дальнейшие изменения этих объектов в редакторе на стилях полей никак не отражаются.

Для иллюстрации изложенного начните новое приложение WinForms (проект Ch02\Styles\Styles.dpr), на пустую форму поместите компонент DataGrid и его свойству Dock присвойте значение Fill. Выберите в меню команду View ▸ DataExplorer и перетащите таблицу Types на компонент DataGrid. Добавьте к форме компонент DataSet, в свойстве DataSource компонента BdpDataAdapter сошлитесь на этот набор данных и откройте адаптер (поместите True в его свойство Active). В свойство DataSource сетки поместите ссылку на таблицу DataTable1. Подготовьте указанный ниже обработчик события Load формы:

```
procedure TWinForm1.TWinForm1_Load(sender: System.Object;
                                   e: System.EventArgs);
var
  TabStyle: DataGridTableStyle;
  ColStyle: DataGridColumnStyle;
begin
  // Создаем объект DataGridTableStyle:
  TabStyle := DataGridTableStyle.Create;
  // Связываем его с таблицей TYPENAKL:
  TabStyle.MappingName := DataSet1.Tables[0].TableName;
  // Другой вариант: TabStyle.MappingName := 'TYPENAKL'
  // Выделяем цветом нечетные ячейки:
  TabStyle.AlternatingBackColor := System.Drawing.Color.Yellow;
```

```
  // Создаем 1-й объект DataGridColumnStyle:
  ColStyle := DataGridTextBoxColumn.Create;
  // Связываем его с полем TypeID:
  ColStyle.MappingName := 'TypeID';
  // Задаем название колонки:
  ColStyle.HeaderText := '№';
  // Текст - прижать вправо:
  ColStyle.Alignment := HorizontalAlignment.Right;
  // Устанавливаем ширину колонки:
  ColStyle.Width := 25;
  // Помещаем стиль в коллекцию:
  TabStyle.GridColumnStyles.Add(ColStyle);
  // Создаем 2-ю колонку:
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'TName';
  ColStyle.HeaderText := 'Тип накладной';
  ColStyle.Alignment := HorizontalAlignment.Center;
  ColStyle.Width := 150;
  TabStyle.GridColumnStyles.Add(ColStyle);
  // Объект DataGridTableStyle готов. Добавляем его в коллекцию:
  DataGrid1.TableStyles.Add(TabStyle);
end;
```

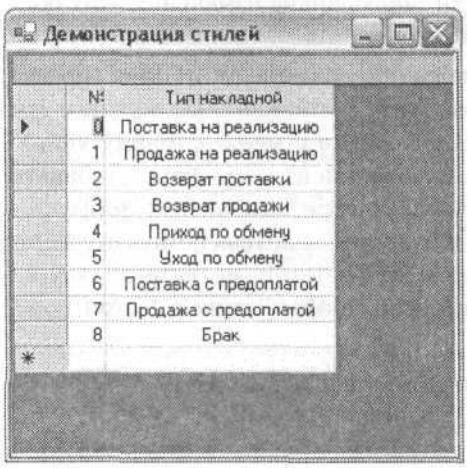


Рис. 2.26. Демонстрация стилевых объектов

Вид окна работающей программы показан на рис. 2.26.

Методы компонента

В табл. 2.23 перечислены методы компонента.

Использование свойства DataBindings визуальных компонентов

При первом знакомстве с ADO.NET поражает отсутствие каких бы то ни было компонентов для визуализации отдельных полей записи. В то же время в практике работы с БД помимо сеточного представления множества записей широко используются так называемые *формы*, служащие для редактирования отдельной записи. Такого рода формы представляют собой контейнер (панель, форма и т. п.) с визуальными компонентами, каждый из которых связан с отдельным полем редактируемой записи.

В классе `Control` определено такое свойство:

```
property DataBindings: ControlBindingsCollection;
```

Таблица 2.23. Методы класса DataGrid

Метод	Описание
`function BeginEdit(GridColumn: DataGridColumnStyle; RowNumber: Integer): Boolean;`	Переводит ячейку в режим редактирования и возвращает `True` в случае успешной операции
`procedure CancelEditing;`	Прекращает редактирование и откатывает все изменения
`function CreateDataGridColumn (ColProp: PropertyDescriptor): DataGridColumnStyle;`	Создает новую колонку с заданными свойствами
`function EndEdit(GridColumn: DataGridColumnStyle; RowNumber: Integer; ShouldAbort: Boolean): Boolean;`	Требует завершения редактирования указанной ячейки и возвращает `True` в случае успешного завершения. Параметр `ShouldAbort` должен иметь значение `True`, если требуется прервать текущий процесс
`procedure Expand(Row: Integer);`	Показывает детальные записи для указанной главной
`function GetCellBounds(Row, Column: Integer): Rectangle;`	Возвращает границы заданной ячейки (см. перекрытые методы)
`finction GetCurrentCellBounds: Rectangle;`	Возвращает границы ячейки с фокусом ввода
`function HetTest(P: Point): HitTestInfo;`	Возвращает объект `HitTestInfo`, содержащий информацию о свойствах `DataGrid` в заданной точке `P`
`function IsExpanded(RowNumber: Integer): Boolean;`	Возвращает `True`, если главная запись `RowNumber` сопровождается показом связанных с ней детальных записей
`function IsSelected(Row: Integer): Boolean;`	Возвращает `True`, если запись `Row` имеет фокус ввода
`procedure NavigateBack;`	Сворачивает детальные записи и показывает главную таблицу
`procedure NavigateTo(Row: Integer; RelationName: String);`	Показывает детальные записи для записи `Row` и реляционной связи `RelationName`
`procedure Select(Row: Integer);`	Передает фокус ввода указанной записи
`procedure SetDataBinding(DS: DataSet; Tabl: String);`	Связывает компонент с нужным набором данных и его таблицей
`procerure UnSelect(Row: Integer);`	Удаляет фокус ввода из указанной записи

Поскольку класс `Control` является родоначальником всех визуальных компонентов, свойство `DataBinding` получают все его потомки.

Это свойство открывает доступ к коллекции `ControlBindingsCollection`, содержащей объекты `Binding`. Каждый объект коллекции устанавливает связь между

каким-либо свойством визуального компонента и полем БД. В результате любой визуальный компонент библиотеки WinForms может отображать данные из БД.

Свойство `DataBindings` — сложное, и обычно оно свернуто, о чем сигнализирует значок слева от него. Щелчок на значке разворачивает два или три подсвойства, в том числе `Advanced` и `Tag`. Именно эти подсвойства визуального компонента могут отображать данные. Если среди них нет нужного, следует щелкнуть на кнопке в строке подсвойства `Advanced`. В результате раскрывается окно расширенной привязки, показанное на рис. 2.27.

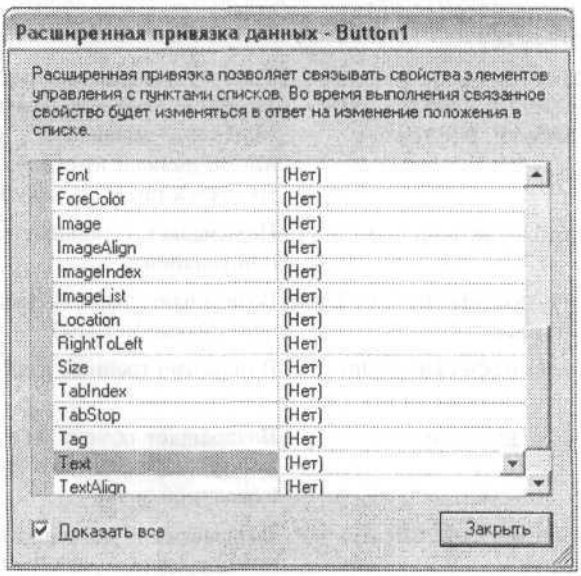


Рис. 2.27. Расширенная привязка

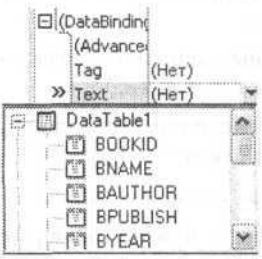


Рис. 2.28. Связывание свойства визуального компонента с полем таблицы БД

В этом окне представлены все свойства компонента. Выберите нужное (обычно — `Text` или `Value`) и закройте окно. После этого список свойств дополнится нужным. Чтобы указать поле БД, которое будет отображать это свойство, щелкните на кнопке в строке свойства (рис. 2.28).

В раскрывшемся списке щелкните на нужном поле. В результате свойство визуального компонента в работающей программе будет отображать значение поля в текущей записи таблицы БД, и изменение отображаемого значения визуального компонента приведет к изменению значения поля в таблице.

Пример создания формы

Для иллюстрации сказанного служит программа (проект Ch02\Form\Form.dpr), окно которой показано на рис. 2.29.

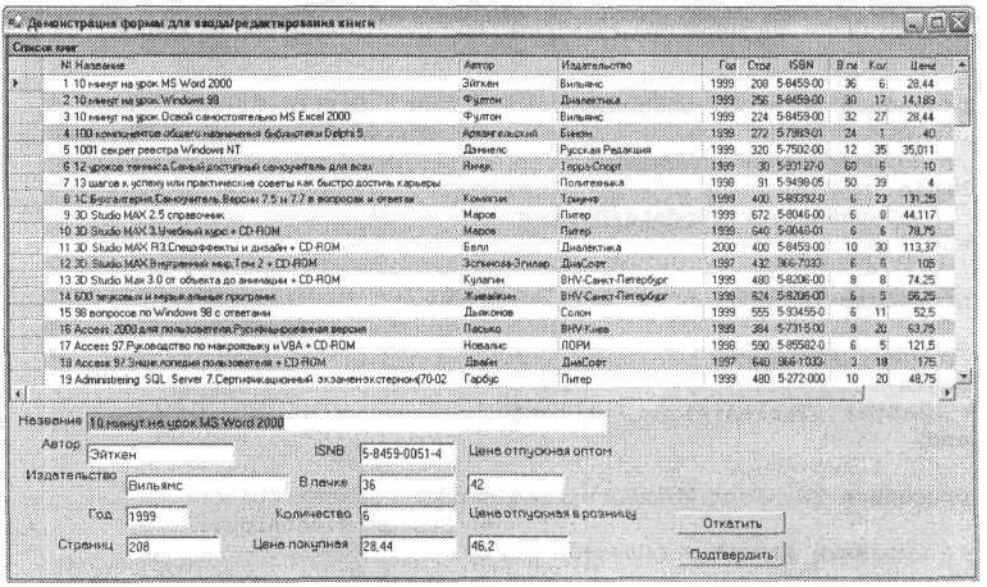


Рис. 2.29. Форма для ввода/редактирования данных о книге

С помощью этой программы можно вводить данные о новой книге в список уже зарегистрированных книг в таблице BOOKS БД «Книголюб», удалять из списка данные о книге и редактировать данные уже существующей в списке книги.

Верхнюю часть окна занимает сетка, отображающая данные сразу для множества книг, нижняя часть представляет собой форму для ввода/редактирования данных о текущей книге — той, на которую указывает значок в служебном столбце сетки. При перемещении этого указателя по записям сетки автоматически меняется информация в компонентах формы.

Для удаления данных о ненужной книге выделите ее в сетке щелчком на служебном столбце и нажмите клавишу Delete. Для вставки данных о новой книге переведите указатель записи на самую нижнюю (пустую) запись и начните вводить информацию в поля записи с помощью сетки или компонентов формы. Учтите, что таблица BOOKS содержит данные о более чем 2000 книг, а в сетке отображается лишь та их часть, которая выбрана значениями свойств StartRecord и MaxRecords адаптера. По умолчанию эти свойства имеют значения 0 и 100 соответственно, то есть в сетке отображается лишь первая сотня книг. Новая запись вставляется в самый конец таблицы. Учтите также, что поле BookID — автоинкрементное и оно не должно быть пустым. Поэтому перед подтверждением ввода новой записи введите в это поле любое число — при вставке записи в таблицу БД сработает так называемый триггер, связанный с событием INSERT, и заменит это значение нужным. Подробнее о триггерах см. главы 1 и 7. Ниже представлен текст триггера BEF_INS_BOOK:

```
CREATE TRIGGER BEF_INS_BOOK FOR BOOKS BEFORE INSERT POSITION 0 AS
BEGIN
  NEW.BOOKID = GEN_ID(GEN_BOOKS,1);
END
```

В следующем листинге представлены обработчики щелчков на кнопках и события Load формы:

```
procedure TWinForm.btSubmit_Click(sender: System.Object;
                                  e: System.EventArgs);
// Подтверждение изменений
begin
  BdpDataAdapter1.Update(DataSet1);
end;

procedure TWinForm.btRollBack_Click(sender: System.Object;
                                    e: System.EventArgs);
// Откат изменений
begin
  DataSet1.Tables[0].RejectChanges
end;

procedure TWinForm.TWinForm_Load(sender: System.Object;
                                 e: System.EventArgs);
// Создание стилевых объектов сетки
var
  TabStyle: DataGridTableStyle;
  ColStyle: DataGridColumnStyle;
begin
  TabStyle := DataGridTableStyle.Create;
  TabStyle.MappingName := 'BOOKS';
  TabStyle.AlternatingBackColor := System.Drawing.Color.LightGray;
  DataGrid1.CaptionText := 'Список книг';
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BookID';
  ColStyle.HeaderText := '№';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 30;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BNAME';
  ColStyle.HeaderText := 'Название';
  ColStyle.Alignment := HorizontalAlignment.Left;
  ColStyle.Width := 400;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BAUTHOR';
  ColStyle.HeaderText := 'Автор';
  ColStyle.Alignment := HorizontalAlignment.Left;
  ColStyle.Width := 100;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BPUBLISH';
  ColStyle.HeaderText := 'Издательство';
  ColStyle.Alignment := HorizontalAlignment.Left;
  ColStyle.Width := 150;
```

```
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BYEAR';
  ColStyle.HeaderText := 'Год';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 35;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BPAGES';
  ColStyle.HeaderText := 'Страниц';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 35;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BISBN';
  ColStyle.HeaderText := 'ISBN';
  ColStyle.Alignment := HorizontalAlignment.Center;
  ColStyle.Width := 60;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BSTAND';
  ColStyle.HeaderText := 'В пачке';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 30;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BQUAN';
  ColStyle.HeaderText := 'Количество';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 30;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BPRICE';
  ColStyle.HeaderText := 'Цена';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 50;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BOPT';
  ColStyle.HeaderText := 'Оптом';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 50;
  TabStyle.GridColumnStyles.Add(ColStyle);
  ColStyle := DataGridTextBoxColumn.Create;
  ColStyle.MappingName := 'BROZN';
  ColStyle.HeaderText := 'В розницу';
  ColStyle.Alignment := HorizontalAlignment.Right;
  ColStyle.Width := 50;
  TabStyle.GridColumnStyles.Add(ColStyle);
  DataGrid1.TableStyles.Add(TabStyle);
end;
```

Класс Binding

Класс `Binding` содержит свойства, методы и события, обеспечивающие однозначную привязку визуального элемента к столбцу таблицы `DataTable`. Свойства класса `Binding` перечислены в табл. 2.24.

Таблица 2.24. Свойства класса Binding

Свойство	Описание
`property BindingManagerBase: BindigManagerBase;`	Управляет текущей записью в отображаемых данных
`property BindingMemberInfo: BindingMemberInfo;`	Содержит информацию об отображаемом элементе данных
`property Control: Control;`	Содержит ссылку на визуальный компонент, отображающий данные
`property DataSourse: Object;`	Содержит ссылку на источник данных
`property IsBinding: Bolean;`	Содержит `True`, если объект класса активен
`property PropertyName: String;`	Содержит имя свойства визуального компонента, с помощью которого отображаются данные

Объект класса `Binding` создается таким конструктором:

```
constructor Create(PropName: String; DataSet: Object; Field: String);
```

Здесь `PropName` — имя свойства, в котором будут отображаться данные; `DataSet` — набор данных; `Field` — привязываемый элемент данных.

С классом связаны два события: `Format` и `Parse`. Первое возникает при передаче данных в элемент отображения, второе — при обратной передаче. В этих обработчиках можно применять нужные методы форматирования для придания данным желаемого вида.

Использование событий рассмотрим на примере проекта Ch02\FormBindFormat\FormBindFormat.dpr (рис. 2.30).

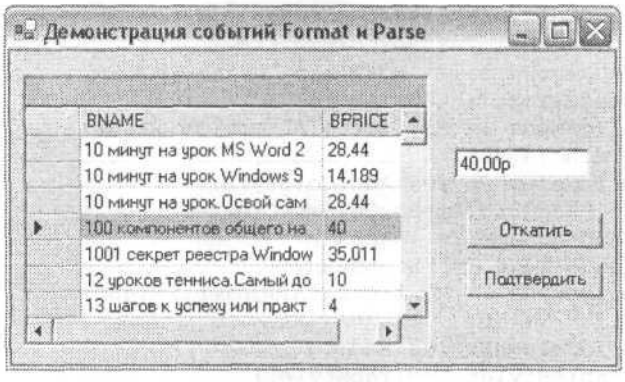


Рис. 2.30. Демонстрация событий Format и Parse

Рядом с таблицей элемент txbPrice класса TextBox отображает цену покупки книги. В таблице BOOKS поле BPrice имеет тип Double. В обработчике Format этот тип представляется как денежный, а в обработчике Parse происходит обратное преобразование:

```
implementation
uses Borland.VCL.SysUtils;

constructor TWinForm.Create;
var
  bdnPrice: Binding;
begin
  inherited Create;
  InitializeComponent;

// Наполняем НД даныыми:
  BdpDataAdapter1.Fill(DataSet1);
// Создаем экземпляр Binding для связи свойства Text с полем BPrice:
  bdnPrice := Binding.Create('Text', DataSet1.Tables[0], 'BPrice');
// Присоединяем к нему два обработчика событий:
  Include(bdnPrice.Format, DoubleFormat);
  Include(bdnPrice.Parse, DoubleParse);
// Очищаем свойство DataBindings поля:
  TextBox1.DataBindings.Clear;
// Добавляем созданный объект bdnPrice:
  TextBox1.DataBindings.Add(bdnPrice);
end;

procedure TWinForm.Button2_Click(sender: System.Object;
                                 e: System.EventArgs);
// Щелчок на кнопке "Откатить"
begin
  DataSet1.RejectChanges
end;

procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
// Щелчок на кнопке "Подтвердить"
begin
  BdpDataAdapter1.Update(DataSet1)
end;

procedure TWinForm.DoubleFormat(Sender: TObject;
                                e: ConvertEventArgs);
// Обработчик события Format
begin
  e.Value := FloatToStrF(Double(e.Value), ffNumber, 10,2) + 'p';
end;

procedure TWinForm.DoubleParse(Sender: TObject;
                               e: ConvertEventArgs);
```

```
// Обработчик события Parse
var
  S: String;
begin
  S := e.Value.ToString;
  if pos('p', S) <> 0 then
    Delete(S, pos('p', S), 1);
  e.Value := StrToFloat(S);
end;
```

Обратите внимание: аргумент `e.Value` в обработчиках `Format` и `Parse` имеет тип `Object` и, следовательно, способен принимать любые данные. В обработчике `Format` он вначале имеет данные типа `Double`, но принимает **`String`**, в обработчике `Parse` — наоборот.

Программный доступ к значениям таблиц

Рассмотренные в предыдущих разделах средства визуализации открывают доступ к табличным данным для пользователя, работающего с программой. Однако часто возникает необходимость программного управления данными. Типичным примером является использование транзакций, в рамках которых осуществляется группа действий по изменению состояния БД. Эти действия должны выполняться по принципу «либо все, либо ни одного». Например, перевод денег с одного счета на другой. Для реализации этой операции сначала нужно изменить данные в одной таблице (уменьшить дебиторский счет на заданную сумму), затем в другой (увеличить кредиторский счет на ту же сумму). Если между этими операциями произойдет случайный сбой, кредиторский счет останется без изменения, а деньги с дебиторского счета бесследно исчезнут. Чтобы исключить подобную ситуацию, оба действия выполняются в рамках одной транзакции. Сетка `DataGrid` автоматически стартует новую транзакцию при редактировании данных в записи. Но эта транзакция также автоматически заканчивается (подтверждается), когда курсор данных переходит к другой записи или сетка теряет фокус ввода.

Для старта транзакции методом `BeginTransaction` связного компонента `BdpConnection` создается объект класса `BdpTransaction`, имеющий методы `Commit` (подтвердить транзакцию) и `Rollback` (откатить транзакцию). В промежутке между стартом и завершением транзакции программа (а не пользователь!) должна изменить данные в обеих таблицах.

Общая схема получения данных такова. Объявляется переменная класса `BindingManagerBase`, и ей присваивается ссылка на объект `BindingContext` таблицы. Класс `BindingManagerBase` — абстрактный, а класс `BindingContext` — его потомок, перекрывающий абстрактные свойства своего родителя. В классе `BindingManagerBase` есть два свойства типа `Integer`: `Count` и `Position`. Первое определяет общее количество записей в НД, второе — положение курсора данных. Свойство `Position` доступно как для чтения, так и для записи, что по-

зволяет выбирать нужную запись в НД. Затем в массиве Rows таблицы DataTable выбирается запись и поле в ней:

```
// Связываем переменную Binding класса BindingManagerBase
// с контекстом таблицы:
Binding := BindingContext[DataSet1.Tables[0]];
// Получаем текущую запись:
CurRow := DataSet1.Tables[0].Rows[Binding.Position];
// Получаем значение N-го поля:
Text := CurRow[N, DataRowVersion.Current].ToString;
```

Значение поля возвращается в объекте класса Object. Его явное приведение к нужному типу разрешается компилятором Delphi, но блокируется CLR. При выполнении следующего оператора JIT-компилятор сообщит о недопустимом приведении типов:

```
NCount := CurRow['NCount'] as Integer;
```

Налицо ошибка разработчиков Delphi. Приходится использовать метод ToString объекта данных:

```
NCount.Parse(CurRow['NCount'].ToString);
```

Изменить значение поля можно только с помощью SQL-оператора **UPDATE**, который формируется программой и передается серверу в объекте класса BdpCommand.

Для иллюстрации вышесказанного рассмотрим пример перевода денег с дебиторского счета на кредиторский (рис. 2.31).

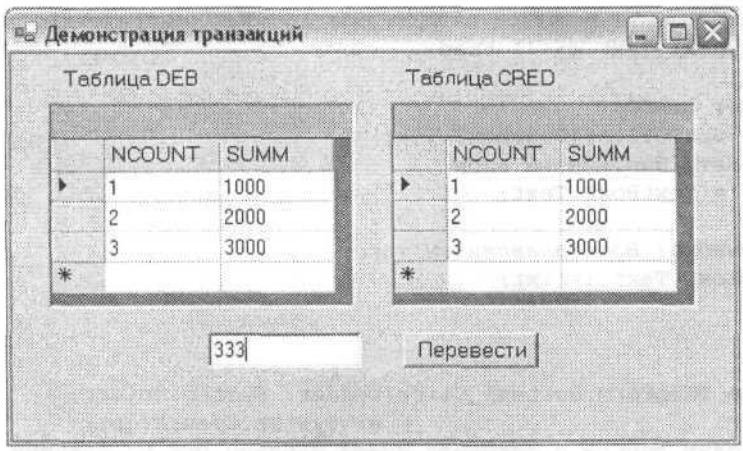


Рис. 2.31. Окно демонстрационной программы

В БД IB_BIBL.GDB есть две небольшие таблицы: DEB и CRED. Начните новое приложение WinForms и поместите на форму компонент BdpConnection, два компонента BdpDataAdapter и компонент DataSet. Настройте компонент BdpConnection на связь с БД IB_BIBL.GDB. Настройте один компонент BdpDataAdapter на отображение всех полей таблицы DEB, а второй — таблицы

CRED. Введите в их свойства DataSet ссылки на компонент DataSet1 и откройте их (задайте True для свойства Active). Поместите на форму две сетки DataGrid и с помощью их свойств DataSet свяжите левую с таблицей DataTable1, а правую — с таблицей DataTable2. Сетки должны наполниться данными.

Поместите на форму поле ввода TextBox и кнопку Button. Напишите следующие обработчики событий: TWinForm_Activated, TextBox1_TextChanged и TButton1_Click:

```
procedure TWinForm.TWinForm_Activated(sender: System.Object;
                                      e: System.EventArgs);
// Передает фокус ввода полю TextBox1 в момент активизации формы
begin
  TextBox1.Focus;
end;

var
  Txt: String = '';  // Глобальная переменная Txt сохраняет
                     // последний правильный ввод в поле TextBox1
procedure TWinForm.TextBox1_TextChanged(sender: System.Object;
                                        e: System.EventArgs);
// Проверяет ввод в поле TextBox1 и отвергает символы, не
// соответствующие правильному текстовому представлению
// вещественных чисел
var
  D: Double;
begin
  if TextBox1.Text <> '' then
  try
    // Нет ошибок?
    D := StrToFloat(TextBox1.Text);
    // -Нет. Запоминаем ввод
    Txt := TextBox1.Text;
  except
    // Ошибка! Восстанавливаем текст
    TextBox1.Text := Txt;
  end;
end;

procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
// Переводит деньги с текущего счета таблицы DEB на текущий
// счет таблицы CRED под защитой транзакции
var
  Comm: BdpCommand;
  Sum, Count: Double;
  Binding: BindingManagerBase;
  Trans: BdpTransaction;
  CommStr: String;
```

```
  NCount: Integer;
  CurRow: DataRow;
begin
  if TextBox1.Text = '' then
    Exit;  // Не введена сумма перевода
  // Проверяем соединение и открываем его, если оно еще не открыто:
  if BdpConnection1.State <> ConnectionState.Open then
    BdpConnection1.Open;
  // Получаем сумму перевода:
  Sum := StrToFloat(TextBox1.Text);
  // Связываем переменную Binding с контекстом таблицы DEB:
  Binding := BindingContext[DataSet1.Tables['DEB']];
  // Получаем текущую запись таблицы DEB:
  CurRow := DataSet1.Tables['DEB'].Rows[Binding.Position];
  // Получаем текущую сумму на счете:
  Count := StrToFloat(CurRow[1, DataRowVersion.Current].ToString);
  // Получаем номер счета:
  NCount := StrToInt(CurRow[0, DataRowVersion.Current].ToString);
  // Проверяем затребованную сумму:
  if Count < Sum then
  begin  // Сумма превышает остаток на счете
    TextBox1.Text := Count.ToString; // Указываем остаток
    TextBox1.Focus;                  // Фокусируем поле ввода
    Exit;                            // Завершаем работу
  end;
  // Определяем новую сумму на счете:
  Count := Count - Sum;
  // Создаем объект для SQL-команды:
  Comm := BdpCommand.Create;
  // Указываем ему соединение:
  Comm.Connection := BdpConnection1;
  // Стартуем транзакцию
  Trans := BdpConnection1.BeginTransaction();
  // Указываем объекту Comm транзакцию:
  Comm.Transaction := Trans;
  Comm.CommandType := CommandType.Text;
  // Формируем текст команды:
  CommStr := 'UPDATE DEB SET SUMM=' + Count.ToString +
    ' WHERE NCOUNT='+ NCount.ToString;
  // Помещаем команду в свойство CommandText:
  Comm.CommandText := CommStr;
  // Получаем контекст таблицы CRED:
  Binding := BindingContext[DataSet1.Tables['CRED']];
  // Текущая запись таблицы:
  CurRow := DataSet1.Tables['CRED'].Rows[Binding.Position];
  // Сумма на счете:
  Count := StrToFloat(CurRow[1, DataRowVersion.Current].ToString);
  // Номер счета:
  NCount := StrToInt(CurRow[0, DataRowVersion.Current].ToString);
  Count := Count + Sum;
```

```
  CommStr := 'UPDATE CRED SET SUMM=' + Count.ToString +
    ' WHERE NCOUNT='+ NCount.ToString;
  try
    Comm.ExecuteNonQuery;          // Снимаем с дебиторского счета
    Comm.CommandText := CommStr; // Новая команда
    Comm.ExecuteNonQuery;          // Начисляем на кредиторский счет
    Trans.Commit;                  // Подтверждаем транзакцию
  except                           // Ошибка!
    Trans.Rollback;                // Откатываем транзакцию
  end;
                                   // Обновляем данные в сетках:
  BdpDataAdapter1.Active := False;
  BdpDataAdapter1.Active := True;
  BdpDataAdapter2.Active := False;
  BdpDataAdapter2.Active := True;
                                   // Очищаем поле ввода:
  TextBox1.Text := '';
  TextBox1.Focus;                  // Передаем ему фокус ввода
end;
```

При подготовке примера я не смог получить нужный результат с помощью такого преобразования:

```
Count.Parse(CurRow[1, DataRowVersion.Current].ToString);
```

В моем случае переменная Count всегда получала нулевое значение, хотя строковое представление суммы было правильным. Пришлось воспользоваться функциями StrToFloat и StrToInt, которые определены в модуле Borland.VCL.SysUtils. Не забудьте вставить ссылку на этот модуль в начале исполняемой части:

```
implementation
uses Borland.VCL.SysUtils;
```

3

Технология BDE.NET

В этой и последующих главах описываются технологии работы с БД, которые поддерживались в предыдущих версиях Delphi и перенесены в версию 2005. Все они называются *XXX*.NET, что говорит об их способности работать на платформе .NET. Доступ к технологиям возможен только в режиме создания VCL-приложений.

Суть технологии

Технология получила свое название по первым буквам словосочетания Borland Database Engine — машина баз данных корпорации Borland. BDE представляет собой набор библиотек DLL, реализующих различные службы, связанные с обслуживанием БД.

Рисунок 3.1 иллюстрирует связь составных частей технологии BDE.NET.

К *компонентам связи* относятся компоненты `TSession` и `TDatabase`. Первый создает канал связи, второй передает серверу дополнительные параметры (имя пользователя и пароль), а также обеспечивает функционирование механизма транзакций. Любая программа, использующая технологию BDE.NET, автоматически (по умолчанию) получает один канал связи `TSession`, что вполне достаточно для большинства типичных приложений.

К *наборам данных* относятся компоненты `TTable`, `TQuery` и `TStoredProc`. Первый открывает доступ к одной из таблиц БД или к виртуальной таблице — представлению (*view*). Второй создает набор данных за счет SQL-запроса, а третий — за счет обращения к хранимой процедуре. Замечу, что BDE имеет собственный интерпретатор SQL, что позволяет использовать `TQuery` не только для обращения к серверным БД, но и для работы с таблицами файл-серверных баз данных.

Источником данных является компонент `TDataSourse`. Он необходим только для связи НД с компонентами визуализации. Фактически он является

Рис. 3.1. Иллюстрация технологии BDE.NET

управляемым переключателем, подключающим визуальные компоненты к НД или отключающим их от него.

В технологии BDE.NET используется множество специальных компонентов для *визуализации данных*, в том числе сетка `TDBGrid`.

Поля

Каждый набор данных в таблице базы данных (ТБД) состоит из записей, а те, в свою очередь, состоят из полей. Таким образом, в структуре таблицы имеется минимум одно поле. Обращаться к полям таблиц баз данных позволяет класс `TField`, работа с которым и является темой этого раздела.

Обзор свойств, методов и событий

В этом разделе рассматриваются некоторые наиболее важные свойства, методы и события класса `TField`.

Свойства

Важнейшие свойства класса `TField` представлены в табл. 3.1.

Таблица 3.1. Некоторые свойства класса TField

Свойство	Описание
`property Alignment: TAlignment;`	Определяет выравнивание значения поля при его отображении: `taLeftJustify` — выровнять влево; `taRightJustify` — выровнять вправо; `taCenter` — выровнять по центру
`property As`*XXXX*`: T`*XXXX*`;`	Группа свойств этого вида приводит значение поля к нужному типу. Вместо символов *XXXX* в названии свойства и типа могут использоваться: `BCD` — приведение к типу двоично-десятичных данных; `Boolean` — к логическому типу; `Currency` — к денежному типу; `DataTime` — к типу дата-время; `Float` — к типу `Double`; `Ineteger` — к типу `Integer`; `String` — к типу `String`; `Variant` — к типу `Variant`

Свойство	Описание
`property AttributSet: String;`	Содержит имя набора атрибутов поля в словаре данных (см. далее примечание)
`property Calculated: Boolean;`	Содержит `True`, если значение поля вычисляется в обработчике события `OnCalcFields`
`property CanModify: Boolean;`	Содержит `True`, если значение поля можно изменять
`property CurValue: Variant;`	Содержит текущее значение с учетом возможных изменений, внесенных другими пользователями
`property DataSet: TDataSet;`	Связанный с полем набор данных (наследник класса `TDataSet` типа `TDBDataSet`)
`property DataSize: Word;`	Содержит размер данных
`property DataType: TFieldType;`	Тип `TFieldType` представляет собой перечисление, определяющее тип данных (см. далее)
`property DefaultExpression: String;`	Содержит SQL-выражение, определяющее значение по умолчанию (то есть значение, которое оказывается в поле, если пользователь не вводит никакого значения)
`property DisplayLabel: String;`	Содержит заголовок поля в сетке `TDBGrid`
`property DisplayText: String;`	Содержит строковое представление значения поля. Это свойство только для чтения. В обработчике события `OnGetText` программист может изменить его
`property DisplayWidth: Integer;`	Определяет ширину поля в символах
`property EditMask: String;`	Определяет маску, которая будет использоваться для фильтрации и форматирования вводимого значения (см. далее)
`type TFieldKind = (fkData, fkCalculated, fkLookup, fkInternalCalc, fkAggregate);` `property FieldKind: TFieldKind;`	Определяет тип поля: `fkData` — обычное поле; `fkCalculated` — вычисляемое поле; `fkLookup` — подстановочное (lookup) поле; `fkInternalCalc` — внешнее вычисляемое поле (см. далее); `fkAggregate` — агрегатное поле
`property FieldName: String;`	Имя поля в соответствующем НД
`property FieldNo: Integer;`	Номер поля в соответствующем НД. Нумерация полей начинается с 1 (только для чтения)
`property HasConstraints: Boolean;`	Содержит `True`, если на значение поля наложены ограничения. Не учитывает ограничения, вводимые маской `EditMask`
`property Index: Integer;`	Индекс поля. Изменение индекса приводит к изменению порядка отображения поля в сетке, но не в НД

продолжение

Таблица 3.1 (продолжение)

Свойство	Описание
`property IsIndexField: Boolean;`	Содержит `True`, если поле — индексное
`property IsNull: Boolean;`	Возвращает `True`, если значение поля есть `Null`
`property KeyFields: String;`	Список индексных полей текущего НД для связи с подстановочным НД
`property Lookup: Boolean;`	Содержит `True`, если тип поля — `ftLookup`
`property LookupCashe: Boolean;`	Разрешает/запрещает кэширование значений подстановочных полей
`property LookupDataSet: TDataSet;`	Определяет НД, в котором ищутся значения полей `LookupKeyFields`
`property LookupKeyFields: String;`	Список полей в подстановочном НД, которые должны соответствовать полям `KeyFields`
`property LookupList: TLookupList;`	Список всех допустимых значений подстановочного поля
`property LookupResultFild: String;`	Текущее значение подстановочного поля
`property NewValue: Variant;`	Новое значение поля (до завершения транзакции)
`property OldValue: Variant;`	Старое значение поля
`property Origin: String;`	Содержит имя поля в физической ТБД
`property ParentField: TObjectField;`	Указывает родительское поле во вложенном поле. Содержит **NIL**, если текущее поле не является частью сложного поля

`type TProviderFlag = (pfInUpdate, pfInWhere, pfInKey, pfHidden);`
`TProviderFlags = set of TProviderFlag;`
`property ProviderFlags: TProviderFlags;`
	Указывает, как использовать это поле при пересылке серверу пакета записей для обновления: `pfInUpdate` — поле появляется в предложении `UPDATE` и, следовательно, может обновляться; `pfInWhere` — поле появляется в предложении **WHERE** и, следовательно, используется для поиска обновленных записей; `pfInKey` — поле используется для поиска записей в случае возникновения ошибки обновления; `pfHidden` — поле включено в пакет для обеспечения уникальности записей и не показывается пользователю

`property ReadOnly: Boolean;`	Содержит `True`, если поле предназначено только для чтения
`property Required: Boolean;`	Если содержит `True`, значение каждой записи поля не может быть пустым
`property Size: Word;`	Текущая длина записи в полях с переменной длиной записи

Свойство	Описание
`property Text: String;`	Текст, которым отображается значение записи в режиме редактирования. В остальных режимах запись отображается содержимым `DisplayText`
`property Value: Variant;`	Содержит текущее значение поля
`property Visible: Boolean;`	Разрешает/запрещает показ поля в визуальных компонентах

ПРИМЕЧАНИЕ

Набор атрибутов поля в словаре данных определяет вид, в котором данные отображаются в визуальных компонентах на этапе разработки программы, а также ограничения и значения по умолчанию. Каждый набор имеет имя, определяемое свойством `AttributeSet`, что позволяет использовать его в других полях.

ПРИМЕЧАНИЕ

Ограничения, определяемые свойством `CustomConstraint`, записываются строками вида `"X>0 and X<100"`.

Параметром свойства `DataType` является следующее перечисление:

```
type TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger,
                   ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD,
                   ftDate, ftTime, ftDateTime, ftBytes,
                   ftVarBytes, ftAutoInc, ftBlob, ftMemo,
                   ftGraphic, ftFmtMemo, ftParadoxOle,
                   ftDBaseOle, ftTypedBinary, ftCursor,
                   ftFixedChar, ftWideString, ftLargeint, ftADT,
                   ftArray, ftReference, ftDataSet, ftOraBlob,
                   ftOraClob, ftVariant, ftInterface,
                   ftIDispatch, ftGuid, ftTimeStamp, ftFMTBcd);
```

Значения перечисления `TFieldType` описаны ниже:

- `ftUnknown` — неизвестный тип;
- `ftString` — строковый тип;
- `ftSmallint` — тип `SmallInt`;
- `ftInteger` — тип `Integer`;
- `ftWord` — тип `Word`;
- `ftBoolean` — логический тип;
- `ftFloat` — вещественный тип;
- `ftCurrency` — денежный тип;
- `ftBCD` — двоично-десятичные данные;
- `ftDate` — дата;
- `ftTime` — время;
- `ftDateTime` — дата-время;

- `ftBytes` — строка байтов фиксированной длины;
- `ftVarBytes` — строка байтов переменной длины;
- `ftAutoInc` — автоинкрементное поле;
- `ftBlob` — двоичное поле переменной длины;
- `ftMemo` — многострочный текст;
- `ftGraphic` — графическое изображение;
- `ftFmtMemo` — форматированный текст;
- `ftParadoxOle` — поле ParadoxOle;
- `ftDBaseOle` — поле DBaseOle;
- `ftTypedBinary` — типизированное двоичное поле;
- `ftCursor` — состояние курсора НД для сохраняемой процедуры Oracle;
- `ftFixedChar` — строка символов фиксированной длины;
- `ftWideString` — строка двухбайтных символов;
- `ftLargeInt` — длинное целое значение;
- `ftADT` — данные абстрактного типа;
- `ftArray` — массив;
- `ftReference` — ссылочный тип;
- `ftDataSet` — набор данных;
- `ftOraBlob` — поле BLOB в таблице Oracle 8;
- `ftOraClob` — поле CLOB в таблице Oracle 8;
- `ftVariant` — поле содержит значение типа вариант;
- `ftInterface` — ссылка на интерфейс `IUnknown`;
- `ftIDispatch` — ссылка на интерфейс `IDispatch`;
- `ftGuid` — глобально-уникальный идентификатор;
- `ftTimeStamp` — дата и время, доступные в технологии dbExpress;
- `ftFMiTBcd` — двоично-десятичные данные, которые слишком велики для `tlBCD`.

ПРИМЕЧАНИЕ

Типы полей `ftCursor`, `ftFixedChar`, `ftWideChar`, `ftLargeInt`, `ftADT`, `ftArray`, `ftReference`, `ftOraBlob` и `ftOraClob` введены для поддержки промышленного сервера БД Oracle версии 8 и выше.

Маска, которую определяет свойство `EditMask`, состоит из трех частей, отделенных друг от друга символом точки с запятой (`;`). Первая часть задает собственно маску ввода, вторая — это символ 0 или 1, определяющий, включается ли в форматируемое значение результат наложения маски на исходный текст или только исходный текст (0 — исходный текст). В третьей части указывается символ, который должен присутствовать в местах ввода символов в поле.

Таким образом, текст в поле может содержать символы маски; например, для ввода семизначного номера телефона текст перед началом ввода может выглядеть

так (доступные для ввода пользователя места обозначены символом X — последним символом в шаблоне):

```
(095)XXX-XX-XX
```

Маска состоит из описателей мест ввода, специальных символов и литералов. Описатель указывает, какой именно символ может ввести пользователь в данном месте (табл. 3.2). Литерал вставляется в текст, отображаемый в поле ввода, но при вводе курсор «перескакивает» через литерал и не дает пользователю возможности изменить его. Литералом считается любой символ, если ему не предшествует символ обратной косой черты (\), если он не является описателем места ввода или специальным символом. С помощью специальных символов формируются дополнительные указания по обработке вводимых данных (табл. 3.3).

Таблица 3.2. Описатели мест ввода

Символ	Описание места ввода
L	Должно содержать букву
l	Может содержать букву
A	Должно содержать букву или цифру
a	Может содержать букву или цифру
C	Должно содержать любой символ
c	Может содержать любой символ
0	Должно содержать цифру
9	Может содержать цифру
#	Может содержать цифру, а также знаки плюс (+) и минус (–)

Таблица 3.3. Специальные символы

Символ	Описание
\	Следующий символ — литерал. Позволяет вставлять в маску литералы из числа символов, относящихся к описателям мест ввода и специальным символам
:	На это место вставляется символ-разделитель Windows для часов, минут, секунд
/	На это место вставляется символ-разделитель Windows для полей даты
;	Разделитель частей шаблона
!	Подавляет вывод всех начальных пробелов
>	Во всех следующих за этим символом местах ввода буквы преобразуются в прописные
<	Во всех следующих за этим символом местах ввода буквы преобразуются в строчные
<>	Отменяет преобразование регистра букв

Примеры использования маски ввода представлены в табл. 3.4.

Таблица 3.4. Использование маски ввода

Маска	Вид в поле ввода	Ввод пользователя	Результат
`(095)000-0000;0;x`	`(095)xxx-xxxx`	`1234567`	`1234567`
`(095)000-0000;1;x`	`(095)xxx-xxxx`	`1234567`	`(095)123-4567`
`(095)\0\00-0000;1;`	`(095)00.-`	`12345`	`(095)001-2345`

Методы

Наиболее важные методы класса `TField` перечислены в табл. 3.5.

Таблица 3.5. Некоторые методы класса TField

Метод	Описание
`procedure Assign(Source: MarshalByRefObject);`	Копирует значение поля из параметра `Source` в собственное свойство `Value`. Типы полей должны быть совместимы
`procedure AssignValue(const Value: TVarRec);`	Преобразует вариант `Value` с помощью метода `As`*XXXX* и помещает результат в собственное свойство `Value`
`procedure Clear;`	Очищает значение поля и помещает в него `Null` (в терминологии SQL)
`constructor Create(AOwner: Component);`	Создает и инициализирует объект-поле
`destructor Destroy;`	Удаляет объект-поле
`function HasParent;`	Возвращает `True`, если поле — вложенное
`function IsValidChar(InputChar: Char): Boolean; virtual;`	Возвращает `True`, если символ `InputChar` может присутствовать в текстовом представлении значения поля
`proxedure RefreshLookupList;`	Заново создает список значений подстановочного поля

События

События класса `TField` представлены в табл. 3.6.

Таблицы 3.6. События класса TField

Событие	Описание
`property OnChange: TFieldNotifyEvent;`	Возникает после изменения данных и успешной записи их в буфер

Событие	Описание
`type TFieldGetTextEvent = procedure(Sender: TField; var Text: String; DisplayText: Boolean) of object;`	Обработчик этого события готовит текст для свойств `DisplayText` и `Text`. Обработчик по умолчанию помещает в переменную `Text` значение `Value.AsString`. Параметр `DisplayText` имеет значение `True`, если текст предназначен для свойства `DisplayText`
`property OnGetText: TFieldGetTextEvent;`	
`property OnSetText: TFieldSetTextEvent;`	Возникает при записи данных из параметра `Text` в свойство `Text`
`property OnValidate: TFieldNotifyEvent;`	Возникает после изменения значения до записи в буфер. Обработчик этого события должен проверить правильность введенных данных

Использование объектов-полей

Объекты для реальных полей

Для удобства работы с конкретным полем можно создать объект-поле класса `TField` или одного из его специализированных потомков — `TStringField`, `TIntegerField`, `TBlobField` и т. д. Объект-поле создается на этапе разработки программы с помощью *редактора полей*. Если определен объект-поле, получить доступ к полю можно по имени этого объекта. Редактор полей присваивает объектам-полям имена путем сцепления имени источника данных и имени поля. Если, например, источник `Firms` связан с НД, у которого определен объект для поля `City`, объекту будет присвоено имя `FirmsCity`, после чего к нему можно обращаться напрямую. Например:

```
FirmsCity.AsString := 'Москва';
```

СОВЕТ

Поскольку на практике создавать объекты-поля приходится довольно часто, имеет смысл называть компоненты-наборы именами связанных с ними таблиц. Например, назвав объект `TTable` именем связанной с ним таблицы `Books`, автоматически получим имена типа `BooksAuthor`, `BooksYear`, `BooksPages` и т. п. Если к одной физической таблице обращаются разные компоненты-наборы, их имена можно дополнять *префиксами*: `tbBooks`, `quBooks`, `spBooks` — для таблицы, запроса и хранимой процедуры соответственно.

Определить во время выполнения приложения, используются ли для набора данных все умалчиваемые поля или только их часть, можно при помощи следующего свойства НД:

```
property DefaultFields: Boolean;
```

Значение `True` указывает, что используются поля, заданные по умолчанию; `False` — поля, определенные при помощи редактора полей.

Если хотя бы для одного поля НД создан объект-поле, все поля НД, для которых такие объекты не определены, становятся недоступными. К «несуществующим» полям обратиться из данного НД нельзя. Вновь вернуться к использованию всех полей НД можно только на этапе разработки программы, удалив в редакторе полей все определенные ранее объекты или добавив с его помощью объекты для недостающих полей.

ПРИМЕЧАНИЕ

Если нужно иметь доступ к полю, но не показывать его значений в компонентах, визуализирующих данные (например, в компоненте `TDBGrid`), в свойство `Visible` этого объекта-поля следует поместить значение `False`.

Для вызова редактора полей нужно дважды щелкнуть на компоненте-наборе или щелкнуть на нем правой кнопкой мыши и выбрать в контекстном меню команду Fields Editor. Предварительно НД должен быть связан с нужной таблицей БД: в `TTable` должны быть определены свойства `DatabaseName` и `TableName`, в `TQuery` — `DatabaseName` и `SQL` (то есть это свойство должно содержать текст SQL-запроса, который создаст НД), в `TStoredProc` — `DatadaseName` и `StoredProcName`.

Чтобы добавить объекты-поля, щелкните в окне редактора правой кнопкой мыши и выберите в контекстном меню команду Add Fields. В списке открывшегося окна Add Fields будут указаны поля, для которых еще не созданы объекты (рис. 3.2, *слева*). Вы можете выбрать любую комбинацию полей в списке и щелкнуть на кнопке OK — для выделенных полей будут созданы объекты, а окно редактора будет содержать список этих полей. Если нужно создать объекты-поля для всех полей НД, выберите команду Add All Fields в контекстном меню редактора. Чтобы удалить определение какого-либо объекта, нужно выделить соответствующее поле в окне редактора и нажать клавишу Delete. Если нужно удалить объявления всех объектов-полей, выберите команду Select All в контекстном меню и после этого нажмите клавишу Delete.

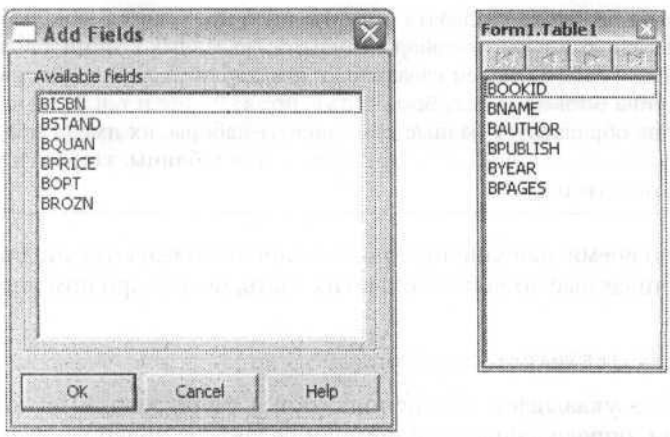


Рис. 3.2. Работа с редактором полей: слева — добавление полей; справа — список полей, для которых созданы объекты

Чтобы изменить свойства конкретного объекта-поля или написать обработчик для какого-либо связанного с ним события, необходимо в редакторе полей выделить нужное поле и, используя инспектор объектов, установить значение в свойство или определить обработчик события.

Объекты для подстановочных полей

Подстановочное (lookup) поле отличается от обычного объекта-поля тем, что содержит данные из другого НД. Непременным условием создания подстановочного поля является выполнение требования реляционной связи главного НД с таблицей подстановки: в главном НД каждое значение какого-либо поля должно ссылаться на одну из записей в таблице подстановки, причем по полю связи в этой таблице должен быть создан уникальный индекс или первичный ключ.

Для создания подстановочного поля нужно вызвать контекстное меню редактора полей (щелкнув в его окне правой кнопкой мыши) и выбрать в нем команду New Field. В появляющемся после этого окне добавления нового поля (рис. 3.3) нужно ввести имя поля в строке Name и выбрать в списке Type его тип. Строка Size заполняется только для текстовых полей и полей типа набора байтов. В первом случае оно показывает максимальную длину в символах строки, которую будет содержать поле (по умолчанию этот размер равен 20 символам, что может быть недостаточным во многих практических случаях, поэтому следует проанализировать структуру таблицы подстановки и уточнить размер того поля, которое и будет содержать информацию для создаваемого объекта), во втором — длину поля в байтах. Для всех других типов значение в строке Size игнорируется. Строка Component в момент ввода имени заполняется автоматически (она содержит имя объекта-поля). Delphi игнорирует «чужой» ввод в строку Component.

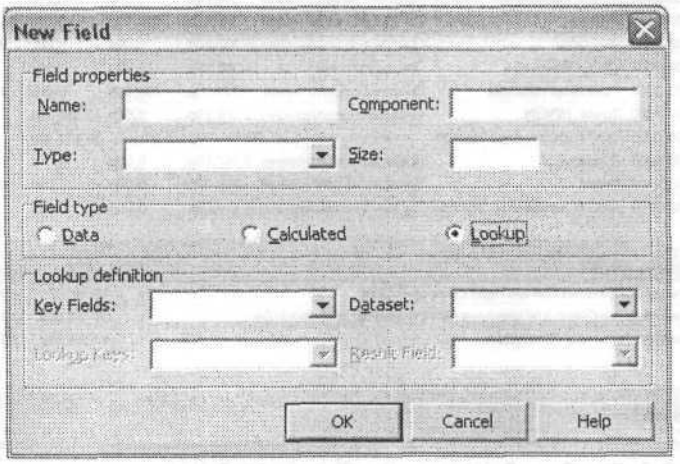


Рис. 3.3. Окно создания нового поля

В группе Field type следует установить переключатель Lookup, затем раскрыть список полей главной таблицы Key Fields и выбрать в нем ссылочное поле. В списке Dataset выбирается подстановочный НД, в списке Lookup Keys — индексное поле подстановочного НД, а в списке Result Field — результирующее поле (именно его значения присоединятся к полям главной таблицы).

Достаточно подробный пример создания подстановочных полей вы найдете в главе 1.

Объекты для вычисляемых полей

Вычисляемые поля не входят в состав конкретной таблицы БД, но могут быть присоединены к НД, связанному с этой таблицей. Они предназначены для отображения данных, которые вычисляются в ходе выполнения программы обычно с помощью значений из других полей той же записи. После присоединения вычисляемого поля к НД оно становится во всем подобно обычным объектам-полям, связанным с реальными полями таблицы БД, за одним исключением: такие поля нельзя редактировать, а при вводе новой записи — помещать в них какое-либо значение.

Для создания вычисляемого поля нужно открыть окно New Field (см. рис. 3.3) редактора полей и, заполнив строки Name, Type и Size, установить переключатель Calculated и закрыть окно щелчком на кнопке OK. Заполнение вычисляемых полей осуществляется в обработчике события `OnCalcFields` набора данных.

Проиллюстрируем создание вычисляемого поля на примере программы, описанной в главе 1. В ней используется таблица `MOVES` (НД `Move`) для отображения списка книг, связанных с накладной. В этой таблице есть поля, в которых указываются количество экземпляров книги (объект `MoveMQuan`) и ее цена (`MoveMPrice`). Создадим для НД `Move` вычисляемое поле `Summa`, в котором укажем сумму покупки (продажи) каждой книги (рис. 3.4).

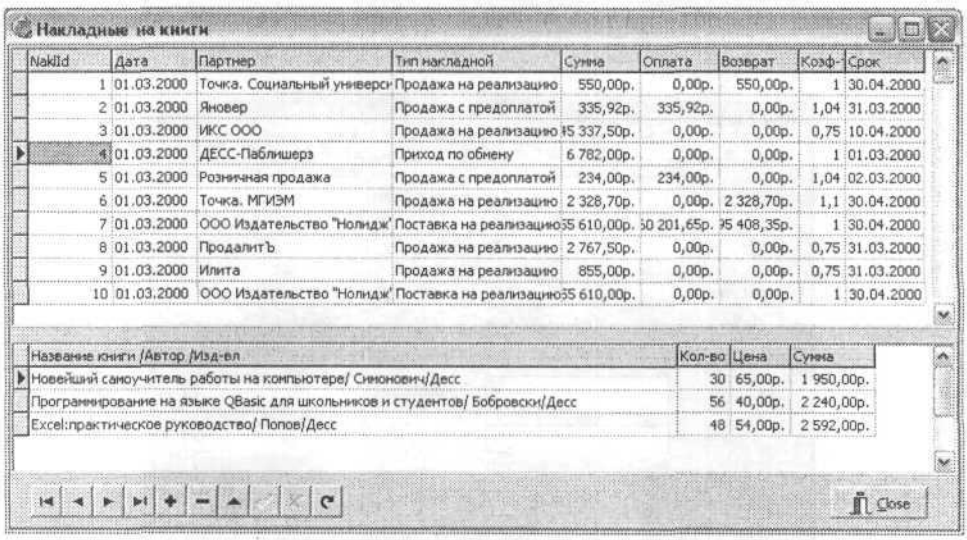


Рис. 3.4. Столбец «Сумма» в нижней сетке отображает вычисляемое поле

Вызовите редактор полей двойным щелчком на НД Move и затем в контекстном меню редактора выберите команду New Field. Введите в строку Name окна New Field название поля Summa, в списке Type выберите денежный тип (Currency), установите переключатель Calculated и закройте окно щелчком на кнопке OK. Если сейчас вызвать инспектор объектов, он будет отображать свойства и события вновь созданного объекта-поля MoveSumma, поэтому щелкните один раз на НД Move, чтобы инспектор объектов показал свойства этого НД. Напишите такой обработчик события OnCalcFields (проект Ch03\CalcField\CalcField.dpr):

```
procedure TDM.MoveCalcFields(DataSet: TDataSet);
begin
  MoveSumma.Value := MoveMQuan.Value * MoveMPrice.Value
end;
```

Напомню, что для сетки DBGrid2 были созданы объекты-столбцы. Чтобы увидеть в сетке столбец со значениями нового поля, создайте для него новый объект-столбец.

Объекты для пустых полей

Если в окне New Field (см. рис. 3.3) установить переключатель Data, будет создан объект-поле, не содержащий никаких данных (пустое поле). Как правило, такое поле создается для того, чтобы в сетке DBGrid появился дополнительный пустой столбец, в котором программа сможет отображать произвольные значения (для этого обычно используется обработчик события OnGetText объекта-поля или события OnDrawColumnCell сетки). Замечу, что на практике удобнее вместо пустых полей использовать пустые объекты-столбцы, создаваемые непосредственно в сетке DBGrid.

Обращение к значению поля

К значению поля можно обратиться при помощи свойств Value и As*XXXX* класса TField или его потомков. В табл. 3.7 перечислены типы значений, возвращаемых следующим свойством:

```
property Value: Variant
```

Таблица 3.7. Типы значений, возвращаемых компонентами в свойстве Value

Тип	Компонент
Variant	Все компоненты
String	TStringField, TBlobField
LongInt	TAutoIncField, TIntegerField, TSmallintField, TWordField
Double	TBCDField, TCurrencyField, TFloatField
Boolean	TBooleanField
TDateTime	TDateField, TDateTimeField, TTimeField

В классе TField (и во всех специализированных классах T*XXXX*Field) для приведения типов полей определены следующие свойства As*XXXX*:

```
property AsBCD: TBcd;
property AsBoolean: Boolean;
property AsCurrency: Currency;
property AsDateTime: TDateTime;
property AsFloat: Double;
property AsInteger: Integer;
property AsString: String;
property AsVariant: Variant;
property AsSQLTimeStamp: TSQLTimeStamp;
```

Каждое из этих свойств приводит значение поля к соответствующему типу данных, указанному в названии свойства. Например, если BooksBookID — поле TAutoIncField, для приведения его к типу **String** нужно воспользоваться свойством AsString:

```
Edit1.Text := BooksBookID.AsString;
```

Разумеется, тип поля должен быть совместимым с типом данных, к которому приводится значение поля. Например, если MoveSumma — поле типа TCurrencyField, попытка привести его к несовместимому типу Boolean приведет к ошибке.

Таблица 3.8 иллюстрирует совместимость значений полей разных типов, при этом используются следующие обозначения:

- = — типы равнозначны;
- + — преобразование возможно;
- + RI — преобразование возможно, при этом выполняется округление до ближайшего целого;
- ? — преобразование происходит, если оно возможно, и часто зависит от формата отображения (свойство DisplayFormat);
- x — преобразование не разрешено;
- memo — имеет значение для мемо-поля;
- п. 1 — преобразование даты к числу дней, прошедших с 01.01.0001;
- п. 2 — преобразование времени делением на 24 часа;
- п. 3 — преобразование в строку True или False.

Таблица 3.8. Совместимость значений полей разных типов

Тип поля	AsString	AsInteger	AsFloat	AsDateTime	AsBoolean
TStringField	=	?	?	?	?
TIntegerField	+	=	+	x	x
TSmallinTField	+	=	+	x	x
TWordField	+	=	+	x	x
TFloatField	+	+ RI	=	x	x
TCurrencyField	+	+ RI	=	x	x

Тип поля	AsString	AsInteger	AsFloat	AsDateTime	AsBoolean
TBCDField	+	+ RI	=	x	x
TDateTimeField	+	x	п. 1	=	x
TDateField	?	x	п. 1	=	x
TTimeField	?	x	п. 2	=	x
TBooleanField	п. 3	x	x	x	=
TBytesField	+	x	x	x	x
TVarBytesField	+	x	x	x	x
TBlobField	memo	x	x	x	x
TMemoField	memo	x	x	x	x
TGraphicField	memo	x	x	x	x

Если для НД не создано ни одного объекта-поля, получить доступ к значению поля этого НД можно с помощью его метода FieldByName:

```
function FieldByName(const FieldName: String): TField
```

Кроме того, получить доступ к значению поля НД можно через его свойства Fields и FieldValues:

```
property Fields[Index]: TField
property FieldValues['Имя_поля']: Variant
```

Обратите внимание: нумерация полей в свойстве Fields начинается с 0, а свойство FieldValues является умалчиваемым свойством для НД, поэтому следующие обращения идентичны (предполагается, что поле 'Year' является 5-м по счету в НД Books):

```
Books.FieldByName('Year').AsInteger := 2000;
Books['Year'] := 2000;
Books.FieldValues['Year'] := 2000;
Books.Fields[4].AsInteger := 2000;
```

Свойство FieldValues НД обладает двумя особенностями: во-первых, оно имеет значение вариантного типа, во-вторых, допускает обращение со списком полей. Это позволяет единственным оператором обращения к этому свойству прочитать или записать сразу несколько полей текущей записи. Такую возможность иллюстрирует листинг 3.1.

Листинг 3.1. Чтение и запись сразу нескольких полей

```
procedure TForm1.Button1Click(Sender: TObject);
var
  N: Integer;
  Price: Currency;
  V: Variant;
begin
  // Читаем поля 1-й записи НД Books:
  V := VarArrayCreate([0, 2], varVariant);
```

продолжение

Листинг 3.1 (продолжение)

```
  V := Books['BQuan;BName;BOpt'];
  N := V[0];
  Price := V[2];
  ShowMessage(Format(
        'В продаже имеется %d экз. книги "%s" по цене %fp.',
        [N, V[1], Price]));
  // Изменяем поля:
  V[0] := 5;
  V[1] := 'Моя книга';
  V[2] := 40.0;
  Books.Edit;
  Books['BQuan;BName;BOpt'] := V;
  Books.Post
end;
```

Проверка правильности введенного в поле значения

Для контроля правильности вводимых в поле значений можно использовать событие `OnValidate` или `OnSetText` объекта-поля. Оба события наступают после изменения значения поля, но до его запоминания в таблице.

В обработчике `OnValidate` в случае обнаружения неверного значения программа должна предотвратить его запоминание, создав исключительную ситуацию или обратившись к глобальной процедуре `Abort`. Например, поле `NaklsNPayedSum` в учебной БД содержит данные по оплате накладной и, естественно, не может иметь отрицательных значений. Для контроля над этим можно написать такой обработчик:

```
procedure TDM.NaklsNPayedSumValidate(Sender: TField);
begin
  if NaklsNPayedSum.Value<0 then
  begin
    ShowMessage('Сумма не может быть отрицательной!');
    Abort
  end
end;
```

Однако следует учесть, что, отказавшись от запоминания неверного значения в таблице, программа тем не менее оставляет НД в состоянии редактирования. Это означает, что пользователь не сможет убрать курсор из этого поля до тех пор, пока не введет в него правильное значение. Если бы мы решили защитить поле от любого изменения с помощью показанного ниже обработчика, то обнаружили бы, что при попытке изменить значение поля программа попросту «зацикливается», не давая пользователю убрать курсор из редактируемого поля:

```
procedure TDM.NaklsNPayedSumValidate(Sender: TField);
begin
  ShowMessage('Сумма не должна изменяться!');
  Abort
end;
```

В данном случае использовать событие OnSetText значительно разумнее, так как в нем можно просто игнорировать неверные значения и не создавать исключительную ситуацию. Дело в том, что в этом обработчике программа должна присвоить полю новое значение или оставить поле без изменения:

```
procedure TDM.NaklsNPayedSumSetText(Sender: TField;
                                    const Text: String);
begin
  if StrToFloat(Text)<0 then
    ShowMessage('Сумма не может быть отрицательной!')
  else
    NaklsNPayedSum.Value := StrToFloat(Text)
end;
```

Обратите внимание: можно не бояться того, что при преобразовании текста в значение с плавающей запятой может возникнуть исключение. При вводе значения в поле какого-то типа, кроме символьного, программа автоматически отвергает любые символы, не соответствующие символьному представлению этого типа.

Формирование текстового представления поля

Часто требуется некоторое дополнительное преобразование для более точного или более правильного представления хранящегося в поле значения. Для этих целей служит обработчик события OnGetText объекта-поля.

Например, пусть все денежные данные в БД хранятся в «старом» масштабе цен (с тремя лишними нулями). После деноминации нужно либо пересчитать все денежные поля во всех таблицах (это не всегда возможно, так как на практике при создании СУБД в 90-х годах таким полям часто присваивался целочисленный тип), либо — что значительно проще — изменить представление значения в обработчике события OnGetText. В следующем обработчике не только устраняются лишние нули, но и вставляются разделители тысяч для удобства чтения длинных сумм:

```
procedure TForm1.BooksPriceGetText(Sender: TField;
                                   var Text: String;
                                   DisplayText: Boolean);
begin
  Text := FloatToStrF(tbBooksPrice.AsFloat/1000, ffNumber, 10, 2)
end;
```

Другой пример. Поле Sex в таблице Clients имеет логический тип. Обработчик выводит текст Муж или Жен в зависимости от значения поля:

```
procedure TForm1.tbClientsSexGetText(Sender: TField;
                                     var Text: String;
                                     DisplayText: Boolean);
begin
  if tbClientsSex.Value then
    Text := 'Муж' else
    Text := 'Жен'
end;
```

Замечу, что обработчики выполняются только в работающей программе, поэтому на этапе разработки вы не сможете увидеть результат их работы.

Обзор полей TxxxField

Строковые поля

Поле `TStringField` предназначено для хранения строк длиной до 255 символов. Следующее его свойство указывает, следует ли производить преобразование символов в кодировку ANSI при чтении строк из таблицы и записи их в нее в том случае, когда кодировка строк в таблице отличается от кодировки ANSI:

```
property Transliterate: Boolean
```

Когда в свойство установлено значение `True`, для преобразования автоматически используются функции `AnsiToNative` при записи строк в таблицу и `NativeToAnsi` при чтении из нее. Для русифицированной версии Windows эти функции позволяют работать с таблицей, в строковых полях которой применяется так называемая альтернативная кодировка символов (кодовая страница 866).

Для работы с полем `TStringField` предназначены компоненты `TDBLabel` и `TDBEdit`.

Поле `TMemoField` позволяет хранить множество строк неопределенной длины. Оно имеет следующее свойство, которое содержит размер поля в байтах:

```
property BlobSize: Integer;
```

Специфические методы поля `TMemoField` перечислены в табл. 3.9.

Таблица 3.9. Методы поля TMemoField

Метод	Описание
`procedure Clear;`	Очищает поле
`procedure LoadFromFile(const FileName: String);`	Загружает содержимое поля из файла
`procedure LoadFromStream (Stream: TStream);`	Загружает содержимое поля из потока
`procedure SaveToFile(const FileName: String);`	Сохраняет содержимое поля в файле
`procedure SaveToStream(Stream: TStream);`	Сохраняет содержимое поля в потоке

Для работы с мемо-полями используются компоненты `TDBMemo` и `TDBRichEdit`.

Целочисленные поля

Целочисленные поля применяются для хранения целых чисел различной длины:

- `TIntegerField` — от –2 147 483 648 до +2 147 483 647;
- `TSmallintField` — от –32 768 до +32 767;
- `TWordField` — от 0 до 65 535.

Следующие два свойства определяют максимальное и минимальное значения поля:

```
property MaxValue: Integer
property MinValue: Integer
```

По умолчанию оба свойства содержат 0, что означает отсутствие ограничений. Если `MaxValue > MinValue` и вводимое в поле значение вышло из указанного диапазона, возникает исключительная ситуация.

Следующее свойство осуществляет форматирование данных перед их показом во всех режимах, кроме режима редактирования:

```
property DisplayFormat: String
```

Форматирование для режима редактирования реализует другое свойство:

```
property EditFormat: String
```

Оба свойства унаследованы полями от общего родителя `TNumericField`. Для описания формата в них могут использоваться символы-спецификаторы, перечисленные в табл. 3.10.

Таблица 3.10. Символы-спецификаторы

Спецификатор	Описание
c	Отображает сначала дату в формате дд.мм.гг, затем пробел и время в формате чч.мм.сс: 20.07.03 19:45:00
d	Отображает день месяца без ведущего нуля: 20
dd	Отображает день с ведущим нулем: 02
dddd	Отображает день недели суббота (для нерусифицированной версии Windows — Saturday)
ddddd	Отображает дату в формате дд.мм.гг: 20.07.03
dddddd	Отображает дату в формате день Месяц год: 20 Июль 2003 (для нерусифицированной версии Windows — 20 Jule 2003)
m	Отображает месяц без ведущего нуля: 7
mm	Отображает месяц с ведущим нулем: 07
mmm	Отображает сокращенное название месяца: июл
mmmm	Отображает полное название месяца: Июль
y или yy	Отображает две последние цифры года: 03
yyy или yyyy	Отображает все цифры года: 2003
h	Отображает час без ведущего нуля: 19
hh	Отображает час с ведущим нулем: 09
n	Отображает минуты без ведущего нуля: 45
nn	Отображает минуты с ведущим нулем: 05
s	Отображает секунды без ведущего нуля: 0
ss	Отображает секунды с ведущим нулем: 00

продолжение

Таблица 3.10 (продолжение)

Спецификатор	Описание
t	Отображает время в формате чч:мм: 19:45
tt	Отображает время в формате чч:мм:сс: 19:45:00
am/pm	Отображает время в 12-часовом формате (am — до полудня, pm — после полудня). Для спецификаторов hh:mm am/pm получим 07:45 pm
ampm	Отображает время в 12-часовом формате, но без указания до/после полудня. Для спецификаторов hh:mm ampm получим 07:45
a/p	Отображает время в 12-часовом формате (a — до полудня, p — после полудня). Для спецификаторов hh:mm a/p получим 07:45 p
/	Отображает используемый в Windows разделитель даты. Для спецификаторов d/m/y получим 20.7.03
:	Отображает используемый в Windows разделитель времени. Для спецификаторов h:m:s получим 19:45:0

Вещественные поля

Вещественные поля используются для хранения вещественных чисел:

- `TFloatField` — числа от $5{,}0 \cdot 10^{-324}$ до $1{,}7 \cdot 10^{+308}$ с точностью 15–16 цифр;
- `TCurrencyField` — аналогично `TFloatField`, но в денежном формате;
- `TBCDField` — вещественные десятичные числа в двоично-десятичном виде с фиксированным количеством разрядов после запятой (Delphi не поддерживает тип `BCD`, поэтому при работе с полем используется тип `Currency`, что определяет до 20 значащих цифр и до 4 цифр после запятой; применяется только в таблицах Paradox).

Следующее свойство определяет количество знаков после десятичной запятой (по умолчанию — 15):

```
property Precision: Integer
```

Это свойство не может иметь значение меньше 3.

Свойства `MaxValue`, `MinValue`, `DisplayFormat` и `EditFormat` аналогичны по назначению одноименным свойствам целочисленных полей.

Логические поля

Поля `TBooleanField` предназначены для хранения логических значений. Их следующее свойство определяет формат отображения данных, например в виде строк `True` или `False`, `Да` или `Нет` и т. п.:

```
property DisplayValues: String
```

Если свойство не определено, значения отображаются как `True` и `False`, в противном случае часть строки до первого символа точки с запятой (`;`) используется для представления истинного значения, а остальная часть — для представления ложного значения (например, `Да;Нет`).

Поля даты и времени

Поля даты и времени перечислены в табл. 3.11.

Таблица 3.11. Поля даты и времени

Поле	Описание
`TDateTimeField`	Содержит значения даты и времени в формате `TDateTime`
`TDateField`	Содержит значения даты в формате `TDate`
`TTimeField`	Содержит значения времени в формате `TTime`
`TSQLTimeStamp`	Содержит значение даты и времени в формате `TSQLTimeStamp`

Следующее свойство перечисленных в таблице полей предназначено для форматирования значений, отображаемых в визуальных компонентах:

```
property DisplayValues: String
```

Строка `DisplayValues` может содержать символы-спецификаторы, перечисленные в табл. 3.10.

Тип `TSQLTimeStamp` введен для максимально точного отображения даты-времени в технологии dbExpress.NET. Он представляет собой запись следующего вида:

```
type TSQLTimeStamp = packed record
    Year : SmallInt;
    Month : Word;
    Day : Word;
    Hour : Word;
    Minute : Word;
    Second : Word;
    Fractions : LongWord;
  end;
```

Здесь `Year` определяет год в диапазоне от 0 до 9999; `Month` — месяц от 1 до 12; `Day` — день месяца от 1 до 28–31; `Hour` — час от 0 до 23; `Minute` — минуты от 0 до 59; `Second` — секунды от 0 до 59; `Fractions` — миллисекунды от 0 до 999.

Поля для хранения значений произвольных форматов

Поле `TBlobField` предназначено для хранения больших двоичных объектов (Binary Large Object, BLOB), то есть неформатированных значений произвольной длины.

Специфические свойства поля `TBlobField` перечислены в табл. 3.12.

Таблица 3.12. Свойства поля TBlobField

Свойство	Описание
`property BlobSize: Integer;`	Содержит текущее значение длины буфера в байтах, необходимого для размещения поля

продолжение

Таблица 3.12 (продолжение)

Свойство	Описание
`type TBlobType = (ftBlob, ftMemo, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary);` `property BlobType: TBlobType;`	Определяет содержимое поля: `ftBlob` — неформатированные данные; `ftMemo` — многострочный текст; `ftGraphic` — графическое изображение; `ftFmtMemo` — многострочный текст в формате RTF; `ftParadoxOle` — OLE-объект таблиц Paradox; `ftDBaseOle` — OLE-объект таблиц dBASE; `ftTypedBinary` — типизированные двоичные данные
`property IsNull: Boolean;`	Содержит `True`, если поле пустое
`property Modified: Boolean;`	Содержит `True`, если изменилось значение поля
`property Transliterate: Boolean;`	Указывает, следует ли производить преобразование символов в кодировку ANSI при чтении строк из реальной ТБД и при записи строк обратно в ТБД в том случае, если строки в ТБД хранятся не в кодировке ANSI или содержат расширенные ASCII-символы
`property Value: String;`	Используется для чтения/записи данных

Специфические методы поля `TBlobField` перечислены в табл. 3.13.

Таблица 3.13. Методы поля TBlobField

Метод	Описание
`procedure Assign(Source: TPersistent);`	Загружает в поле значение поля `Source` другого НД
`procedure Clear;`	Очищает поле
`procedure LoadFromFile(const FileName: String);`	Загружает содержимое поля из файла
`procedure LoadFromStream (Stream: TStream);`	Загружает содержимое поля из потока
`procedure SaveToFile(const FileName: String);`	Сохраняет содержимое поля в файле
`procedure SaveToStream(Stream: TStream);`	Сохраняет содержимое поля в потоке

Поле `TVarBytesField` содержит произвольное значение длиной до 65 535 байт. Текущая длина хранится в первых двух байтах и может быть получена с помощью свойства `DataSize`.

Поле `TGraphicField` является частным случаем поля `TBlobField` и предназначено для хранения изображений.

Компоненты TSession и TDatabase. Транзакции

Описываемые в этом разделе компоненты `TSession` и `TDatabase` используются как в клиент-серверных, так и в файл-серверных БД. Однако их роль в файл-серверных БД невелика. В условиях многопользовательской работы с данными часто возникают ситуации, когда два и более пользователя пытаются изменить одни и те же (или связанные друг с другом) данные. В файл-серверных БД нет надежных способов обеспечения ссылочной целостности и непротиворечивости данных. В клиент-серверных БД для этого активно применяется механизм транзакций. Замечу, что реализация транзакций с помощью методов компонента `TDatabase` типична лишь для файл-серверных БД и БД, обслуживаемых сервером InterBase. В других серверах есть собственные эффективные средства реализации транзакций.

Компонент TSession

Компонент `TSession` реализует так называемый *сеанс связи* с базами данных, в рамках которого обеспечивается управление соединениями программы с базами данных. Каждая программа, работающая с БД любой архитектуры, автоматически получает в свое распоряжение глобальный объект `Session`, поэтому явное размещение на форме дополнительных компонентов `TSession` требуется только в том случае, если программист хочет использовать несколько независимых каналов (потоков команд) для связи с данными (один компонент создает один поток).

ПРИМЕЧАНИЕ

Каждый дополнительный сеанс, связанный с клиент-серверной БД, увеличивает на единицу счетчик подключений сервера.

Свойства, методы и события

Наиболее важные свойства компонента `TSession` перечислены в табл. 3.14.

Таблица 3.14. Свойства компонента TSession

Свойство	Описание
`property AutoSessionName: Boolean;`	Если содержит `True`, Delphi автоматически создает гарантированно уникальное имя сеанса. Такая необходимость возникает при разработке трехзвенных приложений

`TConfigModes = (cfmVirtual, cfmPersistent, cfmSession);`
`TConfigMode = set of TConfigModes;`
`property ConfigMode: TConfigMode;`
	Определяет доступность псевдонимов BDE для сеанса: `cfmVirtual` — доступны все псевдонимы; `cfmPersistent` — доступны только постоянные псевдонимы; `cfmSession` — доступны только те псевдонимы, которые созданы в рамках данного сеанса

`property DatabaseCount: Integer;`	Содержит количество активных компонентов `TDatabase`, обслуживаемых в рамках данного сеанса

продолжение

Таблица 3.14 (продолжение)

Свойство	Описание
`property Databases[Index: Integer]: TDatabase;`	Открывает индексированный доступ к любому активному компоненту TDatabase, который обслуживается данным сеансом
`property KeepConnections: Boolean;`	Если содержит True, временно созданные в рамках сеанса компоненты TDatabase будут сохранять соединение с сервером БД, даже если нет ни одного связанного с ними активного НД
`property SessionName: String;`	Уникальное имя сеанса. Это свойство игнорируется, если свойство AutoSessionName содержит значение True
`property SQLHourGlass: Boolean;`	Если содержит True, указатель мыши изменяет форму при операциях в рамках данного сеанса

Наиболее важные методы компонента TSession перечислены в табл. 3.15.

Таблица 3.15. Методы компонента TSession

Метод	Описание
`procedure AddAlias(const Name, Driver: String; List: TStrings);`	Создает новый временный псевдоним для связи с SQL-сервером: Name — имя псевдонима; Driver — имя обслуживающего драйвера; List — список параметров. Для сохранения временного псевдонима используйте метод SaveConfigFile
`procedure AddPassword(const Password: String);`	Добавляет к сеансу пароль для доступа к защищенным таблицам Paradox
`procedure AddStandardAlias(const Name, Path, DefaultDriver: String);`	Создает новый временный псевдоним для связи с таблицами Paradox, dBase или ASCII: Name — имя псевдонима; Path — путь доступа к таблицам; DefaultDriver — имя обслуживающего драйвера (может быть одним из трех: Paradox, DBASE или ASCIIDRV). Для сохранения временного псевдонима используйте метод SaveConfigFile
`procedure Close;`	Закрывает сеанс и все обслуживаемые им соединения с БД
`procedure DeleteAlias(const Name: String);`	Удаляет псевдоним из сеанса
`procedure DropConnections;`	Освобождает все неактивные соединения
`function FindDatabase(const DatabaseName: String): TDatabase;`	Возвращает NIL, если БД DatabaseName недоступна в текущем сеансе

Метод	Описание
`procedure GetAliasNames(List: TStrings);`	Наполняет список List именами доступных для сеанса псевдонимов с учетом свойства ConfigMode
`procedure GetDatabaseNames (List: TStrings);`	Наполняет список List именами всех доступных для сеанса псевдонимов
`function GetPassword: Boolean;`	Создает и показывает стандартное диалоговое окно для доступа к защищенным таблицам Paradox
`procedure GetStoredProcNames(const DatabaseName: String; List: TStrings);`	Наполняет список List именами хранимых процедур из БД DatabaseName
`procedure GetTableNames(const DatabaseName, Pattern: String; Extensions, SystemTables: Boolean; List: TStrings);`	Наполняет список List именами таблиц из БД DatabaseName: Pattern — шаблон выбора; Extensions = True, SystemTables = True, если должны отображаться, соответственно, расширения табличных файлов и системные таблицы
`function IsAlias(const Name: String): Boolean;`	Возвращает True, если сеанс обслуживает БД с псевдонимом Name
`procedure SaveConfigFile;`	Сохраняет временные псевдонимы, созданные методами AddAlias и AddStandardAlias в конфигурационном файле BDE

События компонента TSession перечислены в табл. 3.16.

Таблица 3.16. События компонента TSession

Событие	Описание
`type TPasswordEvent = procedure(Sender: TObject; var Continue: Boolean) of object;`	Возникает при открытии защищенных таблиц Paradox
`property OnPassword: TPasswordEvent;`	
`property OnStartup: TNotifyEvent;`	Возникает при активизации сеанса

Использование

Как видно из приведенного обзора свойств, методов и событий компонента TSession, его основное назначение — управление псевдонимами, получение информации о базах данных и их таблицах, а также организация беспарольного доступа к защищенным таблицам Paradox.

С помощью методов AddAlias и AddStandardAlias программа может создавать временные псевдонимы, которые обеспечивают доступ к данным только

внутри сеанса. Однако с помощью метода SaveConfigFile можно перенести временные псевдонимы в конфигурационный файл BDE и, таким образом, сделать их постоянными.

Методы GetAliasNames и GetDatabaseNames различаются тем, что первый учитывает значение свойства ConfigMode, а второй игнорирует его и показывает любые доступные сеансу псевдонимы, в том числе локальные, которые создают подключаемые к сеансу компоненты TDatabase (локальные псевдонимы не сохраняются методом SaveConfigFile).

Целая группа методов поддерживает доступ к защищенным таблицам. Если эти методы не используются, при попытке открыть таблицу появляется стандартное диалоговое окно запроса пароля.

Это же окно создается при обращении к методу GetPassword. Если с помощью метода AddPassword указать пароль до открытия таблицы и не предпринимать никаких других мер, таблица будет открыта без диалога с пользователем. Наконец, можно создать обработчик события OnPassword и в нем реализовать нестандартный диалог.

ВНИМАНИЕ

Если в обработчике вызвать метод GetPassword, программа «зациклится».

Для демонстрации некоторых возможностей компонента TSession (рис. 3.5) в файлах к проекту вы найдете проект Ch03\Session\SessionDemo.dpr (листинг 3.2).

Листинг 3.2. Демонстрация некоторых возможностей компонента TSession

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, Borland.Vcl.StdCtrls,
  System.ComponentModel, Borland.Vcl.Db, Borland.Vcl.DBTables;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    Button2: TButton;
    Button4: TButton;
    Button5: TButton;
    Database1: TDatabase;
    Database2: TDatabase;
    procedure FormDestroy(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
```

```
    { Private declarations }
    NewAliasName: String;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
Показывает список постоянных псевдонимов
var
  List: TStringList;  // Для сортировки псевдонимов
begin
  List := TStringList.Create;
  List.Sorted := True;
  // Если убрать следующий оператор, список будет совпадать
  // с тем, что возвращает AddStandardAlias
  Session.ConfigMode := cmPersistent;
  Session.GetAliasNames(List);
  Memo1.Lines.Assign(List);
end;

procedure TForm1.Button2Click(Sender: TObject);
// Показывает список любых псевдонимов
var
  List: TStringList;
begin
  List := TStringList.Create;
  List.Sorted := True;
  Session.GetDatabaseNames(List);
  Memo1.Lines.Assign(List);
end;

procedure TForm1.Button4Click(Sender: TObject);
// Вывод списка таблиц
var
  List: TStringList;
begin
  List := TStringList.Create;
  List.Sorted := True;
  Session.GetTableNames('BIBLDATA','',True,False,List);
  Memo1.Lines.Assign(List)
end;
```

продолжение

Листинг 3.2 (продолжение)

```
procedure TForm1.Button5Click(Sender: TObject);
// Создание нового псевдонима
begin
  if NewAliasName = '' then
  begin
    NewAliasName :=
        InputBox('Создание нового псевдонима', 'Псевдоним:','');
    if NewAliasName = '' then Exit;
    if Session.IsAlias(NewAliasName) then
    begin
      ShowMessage(
        Format('Псевдоним %s уже существует!', [NewAliasName]));
      NewAliasName := '';
      Exit
    end;
    Session.AddStandardAlias(NewAliasName,'C:\','ASCIIDRV');
    Session.SaveConfigFile
  end
end;

procedure TForm1.FormDestroy(Sender: TObject);
// Удаление ненужного псевдонима
begin
  if Session.IsAlias(NewAliasName) then
    Session.DeleteAlias(NewAliasName);
  // В документации говорится, что вызов DeleteAlias
  // должен сопровождаться обращением к SaveConfigFile.
  // Как видите, это не так
end;

end.
```

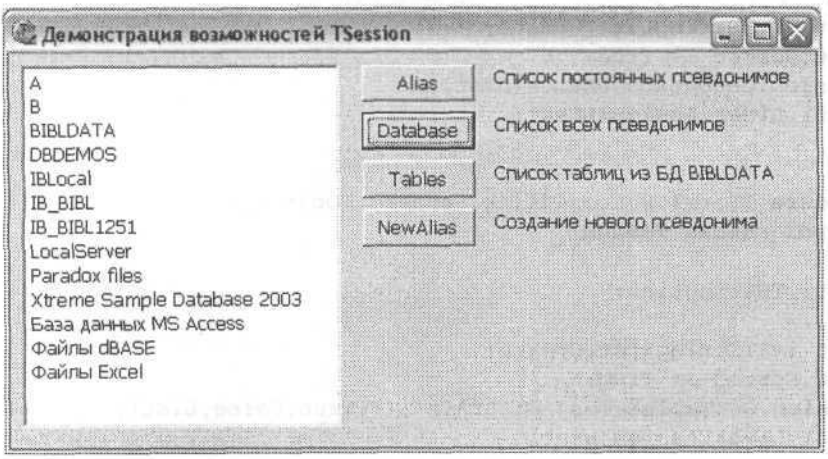


Рис. 3.5. Окно проекта SessionDemo.dpr

Небольшие пояснения. Обработчики щелчков на кнопках Button1 (Alias), Button2 (Database) и Button4 (Tables) помещают создаваемые методами GetAliasNames, GetDatabaseNames и GetTableNames списки в промежуточный компонент List, чтобы отсортировать списки по алфавиту (строки в TMemo.Lines сортировать значительно сложнее).

Компонент TDatabase

В отличие от компонента TSession, который управляет множеством БД, компонент TDatabase обслуживает единственную базу данных (как файловую, так и серверную). Он выполняет две основные функции: создает локальный псевдоним БД и управляет транзакциями.

Наиболее важные свойства компонента TDatabase перечислены в табл. 3.17.

Таблица 3.17. Свойства компонента TDatabase

Свойство	Описание
`property AliasName: String;`	Определяет имя псевдонима БД, которую обслуживает компонент
`property Connected: Boolean;`	Содержит True, если соединение компонента БД активно
`property DatabaseName: String;`	Определяет локальный псевдоним БД или путь доступа к таблицам Paradox или dBASE
`property DataSetCount: Integer;`	Указывает количество обслуживаемых компонентом наборов данных
`property DataSets[Index: Integer]: TDBDataSet;`	Открывает индексированный доступ ко всем обслуживаемым компонентом наборам данных
`property Directory: String;`	Содержит папку размещения таблиц Paradox или dBASE
`property DriverName: String;`	Указывает имя драйвера, обслуживающего БД
`property Exclusive: Boolean;`	Если содержит True, блокирует доступ других пользователей к БД Paradox или dBASE
`property InTransaction: Boolean`	Содержит True, если в БД имеется незавершенная транзакция
`property KeepConnection: Boolean;`	Если содержит True, компонент будет сохранять связь с БД даже после закрытия всех обслуживаемых им НД
`property LoginPrompt: Boolean;`	Разрешает/запрещает диалог с пользователем при первом открытии серверной БД
`property Params: TStrings;`	Содержит список параметров, передаваемых BDE в момент открытия БД

продолжение

Таблица 3.17 (продолжение)

Свойство	Описание
`property ReadOnly: Boolean;`	Если содержит `True`, компонент открывает БД только для чтения
`TTransIsolation = (tiDirtyRead, tiReadCommitted, tiRepeatableRead); property TransIsolation: TTransIsolation;`	Определяет уровень изоляции транзакций (см. далее раздел «Транзакции»)

Наиболее важные методы компонента `TDatabase` перечислены в табл. 3.18.

Таблица 3.18. Методы компонента TDatabase

Метод	Описание
`procedure CloseDatasets;`	Закрывает все связанные с компонентом НД
`procedure Commit;`	Подтверждает все изменения, сделанные в текущей транзакции, и закрывает ее
`procedure Rollback;`	Отменяет все изменения, сделанные в текущей транзакции, и закрывает ее
`procedure StartTransaction;`	Открывает транзакцию

Транзакции

В технологии BDE.NET клиентская программа управляет транзакциями с помощью трех методов компонента `TDatabase`: `StartTransaction` (начинает транзакцию), `Commit` (подтверждает транзакцию) и `Rollback` (отменяет все изменения, сделанные с момента старта транзакции):

```
Database1.StartTransaction; // Стартуем транзакцию
... // Любые изменения в любых таблицах
try
 Database1.Commit     // Пытаемся подтвердить изменения
except
 Database1.Rollback  // Откат, если подтвердить не удалось
end;
```

ПРИМЕЧАНИЕ

Откат транзакции ликвидирует также все последствия возможного срабатывания триггеров на сервере БД в рамках отмененной транзакции.

Изоляция транзакций

Компонент `TDatabase` с помощью своего свойства `TransIsolatioin` обеспечивает нужный уровень изоляции транзакций. Изолировать транзакции необходимо при одновременном доступе к одним и тем же данным нескольких пользователей.

При уровне изоляции *Dirty Read* любой пользователь видит любые изменения в данных, даже если они еще не подтверждены. В этой ситуации может нарушиться непротиворечивость данных: если пользователь, изменивший данные, по каким-либо причинам откатит сделанные им изменения, другой пользователь никак не узнает об этом в момент, когда он будет делать свои изменения. Уровень *Dirty Read* фактически не изолирует транзакции. Только этот уровень возможен для файл-серверных БД, вот почему в СУБД этого типа транзакции используются крайне редко.

Уровень *Read Committed* разрешает транзакции видеть только подтвержденные изменения. Этот уровень рекомендуется для клиент-серверных БД (по умолчанию компонент `TDatabase` автоматически устанавливает уровень *Dirty Read* для файл-серверных и уровень *Read Committed* — для клиент-серверных БД).

Для клиент-серверных БД возможен также уровень *Repeatable Read* (повторяющееся чтение). На этом уровне изоляции также читаются только подтвержденные изменения, однако раз прочитанная запись помещается в локальный буфер и повторно читается из этого буфера, то есть повторное чтение не позволяет увидеть изменений, внесенных другим пользователем, даже если эти изменения подтверждены.

Свойство UpdateMode наборов данных

Успех или неуспех транзакции во многом зависит от свойства `UpdateMode` соответствующего НД. Это свойство определяет, по каким столбцам сервер ищет в таблице запись, чтобы внести в нее изменения. Если пользователь А первым подтвердит изменения, а свойство `UpdateMode` соответствующего НД у конкурирующего пользователя Б имеет заданное по умолчанию значение `upWhereAll`, его транзакция будет отвергнута, так как в этом случае сервер ищет запись, в которой все столбцы имеют те значения, которые были к моменту старта транзакции Б. Если свойство имеет значение `upWhereChange`, ищется запись, у которой ключевые столбцы и столбцы с изменениями остались прежними. В этом случае транзакция Б пройдет успешно, если оба пользователя изменяли разные столбцы, и может быть нарушена непротиворечивость данных. Еще хуже защищаются данные в случае, если свойство `UpdateMode` имеет значение `upWhereKeyOnly`, которое заставляет сервер искать запись только по ключевым столбцам.

Таким образом, значение `tiReadCommitted` (или `tiRepeatableRead`) свойства `TransIsolation` компонента `TDatabase` в сочетании со значением `upWhereAll` свойства `UpdateMode` НД в максимальной степени защищают данные от искажения в условиях многопользовательской работы.

Далее описывается пример перевода денег с дебиторского счета на кредиторский под защитой транзакции. В БД `IB_BIBL.GDB` есть две небольшие таблицы, `DEB` и `CRED`, имитирующие дебиторские и кредиторские счета. На рис. 3.6 показано окно работающей программы.

Рис. 3.6. Окно программы, иллюстрирующей использование транзакции

Ниже показан обработчик щелчка на кнопке Перевести:

```
procedure TForm2.Button1Click(Sender: TObject);
// Переводит деньги с дебиторского счета
// на кредиторский под защитой транзакции
var
  Sum: Double;  // Сумма перевода
  CmdStr: TStringList;
begin
  // Проверяем сумму перевода
  if SpinEdit1.Text = '' then
    Exit;
  Sum := StrToFloat(SpinEdit1.Text);
  if Sum > DEBSumm.Value then
  begin     // Сумма слишком велика
    SpinEdit1.Text := FloatToStr(DEBSumm.Value);
    SpinEdit1.SetFocus;
    Exit;
  end;
  // Строка для снятия денег:
  CmdStr := TStringList.Create;
  CmdStr.Add(Format('UPDATE DEB SET SUMM=%d WHERE NCOUNT=%d',
       [Round(DEBSumm.Value - Sum), DEBNCount.Value]));
  // Добавляем ее в свойство SQL-запроса:
  Query1.SQL := CmdStr;
  CmdStr.Clear;
  // Строка для начисления денег:
  CmdStr.Add(Format('UPDATE CRED SET SUMM=%d WHERE NCOUNT=%d',
       [Round(CREDSumm.Value + Sum), CREDNCount.Value]));
  with DB do
  begin
    StartTransaction;    // Стартуем транзакцию
    try
      Query1.ExecSQL;    // Снимаем деньги
      Query1.SQL := CmdStr;
```

```
      Query1.ExecSQL;      // Начисляем деньги
      Commit;              // Подтверждаем транзакцию
      ShowMessage('OK');
    except                 // Ошибка!
      Rollback;            // Откатываем транзакцию
      ShowMessage('NO');
    end;
    // Обновляем отображаемые данные:
    DEB.Active := False;
    DEB.Active := True;
    CRED.Active := False;
    Cred.Active := True;
    SpinEdit1.Value := 0;
    SpinEdit1.SetFocus;
  end;
end;

procedure TForm2.FormActivate(Sender: TObject);
// Передает фокус ввода полю SpinEdit1 в момент активизации формы
begin
  SpinEdit1.SetFocus;
end;
```

Для ввода суммы перевода используется поле типа `TSpeenEdit` (категория Samples). Этот компонент рассчитан на ввод целых чисел и отвергает любой неправильный ввод. Обратите внимание: перед выполнением операторов UPDATE новые суммы счетов приводятся к целочисленному виду. Сделано это с тем, чтобы исключить влияние локализации на вид строкового представления вещественных чисел: если вы, как и я, работаете с русифицированной версией Windows, разлелителем целой и дробной частей вещественного числа является запятая, в то время как InterBase в типе `Float` использует для этого точку. Чтобы не усложнять программу, я решил отказаться от копеек. На практике это, разумеется, недопустимо.

Наборы данных

Под набором данных (НД) понимается группа записей из одной или нескольких таблиц БД, доступная для компонентов-наборов `TTable`, `TQuery` или `TStoredProc`. Эти компоненты порождены от общего родительского класса `TDBDataSet`, который и рассматривается в этом разделе. Таким образом, все излагаемые здесь сведения в равной степени относятся к любому компоненту-набору. Особенностям работы с компонентами `TTable` и `TQuery` посвящены два следующих раздела, а компонент `TStoredProc` рассматривается в главе 7.

Обзор свойств, методов и событий

Здесь рассматриваются наиболее важные свойства, методы и события класса `TDBDataSet`.

Свойства

Свойства компонента TDBDataSet перечислены в табл. 3.19.

Таблица 3.19. Свойства набора данных

Свойство	Описание
property Active: Boolean;	Открывает/закрывает НД
property ActiveRecord: Integer;	Указывает индекс активной записи
property AggFields: TFields;	Содержит все агрегатные поля НД. В отличие от вычисляемых полей, которые оперируют полями одной записи, агрегатные поля оперируют значениями всех записей одного поля
property AutoCalcFields: Boolean;	Разрешает/запрещает возникновение события OnCalcFields
property Bof: Boolean;	Содержит True, если курсор НД указывает на первую запись
type TBookmarkStr: **String**; **property** Bookmark: TBookmarkStr;	Определяет закладку на текущей записи. К помеченной записи можно быстро перейти методом GotoBookmark
property BookmarkSize: Integer;	Определяет длину закладки в байтах
property BufferCount: Integer;	Указывает количество записей во внутреннем буфере (кэше)
property Buffers: IntPtr;	Открывает индексированный доступ к записям во внутреннем буфере
property CasheBlobs: Boolean;	Разрешает/запрещает кэширование BLOB-полей
property CashedUpdates: Boolean;	Разрешает/запрещает кэширование вносимых пользователем изменений
property CanModify: Boolean;	Указывает, может ли пользователь изменять НД
property CurrentRecord: Integer;	Указывает индекс текущей записи во внутреннем буфере
property Database: TDatabase;	Определяет компонент TDatabase, связанный с данным НД
property DatabaseName: **String**;	Содержит псевдоним базы данных
property DataSetField: TDataSetField;	Содержит ссылку на владельца вложенного НД
property DataSource: TDataSource;	Используется в потомках для указания детального НД в связи *один ко многим*
property DBSession: TSession;	Определяет связанный с НД компонент TSession
property DefaultFields: Boolean;	Содержит True, если для НД не создано ни одного объекта-поля

Свойство	Описание
`property EOF: Boolean;`	Содержит `True`, если курсор НД сместился за последнюю запись
`property FieldCount: Integer;`	Содержит количество полей в НД
`property FieldList: TFieldList;`	Содержит список имен всех полей со всеми вложенными полями
`property Fields: TFields;`	Позволяет обратиться к полям по их индексу (см. далее). Первое поле НД имеет индекс 0
`property FieldValues[const FieldName: String]: Variant;`	Позволяет обратиться к значениям полей по имени поля (см. далее)
`property Filter: String;`	Задает фильтрующее выражение
`property Filtered: Boolean;`	Разрешает/запрещает фильтрацию записей НД
`property FilterOptions: TFilterOptions;`	Определяет условия фильтрации в текстовых полях: `foCaseInsensitive` — учитывать регистр букв; `foNoPartialCompare` — поиск на точное соответствие образцу
`property Found: Boolean;`	Содержит `True`, если вызов одного из методов `FindXXXX` был успешным
`property IsUnidirectional: Boolean;`	Если содержит `True`, НД имеет однонаправленный курсор. В таких НД запрещена иная навигация, кроме как методами `First` и `Next`. Эти НД не поддерживают фильтрацию, закладки, не могут иметь подстановочных полей и отображаться в сетках
`property Modified: Boolean;`	Содержит `True`, если текущая запись НД была изменена
`property RecNo: LongInt;`	Определяет номер текущей записи
`property RecordCount: LongInt;`	Содержит количество записей в текущем НД
`property RecordSize: Word;`	Определяет размер в байтах буфера для размещения одной записи НД
`property SessionName: String;`	Содержит имя компонента `TSession`
`type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);` `property State: TDataSetState;`	Указывает состояние НД: `dsInactive` — закрыт; `dsBrowse` — просмотр; `dsEdit` — редактирование; `dsInsert` — вставка; `dsSetKey` — поиск записи; `dsCalcFields` — установка вычисляемых полей; `dsFilter` — фильтрация записей; `dsNewValue` — обновление свойства `TField.NewValue`; `dsOldValue` — обновление свойства `TField.OldValue`; `dsCurValue` — обновление свойства `TField.CurValue`; `dsBlockRead` — чтение блока записей; `dsInternalCalc` — обновление полей `TField`, для которых `FieldKind = fkInternalCalc`

продолжение

Таблица 3.19 (продолжение)

Свойство	Описание
type TUpdateMode = (upWhereAll, upWhereChanged, upWhereKeyOnly); **property** UpdateMode: TUpdateMode;	Определяет способ использования полей при поиске обновляемых записей: upWhereAll — все поля; upWhereChanged — только ключевые поля и значения обновленных полей; upWhereKeyOnly — только ключевые поля

В Delphi свойство Fields ссылается на объект класса TFields. Однако свойство Fields типа TField этого объекта является умалчиваемым свойством, поэтому два следующих оператора эквивалентны:

```
Caption := Table1.Fields[0].Value;
Caption := Table1.Fields.Fields[0].Value;
```

Свойство FieldValues является умалчиваемым свойством для НД, поэтому два следующих оператора эквивалентны:

```
Caption := Table1.FieldValues['Name'];
Caption := Table1['Name'];
```

Замечу, что параметром обращения к набору данных может быть список полей, в котором соседние поля разделяются точкой с запятой. Это позволяет единственным оператором обращения к свойству прочитать в вариантный массив сразу несколько полей текущей записи:

```
var
  V: Variant;
begin
  V := CreateVarArray([0, 2], varVariant);
  V := Books['BQuan;BName;BOpt'];
  ...
end;
```

Методы

Наиболее важные методы класса TDBDataSet перечислены в табл. 3.20.

Таблица 3.20. Методы набора данных

Метод	Описание
function ActiveBuffer: IntPtr;	Возвращает указатель на буфер текущей записи
procedure Append;	Добавляет пустую запись в конец НД
procedure AppendRecord(**const** Values: **array of const**);	Создает новую запись, заполняет ее поля значениями Values и добавляет ее в НД
procedure ApplyUpdates;	Записывает кэш обновлений в БД
function BookmarkValid(Bookmark: TBookmark): Boolean; **override**;	Возвращает True, если с закладкой Bookmark связано правильное значение

Метод	Описание
`procedure Cancel;`	Отменяет все изменения текущей записи, которые не были сохранены в БД
`procedure CancelUpdates;`	Очищает кэш обновлений
`procedure CheckBrowseMode;`	Если НД находился в состоянии редактирования или вставки, вызывает метод `Post` для записи изменений в БД
`procedure ClearFields;`	Очищает все поля текущей записи
`procedure Close;`	Закрывает НД
`procedure CommitUpdates;`	Очищает кэш после успешного обновления НД
`function CompareBookmarks (Bookmark1, Bookmark2: TBookmark): Integer;`	Сравнивает две закладки и возвращает 1, если они различаются, и 0, если они идентичны или пусты
`procedure Delete;`	Удаляет текущую запись
`procedure Edit;`	Переводит НД в режим редактирования
`function FieldByName(const FieldName: String): TField;`	Обеспечивает доступ к полю по его имени `FieldName`
`function FindField(const FieldName: String): TField;`	Ищет поле `FieldName` в НД и возвращает ссылку на поле или `NIL`, если поле не найдено
`function FindFirst: Boolean;`	Пытается установить курсор на первую запись НД и возвращает `True` в случае успеха
`function FindLast: Boolean;`	Пытается установить курсор на последнюю запись НД и возвращает `True` в случае успеха
`function FindNext: Boolean;`	Пытается установить курсор на следующую запись НД и возвращает `True` в случае успеха
`function FindPrior: Boolean;`	Пытается установить курсор на предыдущую запись НД и возвращает `True` в случае успеха
`procedure First;`	Устанавливает курсор на первую запись в НД
`procedure FreeBookmark(Bookmark: TBookmark); virtual;`	Освобождает память, связанную с закладкой `Bookmark`
`function GetBookmark: TBookmark; virtual;`	Создает закладку на текущей записи и возвращает указатель на нее
`function GetCurrentRecord (Buffer: IntPtr): Boolean;`	Копирует текущую запись в буфер `Buffer` и возвращает `True` в случае успеха
`procedure GetFieldNames(List: TStrings);`	Наполняет список `List` именами всех полей НД
`procedure GotoBookmark(Bookmark: TBookmark);`	Обеспечивает возврат к записи, связанной с закладкой `Bookmark`
`procedure Insert;`	Переводит НД в режим вставки записей
`procedure InsertRecord(const Values: array of const);`	Создает пустую запись, наполняет ее поля значениями `Values` и вставляет ее в НД

продолжение

Таблица 3.20 (продолжение)

Метод	Описание
`function IsEmpty: Boolean;`	Возвращает `True`, если в НД нет записей
`function IsSequenced: Boolean; override;`	Возвращает `True`, если к записям НД можно обращаться с помощью свойства `RecNo`
`procedure Last;`	Устанавливает курсор на последнюю запись
`function Locate(const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean;`	Ищет в полях, перечисленных в параметре `KeyFields`, значения, указанные в `KeyValues` при условиях, заданных параметром `Options`. Если запись найдена, делает ее текущей и возвращает `True`
`function Lookup(const KeyFields: String; const KeyValues: Variant; const ResultFields: String): Variant;`	Используется в детальных НД для поиска в полях `KeyFields` значений `KeyValues`. При успехе возвращает значения полей `ResultFields`
`procedure Next;`	Перемещает курсор к следующей записи
`procedure Open;`	Открывает НД
`procedure Post; virtual;`	Сохраняет вставленную или отредактированную текущую запись в таблице БД
`procedure Prior;`	Перемещает курсор к предыдущей записи
`procedure Refresh;`	Обновляет НД данными из БД
`procedure SetFields(const Values: array of const);`	Устанавливает значения `Values` во все поля текущей записи

События

Все события определены в секции **published** класса `TDBDataSet` и поэтому доступны в инспекторе объектов для компонентов-наследников этого класса.

События компонента `TDBDataSet` перечислены в табл. 3.21.

Таблица 3.21. События набора данных

Событие	Описание
`type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;` `property AfterXXXX: TDataSetNotifyEvent;`	Серия событий `AfterXXXX` возникает после изменения состояния НД. Вместо символов *XXXX* в названиях событий должны указываться следующие: `Cancel` — возникает после отмены изменений в текущей записи; `Close` — возникает после закрытия НД; `Delete` — возникает после уничтожения записи НД; `Edit` — возникает после перехода НД в режим `dsEdit`; `Insert` — возникает после вставки записи; `Open` — возникает после открытия НД; `Post` — возникает после выполнения метода `Post`; `Scroll` — возникает после перехода к следующей записи

Событие	Описание
property Before*XXXX*: TDataSetNotifyEvent;	Серия событий Before*XXXX* возникает непосредственно перед изменением состояния НД. Вместо символов *XXXX* в названиях событий должны указываться следующие: Cancel — возникает перед отменой изменений в текущей записи; Close — возникает перед закрытием НД; Delete — возникает перед уничтожением записи НД; Edit — возникает перед переходом НД в режим dsEdit; Insert — возникает перед вставкой записи; Open — возникает перед открытием НД; Post — возникает перед выполнением метода Post; Scroll — возникает после перехода к следующей записи
property OnCalcFields: TDataSetNotifyEvent;	Возникает при необходимости переопределения вычисляемых полей
type TFilterRecordEvent = **procedure**(DataSet: TDataSet; **var** Accept: Boolean) **of object**; **property** OnFilterRecord: TFilterRecordEvent;	Возникает при фильтрации записей (после установки значения True в свойство Filtered). Обработчик должен поместить True в переменную Accept, если текущая запись удовлетворяет критерию фильтрации
property OnNewRecord: TDataSetNotifyEvent;	Возникает при вставке или добавлении новой записи. Обработчик может установить начальные значения в поля записи и/или произвести каскадные изменения в связанных НД

Открытие и закрытие набора данных

Состояние НД определяется его свойством State, доступным только для чтения на этапе прогона программы. Значения, которые может принимать свойство State, описаны ранее (см. описание свойства State в табл. 3.19).

Начальное состояние любого НД — dsInactive. Чтобы открыть НД, используется его метод Open или свойство Active, в которое нужно поместить значение True. После успешного открытия НД переходит в состояние dsBrowse, а его курсор устанавливается на первую запись.

Чтобы закрыть НД, вызывается метод Close. Замечу, что, если НД закрывается, находясь в режимах dsInsert или dsEdit, изменения, сделанные в текущей записи, не запоминаются. Закрыть НД можно также, поместив значение False в его свойство Active.

Программный доступ к записям

Для просмотра, вставки, редактирования и удаления записей обычно используются соответствующие визуальные компоненты (TDBGrid, TDBEdit, TDBNavigator и т. д.), которые автоматически вызывают нужные методы НД. В некоторых случаях

(например, при обработке статистических данных или при реализации каскадных изменений) необходим программный доступ к информации без использования визуальных компонентов.

При программном доступе прежде всего нужно убедиться в том, что НД содержит записи. Для этого вызывается метод `IsEmpty`, который возвращает `False`, если в НД есть хотя бы одна запись. После использования или редактирования текущей записи переход к следующей реализуется методом `FindNext` или `Next`. Таким образом, для программного доступа ко всем записям НД применяются такого рода операторы:

```
Table1.Open;
if not Table1.IsEmpty then
  repeat
    // Используем информацию из текущей записи или редактируем ее
  until not Table1.FindNext; // Переходим к следующей записи
Table1.Close;
```

Другой вариант:

```
Table1.Open;
while not Table1.EOF do
  begin
    // Используем информацию из текущей записи или редактируем ее
    Table1.Next;  // Переходим к следующей записи
  end;
Table1.Close;
```

Перед редактированием текущей записи набор данных необходимо перевести в состояние `dsEdit` методом `Edit`, а после редактирования сохранить сделанные изменения в соответствующей таблице БД, вызвав метод `Post`:

```
Table1.Edit;
Table1['Number'] := 123456;
Table1.Post;
```

По такой же схеме осуществляется вставка или добавление новой записи, при этом вместо `Edit` вызывается метод `Insert` (вставка записи) или `Append` (добавление записи). Если НД не индексирован, вставленная запись помещается на место текущей записи, а текущая запись становится следующей за ней; добавленная запись помещается в конец НД. Если НД индексирован, оба метода вставляют запись в позицию, определяемую текущим индексом. Рассмотрим следующие операторы:

```
Table1.Insert;  // или Table1.Append;
{Наполнение пустой записи нужными значениями:}
Table1['Number'] := 123456;
Table1['Name'] := 'Иванов И.И.';
Table1['Children'] := False;
Table1.Post;    // Запоминание записи в ТБД
```

Вместо всех этих операторов можно использовать единственный оператор, вызвав метод `InsertRecord` или `AppendRecord`:

```
Table1.InsertRecord([123456, 'Иванов И.И.', False]);
```

Элементы в конструкторе массива должны идти в строгом соответствии с порядком следования полей в НД и содержать допустимые для каждого поля выражения.

Замечу, что вставленная запись становится текущей. Если НД индексирован, вставлять запись в цикле просмотра **repeat** ... **until** или **while not** EOF **do** нельзя, так как положение новой записи в НД будет соответствовать значению ее индексного поля и, следовательно, положение курсора НД может измениться. Точно так же в цикле нельзя изменять значения индексных полей.

Редактирование, удаление и вставка записей возможны только для НД, свойство CanModify которых содержит значение True. Для компонента TTable это условие выполняется, если свойство ReadOnly содержит заданное по умолчанию значение False, для двух других компонентов-наборов — лишь при выполнении определенных условий.

Навигация по набору данных

Под курсором набора данных понимается указатель текущей записи в конкретном НД. Текущая запись — это та запись, над которой в данный момент времени можно выполнять какие-либо операции (удаление, изменение, чтение значений, содержащихся в полях записи).

С помощью методов First, Last, Next, Prior, MoveBy, FindFirst, FindLast, FindNext, FindPrior можно изменять положение курсора и, следовательно, выбирать нужную запись.

Последовательная навигация по записям

Для выполнения действий по последовательному перебору записей, начиная от некоторой стартовой записи и до конца набора данных, используют цикл **while**...**do** или **repeat**...**until**:

```
with Table1 do
begin
  First;
  while not EOF do
  begin
    {Какие-либо действия над очередной записью}
    Next;
  end; {while}
end; {with}
```

```
with Table1 do
begin
  First;
  repeat
    {Какие-либо действия над очередной записью}
  until not FindNext;
end; {with}
```

Точно так же можно перемещаться от конца НД к его началу:

```
with Table1 do
```

```
begin
  Last;
  while not BOF do
  begin
    {Какие-либо действия над очередной записью}
    Prior;
  end; {while}
end; {with}

with Table1 do
begin
  Last;
  repeat
    {Какие-либо действия над очередной записью}
  until not FindPrior;
end; {with}
```

Если НД допускает вызов записи по ее номеру (его свойство `IsSequenced` в этом случае должно содержать значение `True`), можно использовать цикл **`for`**. Например, следующий обработчик события `TForm.OnActivate` поместит все значения строкового поля `BName` НД `Books` в многострочное текстовое поле `Memo1`:

```
procedure TForm1.FormActivate(Sender: TObject);
var
  RecNo: Integer;
begin
  if Books.IsSequenced then
  for RecNo := 1 to Books.RecordCount do
  begin
    Books.RecNo := RecNo;
    Memo1.Lines.Add(Table1['BName'])
  end;
end;
```

Замечу, что для компонентов `TTable` и `TQuery`, связанных с файл-серверными таблицами БД, свойство `IsSequenced` всегда содержит значение `True`. В клиент-серверной архитектуре оно зависит от используемого сервера БД. Свойство `RecNo` набора данных определяет порядковый номер обрабатываемой записи (нумерация начинается с 1), а его свойство `RecordCount` — количество записей.

Во всех случаях следует помнить, что, если в ходе выполнения цикла последовательного перебора записей изменится ключевое поле записи или в НД будет вставлена новая (удалена старая) запись, курсор НД может изменить свое положение и правильная работа цикла нарушится. Например, следующий фрагмент программы не обеспечит удаление всех записей из таблицы:

```
with Table1 do while not EOF do
begin
  Delete;
  Next;
end;
```

После удаления очередной записи курсор НД перейдет на следующую, но вызов Next заставит его сместиться еще раз, и запись будет пропущена. Правильный вариант:

```
with Table1 do while not EOF do
  Delete;
```

Использование закладок

Подобно тому, как в книге нужную страницу можно заложить закладкой и впоследствии быстро найти, в НД аналогичные действия можно осуществить для записи. Для этой цели НД обладает следующими методами: GetBookmark (создает закладку), GotoBookmark (перемещает курсор к закладке), FreeBookmark (удаляет закладку), BookmarkValid (проверяет закладку), CompareBookmarks (сравнивает две закладки). Закладки применяют в том случае, когда нужно временно покинуть текущую запись, а затем вновь к ней вернуться.

Использовать закладки предельно просто:

```
var
  MyBookmark: TBookMark;
...
// Создаем закладку для текущей записи:
  MyBookmark := Table1.GetBookmark;
...
// Переходим к записи с закладкой:
if Table1.BookmarkValid(MyBookmark) then
   Table1.GotoBookmark(MyBookmark);
 ...
// Освобождаем выделенные для закладки ресурсы:
if Table1.BookmarkValid(MyBookmark) then
   Table1.FreeBookmark(MyBookmark);
```

Поиск записей в наборах данных

В этом разделе рассматриваются методы Locate и Lookup, которые можно использовать при работе с любым компонентом-набором. Компонент TTable имеет дополнительную группу методов для поиска записей.

Метод Locate

Метод Locate ищет первую запись, удовлетворяющую критерию поиска, и, если такая запись найдена, делает ее текущей. В этом случае в качестве результата возвращается значение True. Если запись не найдена, возвращается значение False, и курсор не меняет своего положения:

```
function Locate(const KeyFields: String;
                const KeyValues: Variant;
                Options: TLocateOptions): Boolean;
```

Список KeyFields указывает поле или несколько полей, по которым ведется поиск. В случае нескольких поисковых полей их названия разделяются точкой

с запятой. Критерии поиска задаются в вариантном массиве `KeyValues` так, что *i*-е значение в `KeyValues` ставится в соответствие *i*-му полю в `KeyFields`. Параметр `Options` указывает необязательные значения режимов поиска:

- `loCaseInsensitive` — поиск ведется без учета возможной разницы регистра букв в текстовых полях и в критерии поиска;
- `loPartialKey` — запись считается удовлетворяющей условию поиска, если она содержит часть поискового контекста; например, удовлетворяющими контексту «Диа» будут признаны записи со значениями в поле поиска «Диалект», «Диалог-МИФИ», «ДиаСофт» и т. д.

Метод `Locate` производит поиск по любому полю независимо от того, входит оно в состав какого-либо индекса или нет. Если поле (поля) поиска входит в какой-либо индекс, `Locate` автоматически использует его, что существенно ускоряет поиск.

Для иллюстрации использования метода `Locate` изменим пример, рассмотренный в главе 1: поместите на нижнюю панель компонент `TDateTimePicker` (вкладка Win32 палитры компонентов) и напишите следующий обработчик его события `OnChange` (проект Ch03\Locate\Nakls.dpr):

```
procedure TfmNakls.DateTimePicker1Change(Sender: TObject);
begin
  if Nakls.Locate('NDate',Variant(DateTimePicker1.Date),[]) then
    DBGrid1.SetFocus
end;
```

Теперь при вводе в компонент нужной даты программа отыщет в наборе `Nakls` первую накладную с этой датой (рис. 3.7).

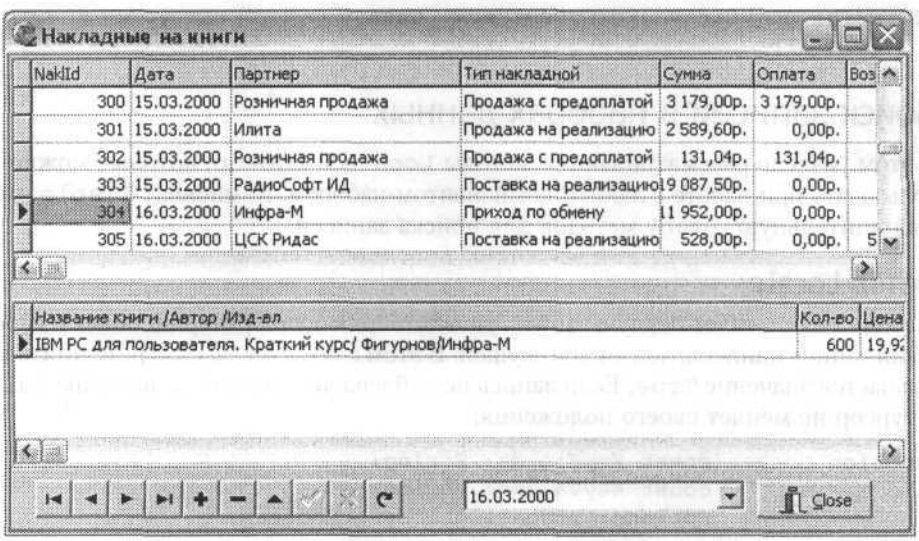


Рис. 3.7. Поиск накладных по дате

Метод Lookup

Метод Lookup находит запись, удовлетворяющую условию поиска, но не делает ее текущей, а возвращает значения некоторых ее полей. Независимо от результата поиска записи указатель текущей записи в НД не изменяется. В отличие от метода Locate, метод Lookup осуществляет поиск только на точное соответствие критерию поиска искомых значений поля записи:

```
function Lookup(const KeyFields: String;
                const KeyValues: Variant;
                const ResultFields: String): Variant;
```

В параметре KeyFields указывается список полей, по которым необходимо осуществить поиск (если задается поиск по нескольким полям, соседние поля в списке разделяются точкой с запятой). Параметр KeyValues определяет искомые значения полей, список которых содержится в параметре KeyFields. В параметре ResultFields перечисляются поля, значения которых требуется получить в случае успешного поиска. Тип результата — Variant или вариантный массив.

Если имеется несколько поисковых полей, каждому *i*-му полю в списке KeyFields ставится в соответствие *i*-е значение в списке KeyValues. Если поиск ведется по одному полю, его поисковое значение можно указывать в качестве параметра KeyValues непосредственно; в случае нескольких полей их необходимо приводить к типу вариантного массива при помощи функции преобразования VarArrayOf.

Как и в методе Locate, в качестве искомых полей можно указывать *поля* как входящие в какой-либо индекс, так и не входящие в него. Если в результате поиска запись не найдена, метод Lookup возвращает Null, что можно проверить с помощью такого приема:

```
var
  LookupResult: Variant;
...
  LookupResult := Table1.Lookup(...);
  if VarType(LookupResult) = varNull then ...
```

Если поиск оказался успешным, Lookup возвращает из найденной записи значения полей, список которых содержит ResultFields. При этом размерность результата зависит от того, сколько результирующих полей указано в ResultFields.

Пусть, например, в НД Nakls ищется запись, ключевое поле NaklID которой имеет значение, задаваемое текстом в поле seNaklID; в качестве результирующих пусть будут поля NDate, Firm, NSum и NPayedSum. Чтобы получить результат, показанный на рис. 3.8, использовался обработчик щелчка на кнопке Искать, представленный в листинге 3.3 (проект Ch03\Lookup\Lookup.dpr).

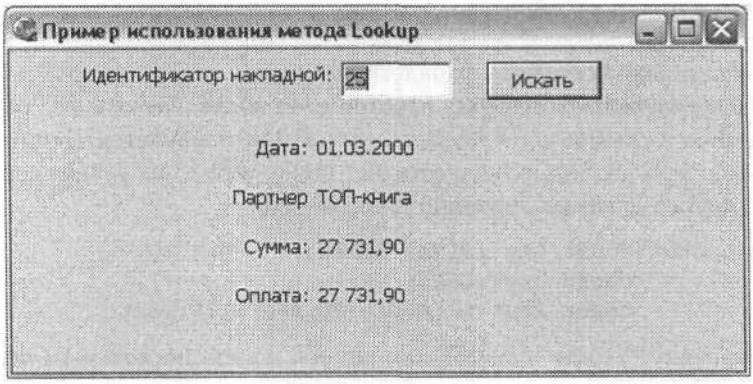


Рис. 3.8. Окно с отображением результата поиска методом Lookup

Листинг 3.3. Обработчики щелчка на кнопке Искать и изменения текста

```
var
  NaklID: Integer = 1;

procedure TForm1.Button1Click(Sender: TObject);
var
  LookupResult: Variant;
begin
  LookupResult := Nakls.Lookup('NaklID', NaklID,
                               'NDate;Name;NSum;NPayedSum');
  if VarType(LookupResult)=varNull then
    ShowMessage('Нет накладной с номером '+
                IntToStr(NaklID))
  else if VarType(LookupResult)=varEmpty then
    ShowMessage('Поиск не проведен')
  else if VarIsArray(LookupResult) then
  begin
    stDate.Caption := LookupResult[0];
    stName.Caption := LookupResult[1];
    stSum.Caption := FloatToStrF(LookupResult[2] as Double,
                                 ffNumber, 10, 2);
    stPayed.Caption := FloatToStrF(LookupResult[3] as Double,
                                   ffNumber, 10, 2);
    edNaklID.SelectAll;
  end else
    stDate.Caption := LookupResult
end;

var
  Txt: String = '1';
procedure TForm1.edNaklIDChange(Sender: TObject);
// Блокирует ввод текста, если нет правильного числа
begin
  if Tag = 0 then  // Пропускаем проверку после восстановления
```

```
  try
    NaklID := StrToInt(edNaklID.Text);  // Пытаемся преобразовать
    Txt := edNaklID.Text;  // Все в порядке: запоминаем текст
  except                   // Ошибка!
    edNaklID.Text := Txt;  // Восстанавливаем текст
    // и устанавливаем курсор на место ошибочного символа
    edNaklID.SelStart := Length(Txt);
  end;
end;
```

В примере для ввода номера накладной используется поле `edNaklID`. Обработчик изменения текста компонента блокирует очередное изменение, если текст не соответствует правильному текстовому представлению целого числа. Если преобразование текста прошло удачно, новый текст запоминается в глобальной переменной `Txt`, в противном случае содержимое этой переменной используется для восстановления предыдущего текста.

Фильтрация записей

Помимо описываемых далее средств для фильтрации данных могут использоваться методы `SetRange`, `ApplyRange` (и сопутствующие им методы) в компоненте `TTable` и секция **WHERE** SQL-запроса в компонентах `TQuery` и `TStoredProc`.

Свойство Filter

Свойство `Filter` задает критерий фильтрации. В этом случае НД окажется отфильтрованным, как только в его свойство `Filtered` будет помещено значение `True`. Синтаксис описания критерия похож на синтаксис секции **WHERE** SQL-запроса с тем исключением, что имена переменных программы указывать нельзя, а можно указывать имена полей и литералы (явно заданные значения). Допустимы обычные операции отношения и логические операции **AND**, **NOT** и **OR**, например:

```
(BOpt>100) AND (BQuan=0)
```

Строку критерия фильтрации можно ввести во время прогона программы или на этапе конструирования формы. Например, с помощью представленного в листинге 3.4 обработчика события `OnClick` компонента `sbFilter` (класс `TSpeedButton`) критерий фильтрации считывается из поля `edFilter` и помещается в свойство `Filter` компонента `Nakls` (проект Ch03\Filter\Nakls.dpr, рис. 3.9 и 3.10).

Листинг 3.4. Обработчик события OnClick компонента sbFilter

```
procedure TfmNakls.sbFilterClick(Sender: TObject);
begin
  if edFilter.Text <> '' then
  with Nakls do
  begin
    Filter := edFilter.Text;
    try
      Filtered := True;
```

продолжение

Листинг 3.4 (продолжение)

```
    except
      Filtered := False;
    end;
  end;
end;
```

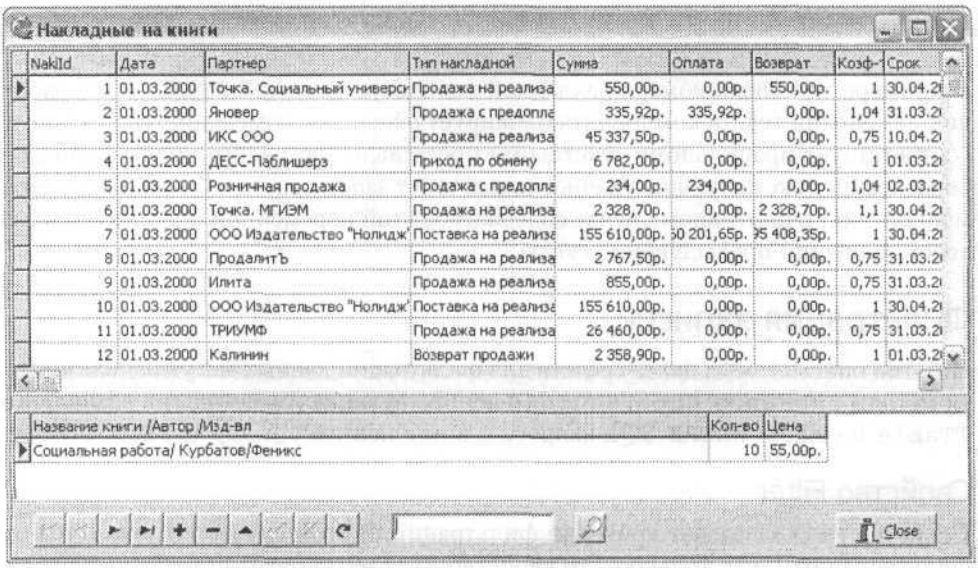


Рис. 3.9. Набор данных до фильтрации

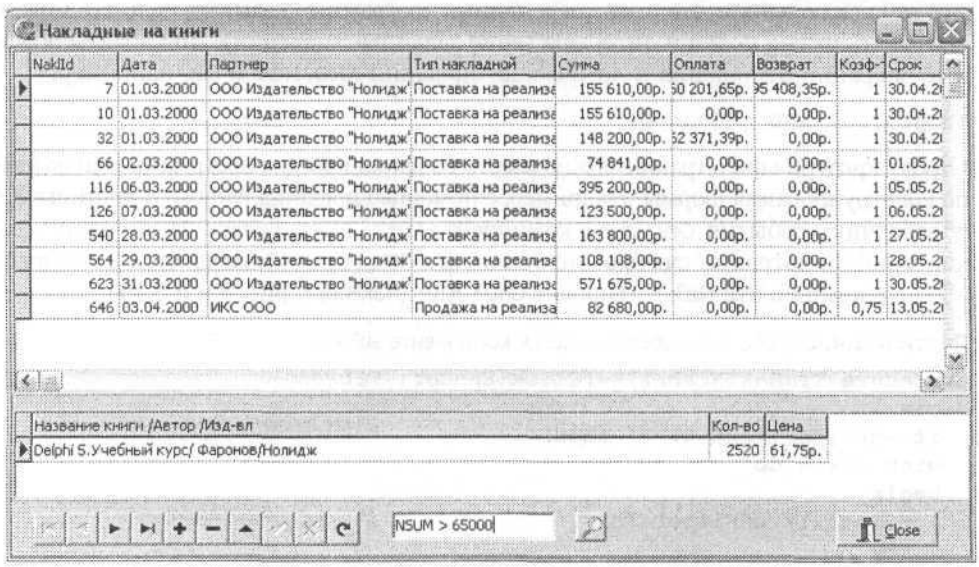


Рис. 3.10. Набор данных после фильтрации

С помощью следующего свойства программист может определить дополнительные условия фильтрации текстовых полей:

```
type TFilterOption = (foCaseInsensitive, foNoPartialCompare);
property FilterOptions: TFilterOptions;
```

Параметры:

- `foCaseInsensitive` — фильтрация производится без учета регистра букв;
- `foNoPartialCompare` — поиск производится на точное соответствие.

Событие OnFilterRecord

Событие `OnFilterRecord` возникает при установке значения `True` в свойство `Filtered`. Обработчик события имеет два параметра: имя фильтруемого набора данных и переменную `Accept`, в которую программа должна поместить значение `True`, если текущая запись удовлетворяет критерию фильтрации.

В отличие от критерия в строке `Filter`, ограниченного рамками синтаксиса условного выражения, критерий, реализуемый в обработчике события `OnFilterRecord`, определяется синтаксисом Delphi и позволяет получать сложные алгоритмы фильтрации. Однако следует помнить о том, что в обработчике `OnFilterRecord` последовательно перебираются все записи таблицы БД, в то время как методы `SetRange`, `ApplyRange` и им сопутствующие методы компонента `TTable` используют индексно-последовательный метод доступа, то есть работают с частью записей в физической таблице БД. Это делает обработчик `OnFilterRecord` предпочтительным для фильтрации небольших объемов записей и сильно ограничивает его применение при больших объемах.

Блокировка таблиц в многопользовательском режиме

При работе с таблицами файл-серверных СУБД в многопользовательском режиме может возникнуть ситуация, когда на момент внесения изменений в какую-либо таблицу БД одним пользователем следует блокировать внесение изменений в ту же таблицу другими пользователями. Более жестким ограничением может служить запрет на просмотр содержимого таблицы со стороны других пользователей.

Запрет достигается с помощью метода

```
procedure LockTable(LockType: TLockType);
```

Здесь параметр `LockType` определяет вид запрета: `ltReadLock` — запретить чтение; `ltWriteLock` — запретить запись. Можно запретить и чтение и запись данных, но для этого нужно вызвать метод `LockTable` дважды.

Попытка внесения изменений в НД, ассоциированный с таблицей БД, для которой запрещена запись данных, равно как и попытка чтения из запрещенной для чтения таблицы, приводят к возбуждению исключительной ситуации `EDBEngineError` со следующим сообщением:

```
Table имя_таблицы is locked
```

После того как необходимость запрета отпадет, он должен быть отменен методом

```
procedure UnlockTable(LockType: TLockType);
```

Здесь параметр `LockType` определяет тип снимаемого запрета.

Обзор событий

Все НД получают в свое распоряжение большое количество событий, унаследованных ими от родительского класса `TBDEDataSet`. Далее рассматриваются наиболее важные из них.

Реализация каскадных изменений и бизнес-правил

Для реализации каскадных изменений и бизнес-правил в файл-серверных БД обычно используются обработчики событий `After`*XXXX* и `Before`*XXXX*. Эти две группы событий связаны с изменениями НД (вставка, удаление или редактирование записи) и отличаются тем, что события `Before`*XXXX* наступают непосредственно *перед* изменением НД, а события `After`*XXXX* — сразу *после* изменения. К ним относятся:

- **property** `After/BeforeCancel` — возникает после/до отмены изменений в текущей записи;
- **property** `After/BeforeClose` — возникает после/до закрытия НД;
- **property** `After/BeforeDelete` — возникает после/до удаления текущей записи;
- **property** `After/BeforeEdit` — возникает после/до редактирования текущей записи;
- **property** `After/BeforeInsert` — возникает после/до вставки новой записи;
- **property** `After/BeforeOpen` — возникает после/до открытия НД;
- **property** `After/BeforePost` — возникает после/до выполнения метода `Post`;
- **property** `After/BeforeScroll` — возникает после/до перехода к новой записи.

Обработчики этих событий в качестве параметра получают ссылку на НД, вызвавший соответствующее событие. Если необходимо прервать работу обработчика и предотвратить изменение ТБД, нужно возбудить исключение или вызвать процедуру `Abort`.

Другие события

Событие `OnCalcFields` возникает, когда программа должна сформировать значения для вычисляемых полей. Следует заметить, что это событие возникает в момент открытия НД и при любом его изменении. Если пользователь интенсивно работает с НД, а алгоритм получения вычисляемых значений достаточно сложен, обработчик этого события может снизить производительность системы. Событие `OnCalcFields` в этом случае следует предварительно отключать, устанавливая значение `False` в свойство НД `AutoCalcFields`.

Следующее событие возникает в момент фильтрации записей:

```
type TFilterRecordEvent: procedure(DataSet: TDataSet;
var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

Обработчик должен поместить в параметр Accept значение True, если запись удовлетворяет условиям фильтрации.

Событие OnNewRecord возникает при вставке новой записи. В его обработчике программа может поместить в запись значения полей, заданные по умолчанию.

Таблицы

Компонент TTable (таблица), которому посвящен этот раздел, является одним из наследников класса TDBDataSet (набор данных), свойства, методы и события которого рассмотрены в предыдущем разделе.

Обзор свойств и методов

Здесь описываются наиболее важные специфические для класса TTable свойства и методы (его события унаследованы от класса TDBDataSet и здесь не рассматриваются).

Свойства

Свойства компонента TTable представлены в табл. 3.22.

Таблица 3.22. Свойства компонента TTable

Свойство	Описание
`property DataSource: TDataSource;`	Содержит имя главного НД в отношении главный–детальный (только для чтения)
`property DefaultIndex: Boolean;`	Если содержит True, НД в момент открытия будет отсортирован по умалчиваемому индексу
`property Exclusive: Boolean;`	Если содержит True, после открытия таблицы запрещает работу с ней другим приложениям
`property Exists: Boolean;`	Указывает, существует ли нужная таблица БД
`property IndexDefs: TIndexDefs;`	Содержит информацию об индексах таблицы
`property IndexFieldCount: Integer;`	Указывает количество полей, которые образуют текущий ключ
`property IndexFieldNames: String;`	Содержит список полей, образующих текущий ключ
`property IndexName: String;`	Содержит имя используемого индекса

продолжение

Таблица 3.22 (продолжение)

Свойство	Описание
`property KeyExclusive: Boolean;`	Если содержит `False`, граничные точки диапазона при фильтрации по ключам входят в поисковый диапазон, в противном случае — не входят
`property MasterFields: String;`	Содержит список полей главной таблицы, по которым в данной таблице будет установлена связь главная–детальная
`property MasterSource: TDataSource;`	Определяет имя главной таблицы в связи главная–детальная
`property ReadOnly: Boolean;`	Если содержит `True`, ТБД используется только для чтения данных
`property TableName: TFileName;`	Содержит имя таблицы
`Type TTableType = (ttDefault, ttParadox, ttDBase, ttASCII, ttFoxPro);` `property TableType: TTableType;`	Определяет тип таблицы: `ttDefault` — тип определяется расширением табличного файла; `ttParadox` — таблица Paradox; `ttDBase` — таблица dBASE; `ttASCII` — текстовая таблица; `ttFoxPro` — таблица FoxPro

Методы

Методы компонента `TTable` перечислены в табл. 3.23.

Таблица 3.23. *Методы компонента TTable*

Метод	Описание

`type TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive, ixExpression, ixNonMaintained);`
`TIndexOptions = set of TIndexOption;`
`procedure AddIndex(const Name, Fields: String; Options: TIndexOptions);`
	Создает новый индекс: `Name` — имя индекса; `Fields` — список индексных полей; `Options` — параметры индекса (см. далее)

`procedure ApplyRange;`	Включает механизм фильтрации записей по ключевым полям
`type TBatchMode = (batAppend, batUpdate, batAppendUpdate, batDelete, batCopy);` `function BatchMove(ASource: TBDEDataSet; AMpde: TBatchMode): Integer;`	Копирует записи из НД `ASource` в текущую таблицу и возвращает количество скопированных записей. Параметр `AMode` определяет режим копирования (см. далее)

Метод	Описание
`procedure CancelRange;`	Отменяет фильтрацию записей по ключевым полям
`procedure CloseIndexFile(const IndexFile: String);`	Закрывает идексный файл `IndexFile`
`procedure CreateTable;`	Создает новую таблицу по информации, хранящейся в свойствах `FieldDefs`, `Fields` и `IndexDefs`
`procedure DeleteIndex(const Name: String);`	Удаляет индекс с именем `Name`
`procedure DeleteTable;`	Удаляет таблицу
`procedure EditKey;`	Переводит таблицу в режим `dsSetKey` и позволяет модифицировать ключи поиска
`procedure EditRangeEnd;`	Позволяет изменить конечное значение диапазона фильтрации по индексным полям
`procedure EditRangeStart;`	Позволяет изменить начальное значение диапазона фильтрации по индексным полям
`procedure EmptyTable;`	Удаляет все записи из таблицы
`function FindKey(const KeyValues: array of const): Boolean;`	Отыскивает запись по указанным значениям `KeyValues` ключевых полей и возвращает `True` в случае успеха
`procedure FindNearest(const KeyValues: array of const);`	Перемещает курсор на запись, наиболее полно удовлетворяющую указанным значениям `KeyValues` ключевых полей
`procedure GetDatailLinkFields (MaterFields, DetailFields: TObjectList);`	Создает два списка ключевых полей: для главной таблицы `MasterFields` и для детальной — `DetailFields`
`procedure GetIndexNames(List: TStrings);`	Наполняет список `List` именами ключевых полей
`procedure GetProviderAttributes (List: TObjectIlst);`	Получает из BDE и помещает в список `List` языковой драйвер, маршрут доступа и имя таблицы
`procedure GotoCurrent(Table: TTable);`	Устанавливает курсор текущего НД в положение курсора НД `Table`, основанного на той же физической таблице
`function GotoKey: Boolean;`	Пытается отыскать запись, ключевые поля которой соответствуют значениям, связанным с предыдущими вызовами методов `StartKey` или `EditKey`. Перемещает курсор на найденную запись и возвращает `True` в случае успеха
`procedure GotoNearest;`	Перемещает курсор на запись, наиболее полно удовлетворяющую значениям ключей, установленных методами `SetKey` или `EditKey`

продолжение

Таблица 3.23 (продолжение)

Метод	Описание
type TLockType = (ltReadLock, ltWriteLock);	Блокирует таблицу, ограничивая доступ к ней другим пользователям: ltReadLock — блокировать запись и чтение; ltWriteLock — блокировать только запись
procedure LockTable(LockType: TLockType);	
procedure OpenIndexFile (IndexFile: **String**);	Открывает индексный файл IndexFile
procedure RenameTable(**const** NewName: **String**);	Переименовывает таблицу
procedure SetKey;	Переводит таблицу в режим dsSetKey и очищает буфер ключей. С помощью метода FieldByName (см. табл. 3.15) программа должна установить новый набор ключей и с помощью метода ApplayRange отфильтровать НД
procedure SetRange(**const** StartValues, EndValues: **array of const**);	Переводит таблицу в режим dsSetKey, устанавливает новый набор начальных (StartValues) и конечных (EndValues) значений буфера ключей и фильтрует НД по индексным ключам
procedure SetRangeEnd;	Переводит НД в режим dsSetKey и очищает буфер конечных значений ключей
procedure SetRangeStart;	Переводит НД в режим dsSetKey и очищает буфер начальных значений ключей
procedure UnlockTable(LockType: TLockType);	Отменяет блокировку таблицы (см. описание метода LockTable)

Аргумент Options метода AddIndex определяет параметры индекса, в том числе:

- ixPrimary — первичный индекс;
- ixUnique — уникальный индекс;
- ixDescending — индекс, сортирующий по убыванию значений;
- ixCaseInsensitive — индекс, чувствительный к регистру букв в текстовых полях;
- ixExpression — индекс, определенный по индексному выражению;
- ixNonMaintained — индекс не обновляется автоматически в момент открытия таблицы.

Параметр AMode метода BatchMove определяет режим копирования:

- batAppend — добавить все записи в конец таблицы;
- batUpdate — обновить записи с одинаковыми ключевыми полями;
- batAppendUpdate — обновить записи с одинаковыми ключевыми полями и добавить остальные записи в конец таблицы;

- batDelete — удалить записи с одинаковыми ключевыми полями;
- batCopy — удалить все записи текущей таблицы и заменить их записями копируемой таблицы.

Индексы

Смена текущего индекса

Возможность менять текущий индекс и таким способом влиять на сортировку табличных данных является отличительной особенностью компонента TTable. В двух других компонентах-наборах программист не может явно указать индекс, а сортировка данных осуществляется в секции **ORDER BY** SQL-запроса.

Для смены текущего индекса предназначены свойства IndexName и IndexFieldNames. В первое нужно поместить имя индекса, во второе — список индексных полей, входящий в состав индекса. Эти свойства взаимоисключающие: установка значения в одно из них очищает другое. Если НД открыт, смена текущего индекса приводит к немедленному изменению порядка следования записей в таблице.

Добавление нового индекса

Добавление нового индекса происходит в режиме эксклюзивного доступа к таблице БД (свойство Exclusive = True) и осуществляется следующим методом:

```
procedure AddIndex(const Name, Fields: String;
                   Options: TIndexOptions)
```

Здесь параметр Name определяет имя индекса, а параметр Fields — список индексных полей. В случае нескольких полей соседние имена в списке разделяет точка с запятой. Должны указываться только поля, объявленные в структуре таблицы БД, в противном случае будет возбуждена исключительная ситуация. Параметр Options является множеством, которое содержит значения, определяющие свойства индекса:

- ixCaseInsensitive — индекс чувствителен к регистру букв;
- ixDescending — индексные поля сортируются по убыванию значений;
- ixPrimary — создается первичный ключ;
- ixUnique — значения полей в индексе должны однозначно определять запись.

Удаление индекса

Удаление существующего индекса происходит в режиме эксклюзивного доступа к таблице БД и осуществляется следующим методом:

```
procedure DeleteIndex(const Name: String)
```

Здесь параметр Name определяет имя удаляемого индекса. При попытке удаления несуществующего индекса возбуждается исключительная ситуация.

Составные индексы

Составными называются индексы, построенные по значениям двух и более полей. Такие индексы широко используются на практике для ускорения поиска нужной записи или группы записей методами `FindKey` и `FindNearest` (см. далее).

Если в таблице построен индекс только по полю `Field1`, а требуется отыскать записи, содержащие нужные значения в полях `Field1` и `Field2`, то методы `FindKey` и `FindNearest` могут установить курсор лишь в начало целой группы записей, поля `Field1` в которых имеют нужные значения, но различаются значениями остальных полей. В этом случае для поиска записей с нужными значениями поля `Field2` придется последовательно просматривать таблицу, начиная с найденной записи. Если построен составной индекс по полям `Field1` и `Field2`, методы `FindKey` и `FindNearest` сразу позиционируют курсор на нужную запись и для таблиц большого размера могут существенно сократить время поиска. Например, в моей практике был случай, когда в таблице, насчитывающей более 800 000 записей, нужно было отыскать группу записей с заданными значениями трех полей. И хотя по каждому полю в отдельности был построен индекс, поиск проходил с заметными задержками (5–6 с), которые вызывали у пользователей вполне оправданное раздражение. После построения составного индекса время поиска сократилось до долей секунды, и пользователи просто перестали замечать задержки.

Для создания составного индекса в серверных таблицах и таблицах типа Paradox с помощью утилиты SQL Explorer используется SQL-запрос типа

```
CREATE INDEX IndexName ON TableName(Field1, Field2, ...)
```

В широко распространенных таблицах типа dBASE нельзя создать составной индекс и, следовательно, реализовать быстрый индексный поиск при участии нескольких полей. В базирующихся на них средствах разработки (dBASE III+, dBASE IV, FoxPro, Clipper) для этих целей обычно используется индекс, построенный по индексному выражению, с приведением нужных полей к строковому типу. Например:

```
STR(FINCODE,5)+LEFT(FINTYPE,1)+DTOS(FINDATE)
```

Это индексное выражение формирует строку из 5 символов преобразования целого значения поля `FINCODE`, первого символа значения строкового поля `FINTYPE` и преобразованного к строке значения даты из поля `FINDATE`.

Увы! Delphi не имеет средств поиска по индексным выражениям. Таким образом, использование таблиц dBASE в приложениях Delphi менее удобно, чем таблиц Paradox: если вы собираетесь разработать приложение для пока еще не созданной БД и выбрали традиционную архитектуру файл-серверных таблиц, обратите на это внимание; если вы создаете приложение для уже существующей и наполненной БД, в которой используются таблицы dBASE, имеет смысл подумать о передаче содержащихся в них данных в таблицы Paradox с помощью компонентов `TBatchMove`.

Эксклюзивный доступ к таблице

Свойство `Exclusive` дает программе эксклюзивный доступ к таблице — в этом случае в свойство нужно поместить значение `True`. Это означает, что никто, кроме вашей программы, не только не сможет вносить изменения в таблицу БД, но вообще не получит к ней доступа. Установить эксклюзивный доступ можно лишь тогда, когда ни один пользователь не имеет доступа к таблице БД и та не открыта.

ПРИМЕЧАНИЕ

Среда Delphi тоже считается пользователем, если на этапе конструирования формы компонент `TTable` активизирован (его свойство `Active` содержит `True`). Поэтому, если в программном коде делается попытка получения прав эксклюзивного доступа к таблице БД, а программа запущена из среды Delphi, эта попытка будет блокирована, поскольку на вашей машине имеется два пользователя, осуществляющих доступ к этой таблице БД: выполняющееся приложение и среда Delphi. Если ваша программа должна обращаться к методам, требующим эксклюзивных прав доступа, нужно закрыть таблицу на этапе конструирования и открывать ее программно, например, в обработчике события `OnCreate` главной формы.

Удаление записей и таблиц

Метод `EmptyTable` удаляет все записи в таблице БД, связанной с данным НД. После этой операции таблица БД будет пустой. Метод применим только к закрытым НД и только в режиме эксклюзивного доступа (см. ранее). В противном случае возбуждается исключение и очистка таблицы БД блокируется.

С помощью процедуры `DeleteTable` таблица удаляется физически. Метод применим только к закрытым НД, но для его выполнения не нужно блокировать доступ к таблице с помощью свойства `Exclusive`.

Поиск записей в таблице

Характерной особенностью компонента `TTable` является реализованная в нем возможность работы с индексами. Эта особенность проявляется также и в поиске записей, где помимо «безындексных» методов `Locate` и `Lookup`, которые компонент `TTable` унаследовал от `TDBDataSet`, таблица имеет два собственных метода, использующих текущий индекс.

Точный поиск

Для точного поиска (поиска на точное соответствие) применяется метод `FindKey`. Он пытается отыскать в НД запись, у которой индексные поля соответствуют значениям, указанным в параметре обращения. Если такая запись найдена, метод `FindKey` возвращает `True`, и указатель текущей записи в НД (курсор НД) устанавливается на эту запись, то есть она делается текущей. Если найдена группа записей, отвечающих условию, текущей становится логически первая из них. Если запись не найдена, курсор НД не перемещается и метод возвращает `False`.

Неточный поиск

Неточный поиск (поиск на неточное, то есть приблизительное, соответствие) осуществляется методом FindNearest. Он пытается отыскать в НД запись, у которой индексные поля соответствуют указанным значениям. Если такая запись найдена, указатель текущей записи в НД перемещается на нее или на следующую за ней запись в зависимости от значения свойства KeyExclusive. Если KeyExclusive = False (по умолчанию), указатель текущей записи перемещается на нее. Если KeyExclusive = True, указатель текущей записи перемещается на следующую запись.

Если запись не найдена, указатель текущей записи перемещается на ближайшую запись с бóльшим значением индекса.

Выборка записей

В большинстве практически важных случаев НД должен содержать не все записи таблицы БД, а лишь некоторую их часть, соответствующую условиям отбора. Как и в случае поиска записей, компонент TTable помимо описанных в предыдущем разделе свойств Filter, Filtered и события OnFilterRecord имеет собственные методы отбора, основанные на использовании текущего индекса.

Следующий метод показывает в НД только те записи, индексные поля которых лежат в диапазоне от StartValues до EndValues:

```
procedure SetRange(const StartValues, EndValues: array of const)
```

Например, представленный ниже обработчик покажет все накладные за один день (если вы захотите использовать такой обработчик, предварительно установите в свойство Nakls.KeyFieldNames имя индексного поля NDate):

```
procedure TfmNakls.DateTimePicker1Change(Sender: TObject);
begin
  Nakls.SetRange([DateTimePicker1.Date],
                 [DateTimePicker1.Date]);
end;
```

Вместо единственного метода SetRange можно использовать целых три: SetRangeStart — для установки начальной границы диапазона фильтрации; SetRangeEnd — для установки конечной границы и ApplyRange — для осуществления фильтрации:

```
procedure TForm1. DateTimePicker1Change(Sender: TObject);
begin
  with Nakls do
  begin
    SetRangeStart;
    NaklsNDate.Value := DateTimePicker1.Date;
    SetRangeEnd;
    NaklsNDate.Value := DateTimePicker1.Date;
    ApplyRange;
  end; {with}
end;
```

В этом варианте вы можете обратиться к свойству KeyExclusive, чтобы указать, надо ли включать то или иное граничное значение в выборку (False — включать). По умолчанию для этого свойства используется значение False, поэтому два предыдущих варианта формировали идентичные выборки. Однако в следующем варианте таблица Nakls всегда будет пустой:

```
procedure TForm1. DateTimePicker1Change(Sender: TObject);
begin
  with Nakls do
  begin
    SetRangeStart;
    KeyExclusive := True;
    NaklsNDate.Value := DateTimePicker1.Date;
    SetRangeEnd;
    KeyExclusive := True;
    NaklsNDate.Value := DateTimePicker1.Date;
    ApplyRange;
  end; // with DM,Nakls
end;
```

Запросы

Характерной особенностью компонента TQuery (запрос) является использование в нем специального языка для работы с реляционными БД — SQL (Structured Query Language — язык структурированных запросов)[1]. С помощью этого языка программист составляет SQL-запрос, который помещается в специально для этого предназначенное свойство TQuery.SQL. После вызова метода Open или ExecSQL компонента TQuery запрос передается машине баз данных (BDE). Последняя имеет встроенный интерпретатор SQL, позволяющий ей выполнять описанные в запросе действия. Если источником данных является серверная БД, BDE передает запрос серверу, в противном случае выполняет его сама. Таким образом, преимуществом технологии BDE.NET является возможность обращаться с запросами к данным, размещенным в файл-серверной БД.

Если запрос требует получения из БД нужных сведений (запрос **SELECT**), сформированные с помощью BDE данные помещаются в локальную таблицу в виде временного файла в каталоге запуска программы и TQuery становится владельцем этой таблицы. Данные из временной таблицы через компонент-посредник TDataSource передаются визуальным компонентам и отображаются в них точно так же, как если бы они были получены компонентом TTable. Однако, в отличие от TTable, пользователь не может их изменять, так как они представляют собой лишь копию реальных данных. Для изменения хранящейся в БД информации формируются специальные запросы (INSERT, UPDATE, DELETE). В этом случае

[1] Основные сведения о языке SQL приведены в приложении Б.

BDE не формирует новые и никак не использует ранее созданные временные таблицы, но лишь интерпретирует запрос и уведомляет программу о том, насколько успешно прошло его выполнение. Таким образом, необходимость программного изменения запроса в случае модификации НД является другим характерным отличием компонентов TQuery от TTable.

Следует оговориться, что при некоторых ограничениях на SQL-запрос компонент TQuery может создать «живой» НД и вносимые в него изменения BDE будет немедленно переносить в таблицу БД точно так же, как это делает компонент TTable.

Подведя итог, нужно сказать, что при работе с локальными или файл-серверными БД скорость доступа к данным у TQuery в общем случае меньше, чем у TTable, так как для своей работы TQuery создает временные таблицы (в файл-серверных БД TTable этого не делает). С другой стороны, широкие возможности SQL позволяют с помощью компонентов TQuery получать НД, которые невозможно получить с помощью TTable (например, объединение в одном НД данных из нескольких таблиц БД). При работе с серверными БД компонент TTable теряет всякие преимущества, так как в этом случае он также создает временную таблицу, являющуюся локальной копией *всей* серверной таблицы БД, а уже затем формирует из нее нужный НД. Затраты времени на создание больших локальных таблиц и значительно более скромные возможности компонента TTable в отношении получения сложных НД практически исключают его использование в клиент-серверных приложениях.

Компонент TQuery порожден от родительского класса TDBDataSet и наследует от него основные методы, события и свойства. Он имеет также собственные свойства, методы и события, наиболее важные из которых рассматриваются в этом разделе.

Обзор свойств

Свойства компонента TQuery представлены в табл. 3.24.

Таблица 3.24. Свойства компонента TQuery

Свойство	Описание
`property Constrained: Boolean;`	Если содержит True, в изменяемом («живом») НД на записи, вводимые или изменяемые пользователем, накладываются ограничения секции **WHERE** оператора **SELECT**
`property DataSource: TDataSource;`	Содержит ссылку на компонент TDataSource, используемый для формирования параметрического запроса
`property Local: Boolean;`	Содержит True, если TQuery работает с локальной или файл-серверной БД

Свойство	Описание
`property ParamCheck: Boolean;`	Если содержит `True`, список параметров будет автоматически обновляться при изменении запроса на этапе прогона программы
`property Params[Index: Word]: TParams;`	Содержит массив объектов-параметров класса `TParams` (см. далее)
`property Prepared: Boolean;`	Содержит `True`, если запрос был подготовлен к выполнению методом `Prepare`
`property RequestLive: Boolean;`	Содержит `True`, если `TQuery` будет возвращать изменяемый («живой») НД
`property RowsAffected: Integer;`	Содержит количество записей, которые были изменены или удалены в результате выполнения запроса
`property SQL: TStrings;`	Содержит текст SQL-запроса
`property Text: PChar;`	Содержит текст SQL-запроса, который был в действительности передан BDE
`property UniDirectional: Boolean;`	Если содержит `True`, курсор НД может перемещаться только вперед. Такие НД требуют меньше памяти и быстрее обрабатываются (см. примечание далее)

ПРИМЕЧАНИЕ

Свойство `UniDirectional` используется в клиент-серверных базах данных для серверов, которые не поддерживают двунаправленные курсоры. В этом случае значение `False`, заданное по умолчанию, требует от Delphi имитировать двунаправленный курсор за счет создания на машине клиента буфера записей. Если НД предполагается просматривать только в одном направлении — от первой записи к последней, — в свойство следует поместить значение `True`. Для файл-серверных БД это свойство игнорируется, так как BDE всегда поддерживает двунаправленные курсоры.

Свойство `Params` содержит набор ссылок на объекты класса `TParams`. Наиболее важные свойства и методы этого класса перечислены в табл. 3.25 и 3.26.

Таблица 3.25. Свойства класса TParams

Свойство	Описание
`property Items[Index: Word]: TParam;`	Содержит массив объектов-параметров класса `TParam` (см. табл. 3.27). Это свойство для класса `TParams` является умалчиваемым. Значение параметра `Index` должно лежать в диапазоне от 0 до `Count-1`
`property ParamValues[const ParamName: String]: Variant;`	Открывает доступ к значению параметра по его имени `ParamName`
`property Count: Integer;`	Содержит количество параметров в массиве `Items`

Таблица 3.26. Методы класса TParams

Метод	Описание
`procedure AddParam(Value: TParam);`	Добавляет объект-параметр к массиву Items
`type TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult);` `function CreateParam(FldType: TFieldType; const ParamName: String; ParamType: TParamType): TParam;`	Создает объект-параметр и добавляет его к массиву Items. Тип параметра ParamType: ptUnknown — не определен; перед передачей в сохраняемую процедуру этот тип должен быть заменен другим; ptInput — предназначен для передачи значения в запрос; ptOutput — предназначен для получения значения из запроса; ptResult — определяет результат запроса; в Items может быть только один параметр этого типа
`function FindParam(const Value: String): TParam;`	Ищет объект-параметр по его имени Value
`procedure RemoveParam(Value: TParam);`	Удаляет объект-параметр Value из списка Items

В табл. 3.27 перечислены наиболее важные свойства класса TParam (см. описание свойства Items в табл. 3.25).

Таблица 3.27. Свойства класса TParam

Свойство	Описание
`property As`*XXXX*`: `*XXXX*`;`	Эти свойства (AsString, AsInteger и т. д.) служат для преобразования значения параметра к нужному типу
`property IsNull: Boolean;`	Содержит True, если с параметром не связано значение
`property Name: String;`	Содержит имя параметра
`property Value: Variant;`	Содержит значение параметра

Обзор методов

Наиболее важные методы компонента TQuery представлены в табл. 3.28.

Таблица 3.28. Методы компонента TQuery

Метод	Описание
`procedure ExecSQL;`	Выполняет запросы INSERT, UPDATE, DELETE и CREATE TABLE. Для выполнения запроса **SELECT** вместо метода ExecSQL используется метод Open или свойство Active
`procedure GetDetailLinkFields (MasterFields, DetailFields: TList); override;`	Заполняет список MasterFields полями главной таблицы и список DetailFields полями детальной таблицы

Метод	Описание
`function ParamByName(const Value: String): TParam;`	Открывает доступ к параметру по его имени `Value`
`procedure Prepare;`	Передает BDE запрос для того, чтобы сама библиотека BDE и удаленный сервер БД распределили свои ресурсы и дополнительно оптимизировали запрос
`procedure UnPrepare;`	Отменяет последствия вызова метода `Prepare`

Свойство SQL

Следующее свойство является коллекцией строк, в которую на этапе разработки или прогона программы помещается текст запроса:

```
property SQL: TStrings
```

Если текст размещен на этапе конструирования, для такого НД можно создать объекты-поля. При любом изменении свойства компонент автоматически переходит в неактивное состояние (его свойство `Active` получает значение `False`). После изменения текста компонент следует открыть заново.

Для введения текста запроса на этапе разработки нужно раскрыть и использовать связанный с этим свойством многострочный текстовый редактор (рис. 3.11).

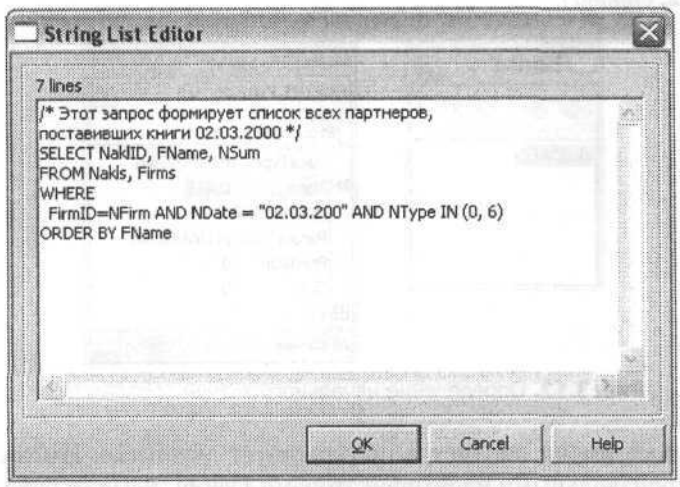


Рис. 3.11. Окно редактора свойства SQL с текстом запроса

Методы Open и ExecSQL

В отличие от таблицы `TTable`, компонент `TQuery` имеет два метода для выполнения запроса. Метод `Open` используется в запросах **`SELECT`**, возвращающих НД. Все остальные запросы (**`CREATE`**, **`DROP`**, **`INSERT`** и т. п.) не возвращают данные, поэтому для их исполнения необходимо обращаться к методу `ExecSQL`.

Параметрические запросы

Важной особенностью компонента TQuery является возможность создания и использования так называемых параметрических запросов. В такого рода запросах группа параметров позволяет уточнить запрос. Параметры меняются в процессе прогона программы и в общем случае порождают одинаковые по структуре, но разные по содержанию НД.

Пусть, например, нужно создать запрос, который на этапе прогона возвращал бы НД со списком партнеров, поставивших книги в определенный день. В этом случае вместо статического (не меняющегося) запроса следует использовать такой параметрический запрос:

```
/* Этот запрос формирует список всех партнеров,
   поставивших книги в день, определяемый параметром DATE */
SELECT NaklID, FName, NSum
FROM Nakls, Firms
WHERE
  FirmID = NFirm AND NDate=:DATE AND NType IN (0, 6)
ORDER BY FName
```

Начальное значение параметра DATE определяется на этапе разработки программы с помощью редактора свойства Params (рис. 3.12, *слева*) и инспектора объектов (рис. 3.12, *справа*).

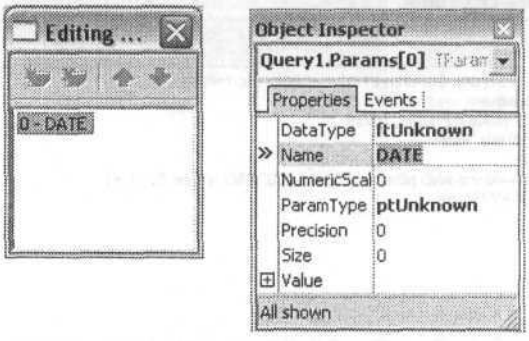


Рис. 3.12. Определение начального значения параметра

В тексте запроса имени параметра предшествует двоеточие. Имена параметров могут быть произвольными идентификаторами Delphi. Регистр букв в них не важен. Для задания начального значения параметра обычно в окне инспектора объектов достаточно определить его свойство Value — Delphi автоматически укажет его тип данных в свойстве DataType. Если тип оказывается неверным (например, вместо Integer выбран SmallInt), следует раскрыть список этого свойства и выбрать тип данных явно. Не следует менять имя параметра в свойстве Name, так как в этом случае Delphi не корректирует текст запроса и в нем будет фигурировать прежнее имя.

Для программного задания значения параметра используется показанное ниже свойство компонента TQuery:

```
property Params[Index: Word]: TParams;
```

Можно также использовать следующий метод компонента TQuery:

```
function ParamByName(const Value: String): TParam;
```

В первом случае конкретный параметр (в параметрическом запросе может быть сколько угодно параметров) задается индексом Index, причем первый по тексту запроса параметр имеет индекс 0, второй — 1 и т. д. Во втором случае параметр определяется своим именем в тексте запроса (но не именем в свойстве Name инспектора объектов!). В любом случае при обращении к параметру его значение нужно явно приводить к типу данных параметра свойством As*XXXX*:

```
quNakls.Params[0].AsDate := DateTimePicker1.Date;
quNakls.ParamByName('Date').AsDate := DateTimePicker1.Date;
```

Если к моменту программного изменения параметра запрос был открыт, его нужно закрыть, установить параметр(ы) и открыть вновь.

Значение Null свидетельствует о том, что полю не присвоено никакого значения («пустое» поле). Значение 0 для числовых полей или пустая строка для текстовых полей — это вполне определенные, *отличные от* Null значения. Чтобы присвоить параметру значение Null при прогоне программы, следует выполнить его метод Clear (на этапе разработки нужно очистить свойство Value параметра в окне инспектора объектов).

Для передачи параметров может служить свойство DataSource компонента TQuery. Если это свойство определено, явные присваивания значений параметрам не производятся, то есть не используются операторы присваивания типа

```
Query1.ParamByName('ИмяПараметра').Value := Значение;
```

В этом случае в наборе данных, связанном со свойством DataSource, отыскиваются поля, имена которых совпадают с именами параметров запроса. Если такие поля есть, их текущие значения берутся в качестве значений параметров, в противном случае возбуждается исключительная ситуация. Замечу, что только таким способом можно реализовать связь главный—детальный, если детальный НД создан с помощью компонента TQuery.

Методы Prepare и UnPrepare

Синтаксис SQL-операторов проверяется только при их выполнении методом Open или ExecSQL. Проверка синтаксиса требует некоторого времени, которое можно сэкономить, заранее подготовив запрос к многократному использованию методом Prepare компонента TQuery. После этого свойство Prepared компонента TQuery получит значение True, и при очередном выполнении запроса его синтаксис уже не будет проверяться.

При выполнении запрос компилируется и запоминается в буфере машины BDE, которая предварительно его оптимизирует, чтобы выполнить запрос в кратчайшее

время. Для освобождения выделенных запросу ресурсов используется метод UnPrepare. При выполнении неподготовленного запроса всегда сначала вызывается Prepare, а после выполнения — UnPrepare.

Изменяемые запросы

Записи НД, возвращаемые компонентом TQuery, могут изменяться подобно тому, как это происходит в компоненте TTable. Возможность изменения НД, возвращаемого после выполнения оператора **SELECT**, определяется свойством CanModify, значение которого устанавливается автоматически в зависимости от выполнения следующих условий:

- свойство RequestLive компонента должно содержать значение True;
- НД формируется обращением только к одной физической таблице БД;
- НД не сортируется (то есть в запросе не должно быть секции **ORDER BY**);
- в НД не создаются значения с помощью агрегатных функций;
- НД не кэшируется (свойство CashedUpdates должно содержать значение False);
- при обращении к серверу Sybase SQL Server таблица БД должна иметь уникальный индекс.

Свойство RequestLive компонента TQuery игнорируется, если нарушено хотя бы одно из перечисленных условий. Такие НД могут лишь отображать данные, а попытка вызвать любой из их методов — Insert, Edit или Delete — породит исключительную ситуацию. Для редактирования, вставки или удаления записей в этом случае нужно использовать либо специальные компоненты TUpdateSQL, либо соответствующие запросы Update, Insert, Delete, передаваемые SQL-серверу или BDE с помощью метода TQuery.ExecSQL. В том и другом случаях имеется одна существенная проблема, связанная с соответствующим обновлением данных на экране пользователя. Дело в том, что в обоих этих случаях обновление записей реализуется независимо от их отображения. Сетка DBGrid, например, будет содержать данные из НД, который лишь отображает их из нескольких таблиц и ничего «не подозревает» о возможном изменении одной или нескольких из них. Чтобы обновить демонстрируемые записи, в локальных БД часто используется метод Refresh компонентов TTable и TQuery, однако применение этого метода в серверных БД запрещено: если пользователь пожелает обновить демонстрируемые на экране данные, он должен будет закрыть НД и затем заново его открыть. При отображении сложного запроса, содержащего тысячи записей, это может привести к существенным задержкам, а если изменять данные требуется часто (например, при вводе нескольких записей подряд), обновление вообще становится невозможным.

Разумеется, в этом случае можно не обновлять отображаемые записи, а лишь тем или иным способом известить пользователя о том, что демонстрируемый ему экран нуждается в обновлении. Наиболее часто для этих целей используется дополнительная кнопка, первоначально скрытая от пользователя и появляющаяся только после внесения изменений. Для привлечения внимания пользователя

к новому интерфейсному объекту кнопку можно сделать «мигающей» и подать соответствующий звуковой сигнал. В некоторых случаях на экран выводится всплывающее окно, частично перекрывающее демонстрируемые данные. В следующем фрагменте после обращения к методу ExecSQL компонента quUpdate класса TQuery в центре экрана появляется всплывающее окно класса TAlarm, содержащее сообщение о необходимости актуализировать данные, и одновременно становится видимой кнопка bbRefresh, с помощью которой это можно сделать:

```
procedure TForm1.bbUpdateClick(Sender: TObject);
begin
  ...
  try
    quUpdate.ExecSQL;  // Изменяем данные
    Application.CreateForm(TAlarm, Alarm);
    with Alarm do
    begin              // Выводим всплывающее окно
      Show;
      SetWindowPos(Handle, hwnd_TopMost,
                   (Screen.Width-Width) div 2,
                   (Screen.Height-Height) div 2,
                   Width, Height, swp_NoActivate);
    end;
    bbRefresh.Show;    // Показываем кнопку обновления
    MessageBeep(mb_IconExclemation); // Издаем звуковой сигнал
  except
    ShowMessage('Ошибка изменения данных!')
  end
end;
```

Однако все это ничуть не уменьшает затраты времени, связанные с закрытием и повторным открытием сложного НД. Обновление «живых» НД реализуется значительно быстрее, но в этом случае работу по созданию сложного НД приходится выполнять на клиентском месте, что обычно медленнее и приводит к заметным задержкам при «листании» данных.

Пусть, например, осуществляется выборка сразу из трех таблиц: главной таблицы NAKLS, содержащей документацию об отгрузке книг покупателям (накладные), и двух таблиц подстановки, FIRMS и TYPENAKL, содержащих информацию о покупателях и типах накладных. Таблицы связаны по парам полей: NAKLS.NFirm — FIRMS.FirmID и NAKLS.NType — TYPES.TypeID. Показанный ниже SQL-запрос создаст необновляемый НД, который, однако, будет достаточно быстро листаться в сетке DBGrid, так как необходимое соединение полей реализуется сервером (или BDE):

```
SELECT
  ...    /* Перечень полей, выбираемых из таблицы NAKLS */,
  FName  /* Поле с именем партнера */,
  TName  /* Поле с наименованием типа накладной */
FROM
  Nakls, Firms, TypeNakl
```

```
WHERE
  FirmID = NFirm AND TypeID = NType
```

Вместо этого можно создать обновляемый НД:

```
SELECT
  ...   /* Перечень полей, выбираемых из таблицы NAKLS */
FROM
   Nakls
```

К обновляемому НД нужно добавить объекты-поля типа `fkLookup` со ссылкой на два вспомогательных НД с данными из таблиц `Firms` и `TypeNakl`. Для конечного пользователя результат будет одинаков, но второй НД можно обновлять методами **EDIT...POST**, в то время как для первого придется использовать SQL-запрос **UPDATE (INSERT, DELETE)** с последующими вызовами методов **CLOSE** и **OPEN**. Поскольку для второго НД соединение полей реализуется клиентом, он будет листаться с заметными паузами. В моей практике, например, одна из главных таблиц имела размер несколько сотен тысяч записей, в то время как таблицы подстановки состояли из нескольких тысяч записей. В первом случае (при объединении полей сервером) листание полного экрана (компонент `TDBGrid`) проходило в доли секунды и было практически незаметным, в то время как во втором — с паузами в 3–4 с. Однако обновление записи главной таблицы в первом случае реализовывалось более чем за минуту, в то время как во втором — за 1–2 с, то есть в 30–60 раз быстрее. Таким образом, при реализации сложных запросов следует искать компромисс между скоростью навигации и временем обновления: для каких-то групп пользователей может быть важным одно, а для других — другое.

Оператор **SELECT** обычно фильтрует записи в соответствии с условиями, описанными в секции **WHERE**. В «живом» запросе пользователь может ввести или изменить записи так, что они перестанут удовлетворять условиям секции **WHERE**. Как должен вести себя в этом случае НД, определяется значением свойства `Constrained`: если оно содержит значение `True`, введенная или модифицированная запись фильтруется и исчезает из НД (но сохраняется в реальной таблице БД), в противном случае присутствует в НД наряду с «правильными» записями. Так сказано в документации и встроенной справочной службе Delphi. На практике (в этом нетрудно убедиться, поставив несложный эксперимент) свойство `Constrained` никак не влияет на поведение НД.

Сортировка в обратном порядке

Сортировка в обратном порядке легко реализуется в компоненте `TTable`, если с таблицей связан нисходящий (`DESCENDING`) индекс. Однако при работе с компонентом `TQuery` явно использовать тот или иной индекс нельзя — выбор индекса реализуется сервером БД. В части **ORDER BY** предложения **SELECT** также нельзя указать порядок сортировки или какое-либо выражение вместо списка полей, поэтому в общем случае нисходящая сортировка в SQL-запросе невозможна.

Однако при сортировке по числовым полям и работе с файл-серверными БД этого можно добиться, введя вычисляемое поле, содержащее «обратные» значения,

и выполнив сортировку по этому полю. Если, например, в поле Summa указана сумма и требуется нисходящий порядок сортировки по этому полю, нужно ввести в запрос вычисляемое поле, в котором положительные значения Summa заменяются отрицательными, и выполнить сортировку по этому новому полю:

```
SELECT ..., -Summa AS SortSumma
FROM ...
ORDER BY SortSumma
```

К сожалению, этот прием не подходит для нисходящей сортировки по текстовым полям.

Следует заметить, что хотя SQL-интерпретатор BDE допускает сортировку по вычисляемым полям, стандарт SQL92 этого не требует, поэтому при работе с серверной БД нужно проверять возможность подобной сортировки. В InterBase, например, сортировка по вычисляемым столбцам невозможна, и для нисходящей сортировки требуется реальный «обратный» столбец.

Визуализация данных

Характерной особенностью визуализации данных (визуализирующие компоненты сосредоточены в категории Data Controls палитры компонентов Delphi) является их подключение к необходимым компонентам-наборам через связующие компоненты TDataSource (источник данных). В этом разделе описываются основные приемы работы как с компонентами-источниками TDataSource, так и с компонентами, предназначенными для визуализации данных.

Компонент TDataSource

Свойства

Для правильной работы компонента TDataSource необходимо указать в свойстве DataSet имя связанного с ним компонента-набора. После этого компонент TDataSource играет роль своеобразного клапана, открывающего доступ компонентам визуализации к данным или блокирующего этот доступ. Для этого используется свойство Enabled компонента: если программа поместит в него значение False, связь данных с отображающими их компонентами блокируется. Этот прием широко используется на практике, если нужно осуществить навигацию по записям НД, не отображая этот процесс в компонентах визуализации. Например, в процедуре Count, представленной в листинге 3.5, реализуется подсчет сумм всех накладных.

Листинг 3.5. Подсчет сумм всех накладных

```
procedure TfmNakls.Count;
// Посчитывает общее количество накладных, а также суммы накладных
// поставки, продажи, возврата и обмена книг
var
  N: Integer;
```

продолжение

Листинг 3.5 (продолжение)

```
  SI, SO, SR, SC: Real;
  BM: TBookmark;
  BegTime: TDateTime;  // Измерение времени счета
begin
  N := 0;  // Общее количество
  SI := 0; // Сумма поставок
  SO := 0; // Сумма продаж
  SR := 0; // Сумма возврата
  SC := 0; // Сумма обмена
  BegTime := Time;
  // Запоминаем текущую запись
  BM := Nakls.GetBookmark;
  // Отключаем доступ к данным сетки DBGrid1 и DBGrid2,
  // если установлен флажок CheckBox1
  if CheckBox1.Checked then
  begin
    DataSource1.Enabled := False;
    DataSource2.Enabled := False;
  end;
  // Переходим в начало НД
  Nakls.First;
  repeat
    inc(N);
    case NaklsNType.AsInteger of
    0,6: SI := SI+NaklsNSum.Value;
    1,7: SO := SO+NaklsNSum.Value;
    2,3: SR := SR+NaklsNSum.Value;
    4,5: SC := SC+NaklsNSum.Value;
    end;
  until not Nakls.FindNext;
  // Переходим к начальной записи и удаляем закладку
  Nakls.GotoBookmark(BM);
  Nakls.FreeBookmark(BM);
  // Включаем доступ к данным
  DataSource1.Enabled := True;
  DataSource2.Enabled := True;
  // Отображаем результат
  ShowMessage('Всего '+IntToStr(N)+' накл. Поставка: '+
    FloatToStrF(SI, ffNumber, 10, 2)+'р. Продажа: '+
    FloatToStrF(SO, ffNumber, 10, 2)+'р. Возврат: '+
    FloatToStrF(SR, ffNumber, 10, 2)+'р. Обмен: '+
    FloatToStrF(SC, ffNumber, 10, 2)+
              'р.'#13'Время счета: '+TimeToStr(Time-BegTime));
  DBGrid1.SetFocus;
end;
```

Несложные замеры времени показывают, что при отключении доступа время работы процедуры составляет менее 1 с, а при сохранении — 8 с (процессор —

1700 МГц, память — 256 Мбайт). В нашей демонстрационной БД содержится информация всего примерно 700 накладных. В реальной БД с десятками тысяч записей экономия времени может достигать нескольких минут, в особенности на менее быстрых компьютерах.

События

Для компонента TDataSource определены три события, возникающие при изменении состояния связанного с ним НД:

```
type TDataChangeEvent = procedure(Sender: TObject;
                                  Field: TField) of object;
property OnDataChange: TDataChangeEvent;
property OnStateChange: TNotifyEvent;
property OnUpdateData: TNotifyEvent;
```

Событие OnDataChange возбуждается сразу после изменения какого-либо поля и при переходе к следующему полю той же записи или при переходе к новой записи. В первом случае параметр Field содержит измененное поле, во втором — **NIL**. Это событие удобно использовать для синхронизации содержимого меток TLabel, TStaticText или других не связанных с данными компонентов, с изменением курсора или содержимого полей текущей записи НД.

Событие OnStateChange возникает при каждом изменении состояния связанного НД. Его удобно использовать для предоставления или отмены доступа к командам меню, кнопкам и другим интерфейсным компонентам.

Событие OnUpdateData возбуждается непосредственно перед запоминанием изменений в таблице (до выполнения метода Post). В его обработчике можно запретить внесение изменений для тех или иных групп пользователей.

Компонент TDBGrid

Компонент TDBGrid (сетка) отображает содержимое НД в виде таблицы, в которой столбцы соответствуют полям НД, а строки — записям.

Свойства

Свойства компонента TDBGrid представлены в табл. 3.29.

Таблица 3.29. Свойства компонента TDBGrid

Свойство	Описание
type TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWin); TBorderStyle = bsNone..bsSingle; **property** BorderStyle: TFormBorderStyle;	Определяет стиль рамки компонента (см. далее)

продолжение

Таблица 3.29 (продолжение)

Свойство	Описание
`property Columns: TDBGridColumns;`	Содержит индексированный набор объектов-столбцов типа `TDBGridColumns` (см. далее)
`property DataSource: TDataSource;`	Содержит ссылку на компонент типа `TDataSource`, служащий источником данных
`property DefaultDrawing: Boolean;`	Если содержит `True`, содержимое ячеек сетки прорисовывается автоматически, в противном случае — в обработчике события `OnDrawColumnCell` или `OnDrawDataCell`
`property EditorMode: Boolean;`	Если содержит `True`, пользователь может редактировать ячейку после нажатия клавиши F2 или Enter. Игнорируется, если свойство `Options` включает значение `goEditing` или `goAlwaysShowEditor`
`property FieldCount: Integer;`	Содержит количество столбцов
`property Fields[Index: Integer]: TField;`	Открывает индексированный доступ к полям связанного с компонентом НД
`property FixedColor: TColor;`	Определяет цвет фиксированных полей
`type TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgRowSelect, dgAlwaysShowSelection, dgConfirmDelete, dgCancelOnExit, dgMultiSelect);`	Определяет вид и поведение компонента (см. далее)
`TDBGridOptions = set of TDBGridOption;`	
`property Options: TDBGridOptions;`	
`property ReadOnly: Boolean;`	При значении `True` запрещает модификацию данных
`property SelectedField: TField;`	Открывает доступ к полю НД, связанному с текущим столбцом
`property SelectedIndex: Integer;`	Определяет индекс выбранного столбца в свойстве `Columns`
`property TitleFont: TFont;`	Определяет шрифт для заголовков столбцов

Свойство `BorderStyle` определяет стиль рамки компонента. Допустимые значения:

- `bsNone` — нет рамки;
- `bsSingle` — рамка толщиной 1 пиксел;

- bsSizeable — обычная рамка изменяемых размеров;
- bsDialog — диалоговая рамка (без возможности изменения размеров);
- bsToolWindow — рамка bsSingle с уменьшенным заголовком;
- bsSizeToolWin — рамка bsSizeable с уменьшенным заголовком.

Свойство Options определяет вид и поведение компонента. Допустимые значения:

- dgEditing — разрешает изменение НД;
- dgAlwaysShowEditor — автоматически переводит столбец в режим редактирования при его выделении;
- dgTitles — показывает заголовки столбцов;
- dgIndicator — показывает индикатор текущей строки в самом левом фиксированном столбце;
- dgColumnResize — разрешает пользователю вручную изменять ширину столбцов;
- dgColLines — показывает разделяющие вертикальные линии;
- dgRowLines — показывает разделяющие горизонтальные линии;
- dgTabs — разрешает переход от столбца к столбцу с помощью клавиши Tab;
- dgRowSelect — разрешает выделение цветом всей выбранной строки;
- dgAlwaysShowSelection — выделение текущей строки сохраняется, если компонент теряет фокус ввода;
- dgConfirmDelete — удаление строки должно подтверждаться;
- dgCancelOnExit — если пользователь вставил пустую строку и покинул ее, она не помещается в НД;
- dgMultiSelect — разрешает множественный выбор строк.

Объекты класса TDBGridColumns имеют заданное по умолчанию свойство Items, открывающее индексированный доступ к объектам-столбцам класса TColumn, и свойство Count, возвращающее количество элементов в свойстве Items. В табл. 3.30 и 3.31 перечисляются наиболее важные методы класса TDBGridColumns и свойства класса TColumn.

Таблица 3.30. Методы класса TDBGridColumns

Метод	Описание
`function Add: TColumn;`	Создает и возвращает объект-столбец
`procedure RebuildColumns;`	Удаляет старые определения объектов-столбцов и создает новые со свойствами, заданными по умолчанию
`procedure RestoreDefaults;`	Восстанавливает определения всех объектов-столбцов, заданные по умолчанию

Таблица 3.31. Свойства класса TColumn

Свойство	Описание
`type TColumnButtonStyle = (cbsAuto, cbsEllipsis, cbsNone);` `property ButtonStyle: TColumnButtonStyle;`	Определяет стиль назначенной столбцу кнопки: `cbsAuto` — столбец содержит кнопку раскрывающегося списка; `cbsEllipsis` — столбец содержит кнопку, щелчок по которой вызывает событие `OnEditButtonClick`; `cbsNone` — столбец не содержит кнопки
`property Color: TColor;`	Определяет цвет столбца
`property DisplayName: String;`	Содержит заголовок столбца
`property Field: TField;`	Содержит ссылку на связанное со столбцом поле НД
`property FieldName: String;`	Содержит имя связанного со столбцом поля НД
`property Font: TFont;`	Определяет шрифт столбца
`property PickList: TStrings;`	Определяет элементы раскрывающегося списка, если в `ButtonStyle` установлено значение `cbsAuto`
`property ReadOnly: Boolean;`	Если содержит `True`, данные в столбце нельзя изменять
`property Title: TColumnTitle;`	Содержит ссылку на объект-заголовок, имеющий стандартные для визуальных компонентов свойства `Alignment`, `Caption`, `Color` и `Font`, позволяющие управлять текстом в заголовке столбца
`property Visible: Boolean;`	Если содержит `False`, столбец не отображается в сетке
`property Width: Integer;`	Содержит ширину столбца в пикселах

Методы

Методы компонента `TDBGrid` перечислены в табл. 3.32.

Таблица 3.32. Методы компонента TDBGrid

Метод	Описание
`procedure DefaultDrawColumnCell (const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState);`	Осуществляет прорисовку ячейки таблицы, заданную по умолчанию: `Rect` — координаты ячейки; `DataCol` — индекс столбца; `Column` — столбец ячейки; `State` — состояние прорисовки
`function ValidFieldIndex (FieldIndex: Integer): Boolean;`	Возвращает `True`, если со столбцом с индексом `FieldIndex` связано поле НД. Если поле вычисляемое или оно не связано со столбцом, возвращает `False`

События

События компонента TDBGrid перечислены в табл. 3.33.

Таблица 3.33. События компонента TDBGrid

Событие	Описание
`type TDBGridClickEvent = procedure (Column: TColumn) of object;`	Возникает после щелчка мышью в ячейке
`property OnCellClick: TDBGridClickEvent;`	
`property OnColEnter: TNotifyEvent;`	Возникает в момент получения ячейкой фокуса ввода
`property OnColExit: TNotifyEvent;`	Возникает перед тем, как ячейка потеряет фокус ввода
`type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: LongInt) of object;`	Возникает при перемещении столбца: FromIndex — позиция столбца до перемещения; ToIndex — новая позиция столбца
`property OnColumnMoved: TMovedEvent;`	
`type DrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;`	Возникает при необходимости прорисовки ячейки: Rect — прямоугольник прорисовки; DataCol — индекс столбца в свойстве Columns; Column — столбец; State — состояние прорисовки
`property OnDrawColumnCell: TDrawColumnCellEvent;`	
`property OnEditButtonClick: TNotifyEvent;`	Возникает при щелчке на кнопке в столбце
`type TDBGridClickEvent = procedure (Column: TColumn) of object;`	Возникает при щелчке на заголовке
`property OnTitleClick: TDBGridClickEvent;`	

Создание объектов-столбцов

Объекты-столбцы предназначены для гибкого управления свойствами визуального представления данных, отображаемых в столбцах сетки. С их помощью можно менять порядок следования отображаемых полей НД, заголовки столбцов, используемые в столбце шрифт и фоновый цвет.

Обычно объекты-столбцы создаются на этапе разработки программы с помощью *редактора столбцов* (рис. 3.13). Для его вызова нужно выбрать команду Columns Editor в контекстном меню компонента.

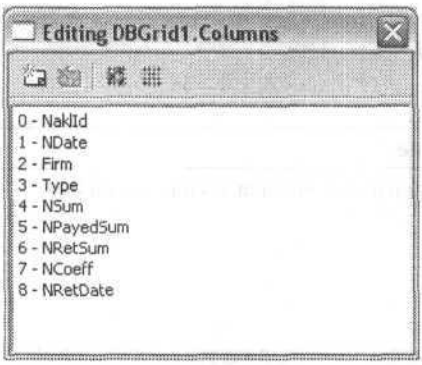


Рис. 3.13. Окно редактора столбцов

С помощью кнопки создается новый объект-столбец; кнопка позволяет удалить выделенный в окне редактора объект; с помощью кнопки создаются объекты-столбцы для всех доступных полей НД; наконец, кнопка служит для восстановления свойств выделенного объекта-столбца, заданных по умолчанию.

При работе с редактором сначала щелкните на кнопке , чтобы создать столбцы для всех полей НД, затем удалите ненужные столбцы. «Схватив» столбец мышью, перетащите его на нужное место (увеличенный на единицу номер слева от столбца покажет, каким по счету слева направо будет столбец в сетке). Для изменения заголовка столбца раскройте сложное свойство `Title` в окне инспектора объектов и измените его вложенное свойство `Caption`.

ПРИМЕЧАНИЕ

Если в сетке `TDBGrid` нужно оставить пустым заголовок какого-либо столбца, недостаточно просто удалить содержимое вложенного свойства `Caption` свойства `Title` соответствующего объекта-столбца — нужно поместить в заголовок хотя бы один пробел.

С помощью трех других вложенных свойств можно изменять выравнивание текста заголовка относительно границ столбца (`Alignment`), фоновый цвет заголовка (`Color`), цвет и шрифт текста (`Font`). Фоновый цвет столбца (кроме заголовка) и цвет/шрифт для отображаемых в нем данных изменяются с помощью свойств `Color` и `Font`. Защитить отображаемые в столбце данные от изменения можно с помощью свойства `ReadOnly`, а временно удалить его из сетки — с помощью свойства `Visible`.

Создание объектов-столбцов на этапе прогона программы обычно не требуется. Исключением является разработка «умной» программы, которая запоминает сделанные пользователем в столбцах изменения (порядок их следования, ширину) при завершении работы и восстанавливает в момент очередного старта. Наиболее подходящим местом для запоминания параметров объектов-столбцов является реестр Windows.

Два обработчика, представленные в листинге 3.6, демонстрируют технику запоминания объектов-столбцов в реестре (обработчик `FormClose`) и создания столбцов (`FormCreate`).

Листинг 3.6. Создание столбцов и запоминание их параметров в реестре Windows

```
uses Registry; { При работе с компонентом TRegistry нужна ссылка на
модуль Registry }
procedure TfmNakls.FormClose(Sender: TObject;
                             var Action: TCloseAction);
var
  Registry: TRegistry;
  k: Integer;
```

```
begin
  Registry := TRegistry.Create;
  with Registry do
  begin
    for k := 0 to DBGrid1.Columns.Count-1
              do with DBGrid1.Columns do
    begin
      OpenKey('\Software', True);
      OpenKey('Nakls', True);
      OpenKey('DBGrid1', True);
      OpenKey('Col'+IntToStr(k), True);
      WriteString('Field', Items[k].FieldName);
      WriteString('Caption', Items[k].Title.Caption);
      WriteInteger('Width', Items[k].Width);
      CloseKey
    end;  // for
    Destroy;
    Action := caFree
  end  // with
end;

procedure TfmNakls.FormCreate(Sender: TObject);
var
  Registry: TRegistry;
  k, N: Integer;
begin
  Registry := TRegistry.Create;
{ Проверяем существование ключа software/nakls/DBGrid1, иначе при
первом запуске программы все столбцы в сетке будут уничтожены }
  if (Registry.OpenKey('software',False)) and
     (Registry.OpenKey('nakls',False)) and
     (Registry.OpenKey('DBGrid1',False)) then
  with Registry do
  begin
    N := DBGrid1.Columns.Count-1;
    // Удаляем столбцы в сетке:
    DBGrid1.Columns.Clear;
    for k := 0 to N do with DBGrid1.Columns do
    begin
      OpenKey('\Software', True);
      OpenKey('Nakls', True);
      OpenKey('DBGrid1', False);
      OpenKey('Col'+IntToStr(k), False);
      Add;  // Добавляем очередной столбец
      Items[k].FieldName := ReadString('Field');
      Items[k].Title.Caption := ReadString('Caption');
      Items[k].Width := ReadInteger('Width');
      CloseKey
    end
  end;
  Registry.Destroy
end;
```

Пустые столбцы

Если создать объект-столбец и оставить незаполненным его свойство `FiledName`, будет создан *пустой столбец*. С помощью обработчика `OnDrawColumnCell` сетки (см. далее) в таком столбце можно выводить какую-либо информацию, основываясь на информации из действительно существующих полей набора данных. Например, из трех полей, содержащих, соответственно, название книги, имя автора и название издательства, можно в пустом столбце показать все три строки вместе. Однако для формирования нужной информации из полей НД значительно удобнее использовать событие `OnGetText` объекта-поля или создать в НД вычисляемое поле с соответствующим алгоритмом формирования выводимого значения. Чаще всего пустые столбцы действительно остаются пустыми, но при этом в строки столбца вставляются кнопки (для этого в свойство `ButtonStyle` устанавливается значение `cbsEllips`). При щелчке на кнопке (она становится видимой только в режиме редактирования ячейки с кнопкой) возникает событие `OnEditButtonClick`, которое программист может использовать по своему усмотрению. Например, графические поля, если их выводить для каждой записи в отдельном столбце сетки или даже для текущей записи с помощью отдельного компонента `TDBImage`, требуют существенных затрат времени на вывод. Для ускорения листания сетки вместо графического поля устанавливается пустой столбец с кнопкой, а в обработчике события `OnEditButtonClick` предусматривается вызов формы, показывающей содержимое графического поля текущей записи или активизацию компонента `TDBImage`, расположенного на той же форме, что и сетка `TDBGrid`. Аналогичным образом можно поступать с мемо-полями.

На рис. 3.14 показано окно отбора книг из списка (проект Ch03\EmptyCol\ EmptyCol.dpr). Самый правый столбец сетки — пустой. С помощью его кнопки в обработчике события `OnEditButtonClick` открывается диалоговое окно, в котором пользователь может указать количество экземпляров книги. Если диалог с пользователем заканчивается успешно, столбец будет отображать выбранные пользователем книги. Это достигается с помощью обработчика события `OnDrawColumnCell`.

Замечу, что показанное на рисунке окно появляется после выбора команды Вставить в контекстном меню нижнего окна со списком связанных с накладной книг.

Формирование списка возможных значений столбца

Свойство `PickList` объекта-столбца содержит список возможных значений столбца. Если это свойство заполнено, при редактировании ячейки столбца в ее правом углу появляется небольшая кнопка, щелчок на которой раскрывает список. Это свойство для столбцов, отображающих подстановочные (lookup) поля НД, автоматически формируется содержимым отображаемого поля. На рис. 3.15 показан список, который содержит допустимые значения столбца Тип накладной. Для остальных столбцов это свойство в большинстве случаев остается пустым, но программист может заполнить его как на этапе разработки программы, так и при ее прогоне.

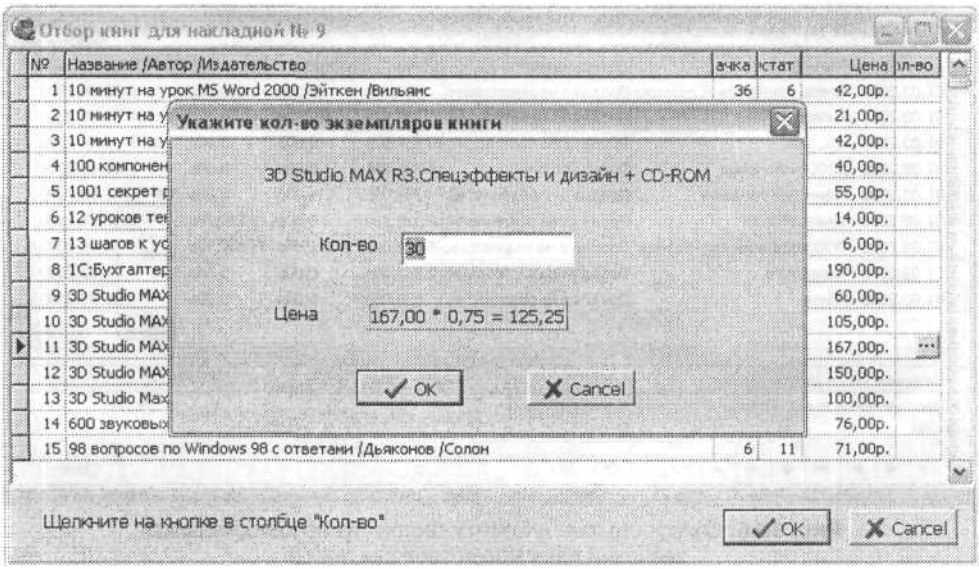


Рис. 3.14. Отбор книг для накладной

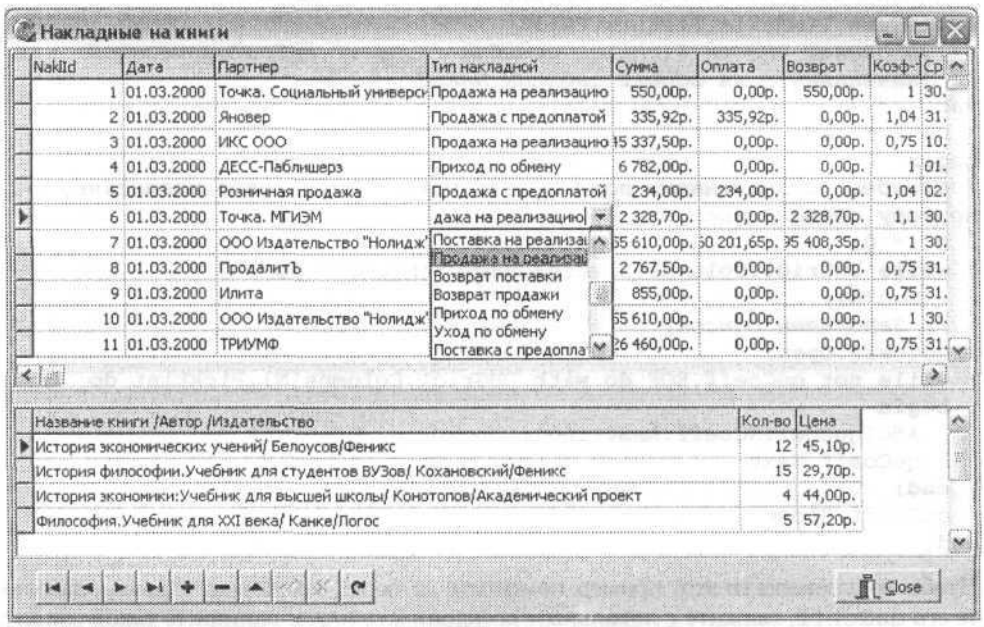


Рис. 3.15. Пример раскрывающегося списка для подстановочного поля

В процедуре `PickListFill`, представленной в листинге 3.7, список возможных значений столбца Коэф-т заполняется неповторяющимися значениями поля `NCoeff` таблицы `NAKLS` (проект Ch03\PickList\PickList.dpr, рис. 3.16).

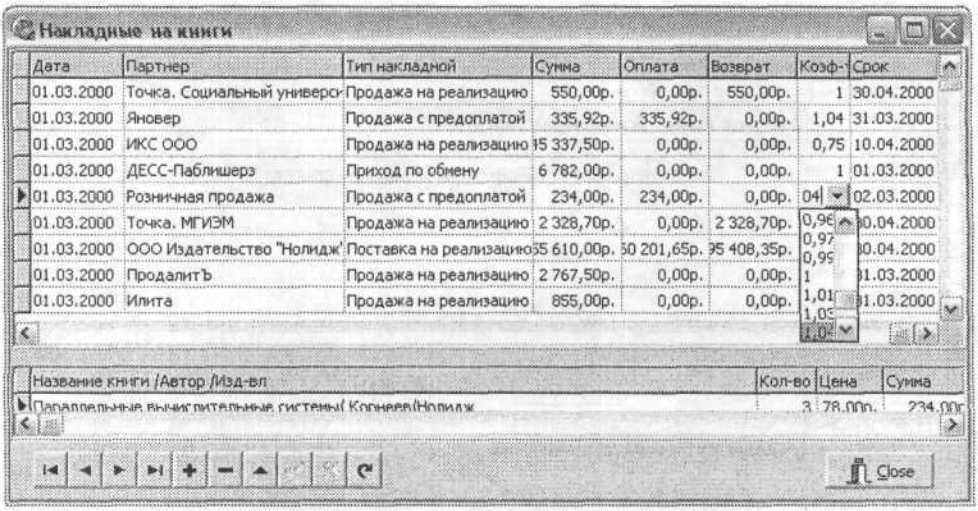


Рис. 3.16. Список столбца «Коэф-т» состоит из неповторяющихся значений поля NCoeff таблицы NAKLS

Листинг 3.7. Заполнение столбца сетки значениями поля набора данных

```
procedure TfmNakls.PickListFill;
// Заполняет список PickList столбца "Коэф-т" неповторяющимися
// значениями поля NCoeff таблицы NAKLS
var
  N: Integer;
begin
{ Столбец со значениями поля NCoeff может менять свое положение,
поэтому сначала ищем его: }
  N := 0;
  while DBGrid1.Columns[N].FieldName<>'NCoeff' do
    inc(N);
  // Заполняем список:
  quCoeff.Open;
  while not quCoeff.EOF do with DBGrid1.Columns[N].PickList do
  begin
    Add(quCoeffNCoeff.AsString);
    quCoeff.Next
  end;
  quCoeff.Close
end;
```

Чтобы воспроизвести этот пример, поместите на форму компонент TQuery, назовите его quCoeff, свяжите с локальным псевдонимом AAA и напишите такой запрос:

```
SELECT DISTINCT NCoeff
FROM Nakls
ORDER BY NCoeff
```

Создайте для НД quCoeff объект-поле quCoeffNCoeff. Процедуру PickListFill можно вызвать в конце обработчика FormCreate.

Управление отображением данных

С помощью обработчика события OnDrawColumnCell программист может учесть характер отображаемых данных. То, как осуществляется прорисовка данных в сетке TDBGrid — стандартным способом или по особому сценарию, — определяется свойством DefaultDrawing. Если это свойство имеет значение False, алгоритм прорисовки должен содержаться в обработчиках события OnDrawColumnCell или OnDrawDataCell. Замечу, что обработчик события OnDrawDataCell введен для совместимости с ранними версиями Delphi:

```
property OnDrawColumnCell: TDrawColumnCellEvent;
TDrawColumnCellEvent = procedure (Sender: TObject;
                                  const Rect: TRect;
                                  DataCol: Integer;
                                  Column: TColumn;
                                  State: TGridDrawState) of object;
```

Параметры здесь следующие:

- Rect — координаты области прорисовки;
- DataCol — порядковый номер текущего столбца, начиная с нулевого;
- Column — текущий столбец;
- State — состояние ячейки:
  - gdSelected — ячейка выделена;
  - gdFocused — ячейка имеет фокус ввода;
  - gdFixed — ячейка относится к фиксированной строке (строке заголовков столбцов) или столбцу.

Если параметр State содержит значение gdSelected (то есть ячейка выделена), в обработчике события OnDrawColumnCell необходима прорисовка инверсного прямоугольника, а если значение gdFocused — пунктирного прямоугольника выделения.

Для вывода ячеек стандартным образом используется метод DefaultDrawColumnCell.

Следует заметить, что если для сетки свойство DefaultDrawing содержит значение True, заданное по умолчанию, обработчики будут вызваны *после завершения* стандартной процедуры прорисовки, то есть программный вывод в этом случае накладывается на стандартный. Это может стать причиной неправильного отображения данных, например так, как показано на рис. 3.17 (проект Ch03\DrawCell\DrawCell.dpr).

В этом примере использовался обработчик события OnDrawColumnCell, представленный в листинге 3.8.

Листинг 3.8. Обработчик события OnDrawColumnCell

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
```

продолжение

Листинг 3.8 (продолжение)

```
begin
  if Books['BPrice']>10 then with DBGrid1.Canvas do
  begin
    Font.Style := [fsItalic];
    TextOut(Rect.Left, Rect.Top, Column.Field.Text)
  end
end;
```

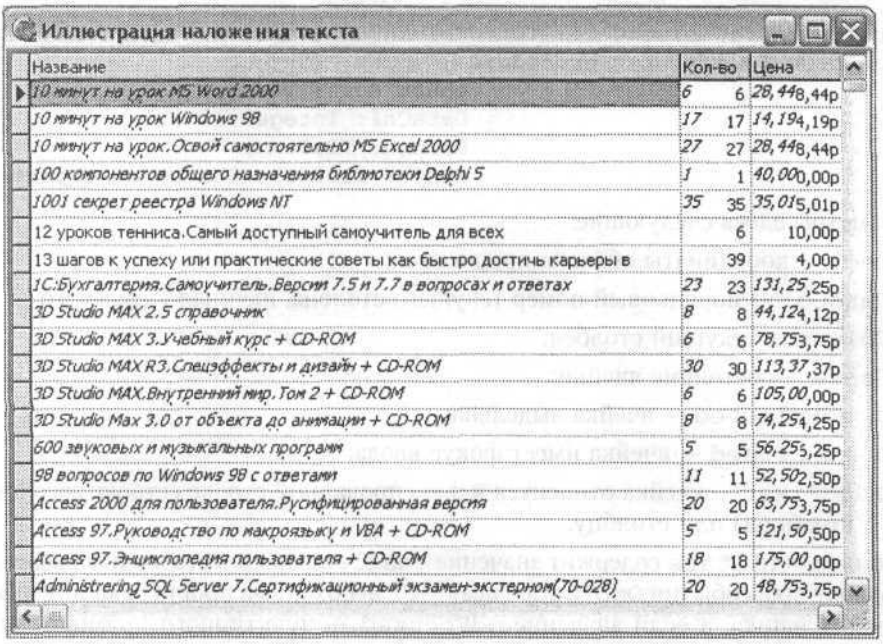


Рис. 3.17. Пример двойной прорисовки строк, в которых «Цена» больше 10 р.

Для большей наглядности нестандартный вывод сделан намеренно отличающимся от стандартного (курсив, текст не выравнивается).

Чтобы исключить нежелательное влияние стандартного вывода, нужно вначале закрасить прямоугольник прорисовки:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  if Books['Price']=0 then with DBGrid1.Canvas do
  begin
    FillRect(Rect);           // Удаляем стандартный вывод
    Font.Style := [fsItalic];
    TextOut(Rect.Left, Rect.Top, Column.Field.Text)
  end
end;
```

При двойной прорисовке может замедлиться скорость листания данных в сетке, особенно для компьютеров с невысокой производительностью. В этом случае нужно в свойство DefaultDrawing поместить значение False и позаботиться о стандартной прорисовке:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  if Books['Price']=0 then with DBGrid1.Canvas do
  begin
    Font.Style := [fsItalic];
    TextOut(Rect.Left, Rect.Top, Column.Field.Text)
  end
    else // Без следующего оператора
         // стандартные строки будут пустыми!
    DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State)
end;
```

Событие OnDrawColumnCell возникает при прорисовке каждой ячейки, при этом текущей записью базового НД становится запись с прорисовываемой ячейкой. Это дает возможность проверить информацию в ячейке, отличающейся от рисуемой. В обработчике, представленном в листинге 3.9, в списке накладных серым фоном закрашиваются строки, по которым нет задолженности (проект Ch03\DrawCell2\DrawCell.dpr, рис. 3.18).

Листинг 3.9. Выделение записей, по которым нет задолженности

```
uses db; // В модуле db определена константа ftString

{$R *.DFM}

procedure TfmNakls.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
// Закрашивает серым фоном строки с данными о накладных,
// по которым нет задолженности
begin
  if (NaklsNSum.Value=NaklsNPayedSum.Value+NaklsNRetSum.Value)
    and not (gdSelected in State) then
  with DBGrid1.Canvas do
  begin
    Brush.Color := clGray;
    FillRect(Rect);
    Font.Color := clWhite;
    if Column.Field.DataType=ftString then
    // В текстовых полях текст выравнивается влево
      TextOut(Rect.Left+2,Rect.Top+2,Column.Field.Text)
    else
```

продолжение

Листинг 3.9 (продолжение)

```
    // В остальных полях — вправо
      TextOut(Rect.Right-TextWidth(Column.Field.Text)-
              2, Rect.Top+2,Column.Field.Text)
  end;
end;
```

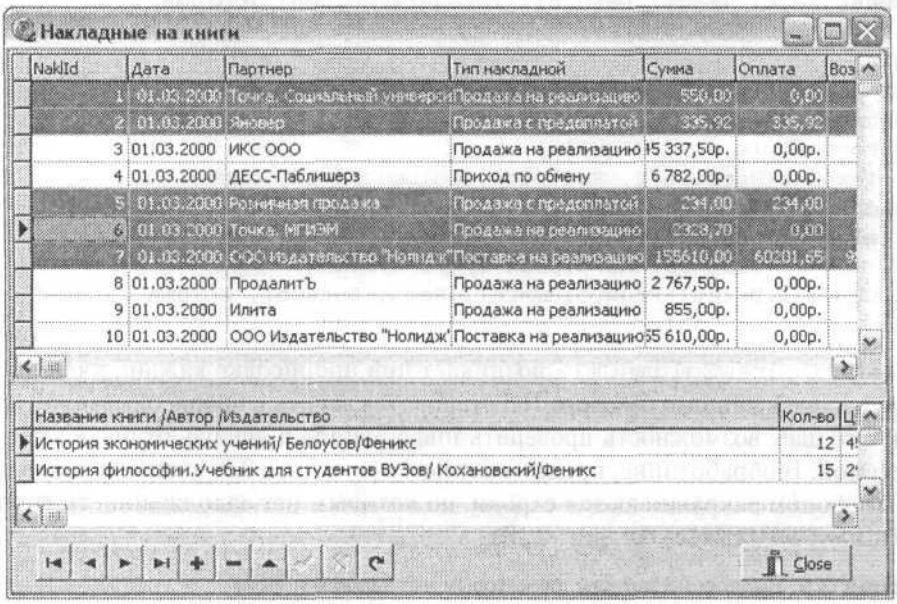


Рис. 3.18. Серым цветом выделяются оплаченные накладные

В этом обработчике проверяется активность ячейки, и, если ячейка активна (`gdSelected in State`), для вывода используется обработчик `DefaultDrawDataCell`, заданный по умолчанию. Сдвиг на 2 пиксела вправо и вниз относительно левого верхнего угла прямоугольника `Rect` ячейки реализован в обработчике `DefaultDrawDataCell` для стандартного шрифта MS Sans Serif высотой 8 пунктов, поэтому если вы хотите, чтобы выводимый вами текст не отличался от стандартного, указывайте эти величины смещения (для другого шрифта они нуждаются в корректировке). В обработчике проверяется тип данных: строковые данные выравниваются по левому краю столбца, остальные — по правому (именно так поступает обработчик `DefaultDrawDataCell`).

Дополнительные возможности сетки

У компонента есть свойство, с помощью которого можно управлять поведением и внешним видом сетки:

```
type
TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles,
                 dgIndicator, dgColumnResize, dgColLines,
                 dgRowLines, dgTabs, dgRowSelect,
```

```
                dgAlwaysShowSelection, dgConfirmDelete,
                dgCancelOnExit, dgMultiSelect);
TDBGridOptions = set of TDBGridOption;
property Options: TDBGridOptions;
```

Параметры здесь следующие:

- `dgEditing` — разрешает изменение НД;
- `dgAlwaysShowEditor` — при выделении столбца автоматически переводит его в режим редактирования;
- `dgTitles` — показывает заголовки столбцов;
- `dgIndicator` — показывает указатель текущей строки в самом левом фиксированном столбце;
- `dgColumnResize` — разрешает пользователю с помощью мыши менять ширину столбцов;
- `dgColLines` — показывает в сетке вертикальные разделяющие линии;
- `dgRowLines` — показывает горизонтальные разделяющие линии;
- `dgTabs` — разрешает переход от столбца к столбцу с помощью клавиши Tab;
- `dgRowSelect` — разрешает выделение цветом всей текущей строки;
- `dgAlwaysShowSelection` — выделение текущей строки цветом сохраняется, даже если компонент теряет фокус ввода;
- `dgConfirmDelete` — требует подтверждения удаления строки;
- `dgCancelOnExit` — если пользователь вставляет пустую строку и покидает ее, она не сохраняется в таблице БД;
- `dgMultiSelect` — разрешает множественный выбор строк.

Замечу, что отсутствие в свойстве значения `dgEditing` защищает отображаемые в сетке данные от изменения независимо от значения свойства `ReadOnly` любого ее столбца. Точно так же защищает строки вставка в свойство значения `gdRowSelected`, кроме того, в этом случае из свойства автоматически удаляются значения `dgEditing` и `dgAlwaysShowEditor` (если, разумеется они входили в множество `Options`; эти значения нельзя вставить в свойство, пока в нем присутствует значение `dgRowSelected`). В защищенной сетке нельзя изменить ни одной ячейки текущей строки, но можно удалить строку или вставить пустую строку (если в свойстве нет значения `dgCancelOnExit`). Если вы не собираетесь защищать данные, полезно вставить в свойство `Options` значение `dgAlwaysShowEditor` — в этом случае столбец автоматически переводится в режим редактирования, как только он получает фокус ввода; если это значение отсутствует, редактировать выделенный столбец можно после нажатия клавиши F2 или Enter либо после двойного щелчка на ячейке мышью.

Компоненты для визуализации полей текущей записи

В отличие от сетки `TDBGrid`, все рассматриваемые в этом разделе компоненты визуализируют отдельные поля текущей записи. В связи с этим говорят о двух способах визуализации данных: в виде *таблицы* (сетки) и в виде *формы*. Последняя

представляет собой обычную форму, на которую помещены компоненты визуализации полей текущей записи. На рис. 3.19 показан пример формы для ввода и редактирования данных по накладной.

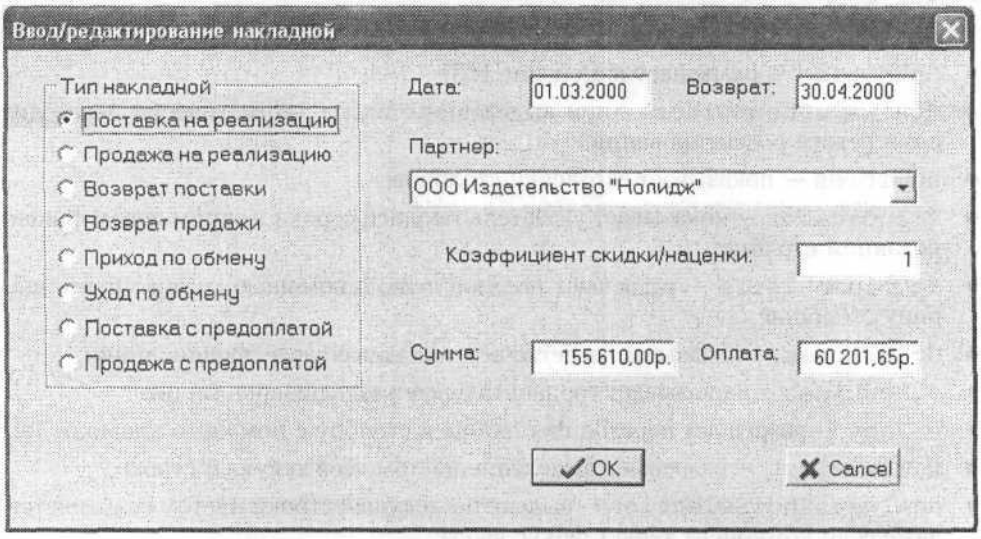


Рис. 3.19. Пример формы для ввода и редактирования данных по накладной

Для нормальной работы любого из описываемых далее компонентов нужно определить два его свойства: поместить в свойство `DataSource` имя связанного с набором данных компонента-источника `TDataSource`, а в свойство `DataField` — имя отображаемого поля.

Компонент TDBText

С помощью компонента `TDBText` текстовое содержимое различных полей НД можно показывать в том виде, в котором оно отображается в сетке `TDBGrid`. Фактически компонент повторяет функциональность метки `TLabel` за тем исключением, что его текст формируется автоматически на основании значения некоторого поля текущей записи. Отображаемый компонентом текст нельзя изменять, и, следовательно, с его помощью нельзя редактировать связанное с ним поле. Разумеется, тип отображаемого поля должен обеспечивать приведение к текстовому значению.

Для использования компонента `TDBText` нужно связать его с соответствующим полем НД с помощью свойств `DataSource` и `DataField`.

Компонент TDBEdit

Компонент `TDBEdit` позволяет редактировать значение поля текущей записи НД. Он повторяет функциональность компонента `TEdit`, позволяющего корректировать текстовое значение переменной, но источником данных и их приемни-

ком для него служит поле НД. Тип этого поля должен обеспечивать приведение к текстовому значению. В форме, показанной на рис. 3.19, компонент TDBEdit использован для ввода и редактирования полей дат, сумм и коэффициента скидки/наценки.

При вводе значения в компонент TDBEdit программа автоматически следит за тем, чтобы оно было совместимо по формату с полем НД. Ввод неверных значений блокируется. Например, если в компонент, связанный с полем типа дата-время, попытаться поместить произвольный текст, будет возбуждена исключительная ситуация.

Свойства, методы и события компонента аналогичны свойствам, методам и событиям стандартного компонента TEdit.

Компонент TDBCheckBox

Компонент TDBCheckBox обладает функциональностью стандартного флажка TCheckBox, но источником данных и их приемником для него служит поле НД, которое может быть логическим или символьным. В последнем случае необходимо соответствующим образом установить значения текстовых свойств ValueChecked и ValueUnchecked компонента. Например:

```
DBCheckBox1.ValueChecked := 'True;Yes;On;Да;Д';
DBCheckBox1.ValueUnchecked := 'False;No;Off;Нет;Н';
```

Компонент TDBRadioGroup

Компонент TDBRadioGroup служит для представления фиксированного набора возможных значений поля при помощи группы переключателей. Этот компонент обладает функциональностью стандартного переключателя TDBRadioGroup, но источником данных и их приемником для него служит поле НД. На рис. 3.19 этот компонент позволяет выбрать тип накладной.

Количество и названия вариантов возможных значений поля определяются в свойстве

```
property Items: TStrings;
```

Компонент будет связывать установку того или иного переключателя с соответствующим значением поля. Для этого список возможных значений определяется свойством

```
property Values: TStrings;
```

Например, для формы с рис. 3.19 свойства Items и Values компонента dbrgType типа TDBRadioGroup определены так, как показано на рис. 3.20.

Списочные компоненты

Визуализирующие компоненты типа TDBListBox, TDBComboBox, TDBLookupComboBox, TDBLookupListBox имеют связанные с ними списки TStrings и поэтому дальше объединяются общим названием *списочные* компоненты БД.

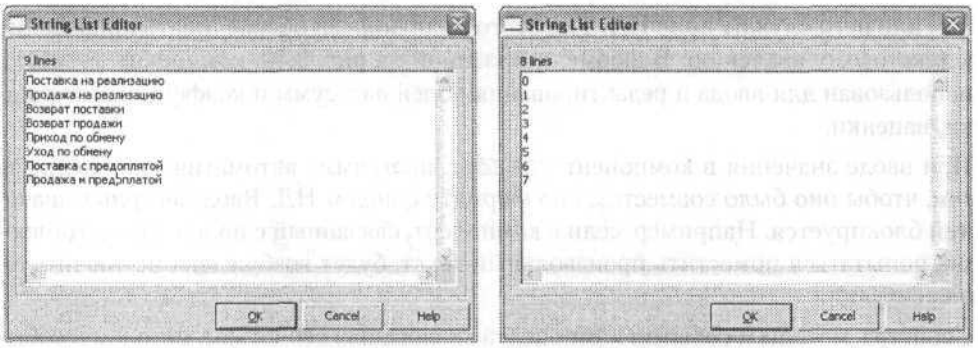


Рис. 3.20. Значения свойств компонента TDBRadioGroup: слева — Items; справа — Values

Списочные компоненты предназначены для отображения состояния конкретного поля текущей записи НД, а также для ввода в это поле нового значения (но никак не для навигации по НД!). Первые два компонента имеют первоначально пустые списки, которые программист должен заполнить перед тем, как они станут доступными пользователю. Компоненты `TDBLookup`*XXXX* формируют свои списки значениями, взятыми из поля другого НД, и не нуждаются в их наполнении — вот, собственно, и вся разница между ними и первыми двумя компонентами.

Компоненты `TDBListBox` и `TDBLookupListBox` могут отображать текущее значение связанного с ними поля только в том случае, если это значение совпадает с одним из списочных значений компонента, два других отображают любое значение поля, даже если этого значения нет в связанных с компонентами списках. Точно так же вводить новое значение компоненты *XXXX*`ListBox` могут только из своих списков, в то время как компоненты *XXXX*`ComboBox` в модификациях `csSimple` и `csComboBox` содержат текстовую строку для ввода значения, которого нет в списке.

Компоненты `TDBLookup`*XXXX* могут использоваться двумя способами: если в редактируемом НД есть нужное подстановочное поле, для них, как и для других рассматриваемых в этом разделе компонентов, достаточно определить лишь свойства `DataSource` и `DataField`, причем в последнем указать имя подстановочного поля; если в НД нет подстановочного поля, в свойство `DataSource` помещается имя компонента-источника, связанного с редактируемым НД, в свойство `DataField` — имя поля с шифром (ссылочного поля) этого НД, в свойство `ListSource` — имя компонента-источника, связанного с подстановочным НД, в `ListField` — имя отображаемого (результирующего) поля и, наконец, в `KeyField` — имя индексного поля, на которое ссылается поле `DataField` редактируемого НД. Например, в форме на рис. 3.19 компонент `TLookupComboBox` используется для отображения имени партнера. В первом варианте в его свойство `DataSource` нужно поместить значение `DataSource1`, а в свойство `DataField` — имя подстановочного поля `Firm`. Во втором варианте предварительно на форме нужно разместить компонент `DataSource3`, связав его с НД `Firms`, в свойство `DataSource` по-прежнему помещается значение `DataSource1`, а вот в свойство `DataField` — имя подстановочного поля `NFirm`, при этом в `ListSource` — значение `DataSource3`,

в ListField — FName и в KeyField — FirmID. Способ использования компонента не имеет значения: в любом случае в поле NFirm НД Nakls будет помещен шифр выбранного партнера.

Компонент TDBMemo

Компонент TDBMemo предназначен для отображения и редактирования мемо-полей (полей комментариев), которые служат для хранения в таблицах БД многострочных текстов. Компонент TDBMemo является практически аналогом компонента TMemo — с той разницей, что источником данных для него служит мемо-поле НД. В связи с этим его свойства, методы и события в основном совпадают со свойствами, методами и событиями компонента TMemo.

Специфичными для компонента являются свойства DataSource и DataField (в которые следует поместить имена компонента-источника и мемо-поля соответственно), а также следующие свойства:

```
property AutoDisplay: Boolean;
property Field: TField;
```

Если в свойство AutoDisplay помещено значение True, любые изменения мемо-поля автоматически отображаются в компоненте, а при открытии НД компонент будет отображать содержимое мемо-поля текущей записи. Если в свойство AutoDisplay помещено значение False, содержимое мемо-поля заменяется его названием, а для просмотра или редактирования его значения нужно либо дважды щелкнуть на компоненте, либо выделить его и нажать клавишу Enter.

Свойство Field открывает доступ к мемо-полю.

При корректировке текста в компоненте TDBMemo набор данных, которому принадлежит поле, автоматически переводится в состояние dsEdit. Замечу, что изменение значения свойств Text или Lines не переводит НД в режим редактирования, и эти изменения не переносятся в мемо-поле. Поэтому перед внесением изменений в значения свойств Lines или Text следует переводить НД в режим редактирования методом Edit, а затем запоминать изменения методом Post.

Компонент TDBRichEdit

Компонент TDBRichEdit позволяет просматривать и корректировать информацию в мемо-поле форматированного комментария. Текст форматированного комментария может содержать фрагменты, набранные различным шрифтом, размером, стилем, цветом и т. д. В отличие от компонента TDBMemo, который позволяет работать только с однородным (неформатированным) текстом, компонент TDBRichEdit умеет интерпретировать специальные символы разметки текста в формате RTF (Rich Text Format — расширенный текстовый формат). Многие свойства, методы и события компонента аналогичны по назначению одноименным свойствам, методам и событиям компонента TRichEdit и в этом разделе не описываются.

Специфичными для компонента являются свойства DataSource, DataField, AutoDisplay и Field, описанные в предыдущем подразделе.

Компонент TDBCtrlGrid

Как уже говорилось, существует два средства визуализации данных: таблицы и формы. В таблицах пользователь может видеть одновременно несколько записей НД, что облегчает ему контроль за данными и выбор необходимой записи. При применении формы пользователь в каждый момент видит на экране лишь одну запись НД, но зато имеет простой доступ к любому ее полю, в том числе многострочному или графическому. Компонент `TDBCtrlGrid` в известной мере сочетает в себе удобства обоих средств: он представляет собой таблицу, каждая ячейка которой отображается в виде формы (проект Ch03\CtrlGrid\CtrlGrid.dpr, рис. 3.21).

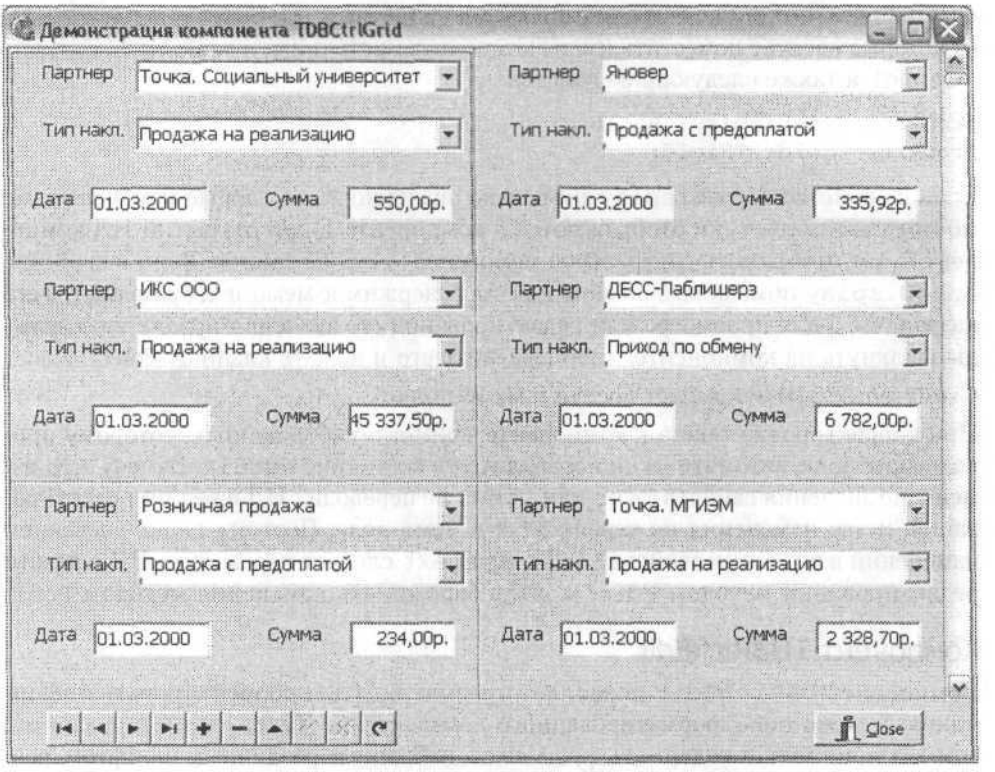


Рис. 3.21. Пример использования компонента TDBCtrlGrid

Для работы с компонентом необходимо поместить его на форму, связать с компонентом `TDataSource`, который, в свою очередь, связать с каким-либо набором данных (`TTable` или `TQuery`), и затем разместить на нем необходимые компоненты для работы с полями базы данных — `TDBText`, `TDBEdit`, `TDBCheckBox` и т. п. Компоненты для работы с полями базы данных требуется разместить в верхней строке компонента `TDataSource` или, если количество столбцов `ColCount` компонента больше 1, в левой верхней ячейке компонента (рис. 3.22).

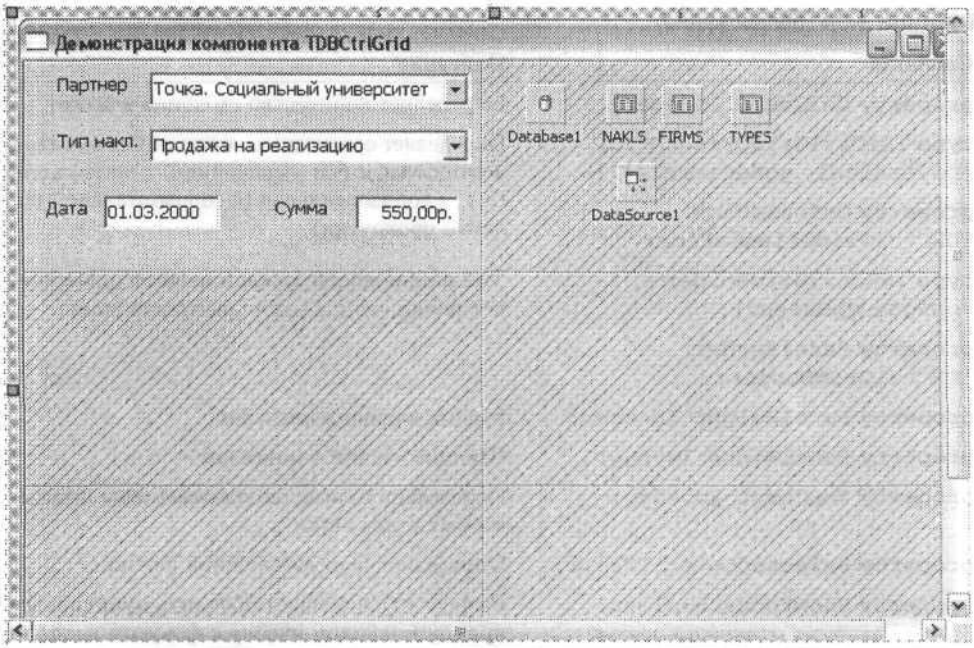


Рис. 3.22. Пример работы с компонентом TDBCtrlGrid на этапе разработки программы

Во время прогона программы расположение компонентов в верхней строке (верхней левой ячейке) компонента `TDBCtrlGrid` и их состав будут реплицированы на все оставшиеся строки (ячейки), как это видно из рис. 3.21.

Приемы работы с компонентами `TDBCtrlGrid` и `TDBGrid` аналогичны. Текущая строка (ячейка) выделяется пунктирным прямоугольником. Для навигации по сетке компонента используются те же клавиши, что и для `TDBGrid`. Для вставки новой записи необходимо нажать клавишу Insert или попытаться перейти с последней записи в НД вниз на одну строку (для сетки с одним столбцом). Для изменения записи достаточно ввести новое значение в какое-либо поле. Для удаления записи необходимо нажать комбинацию клавиш Ctrl+Delete. При этом на возможности редактирования, добавления и удаления записей в компоненте `TDBCtrlGrid` влияет ряд свойств как других компонентов, размещенных в ячейках компонента `TDBCtrlGrid`, так и самого компонента `TDBCtrlGrid`.

Далее приводится обзор некоторых свойств, методов и событий компонента `TDBCtrlGrid`.

Свойства компонента `TDBCtrlGrid` представлены в табл. 3.34.

Таблица 3.34. Свойства компонента TDBCtrlGrid

Свойство	Описание
`property AllowDelete: Boolean;`	Разрешает/запрещает удаление записей
`property AllowInsert: Boolean;`	Разрешает/запрещает вставку записей

продолжение

Таблица 3.34 (продолжение)

Свойство	Описание
`property ColCount: Integer;`	Определяет количество столбцов в сетке
`type TDBCtrlGridOrientation = (goVertical, goHorizontal);` `property Orientation: TDBCtrlGridOrientation;`	Определяет ориентацию сетки: `goVertical` — вертикальная (по умолчанию); `goHorizontal` — горизонтальная (с горизонтальной полосой прокрутки)
`type TDBCtrlGridBorder = (gbNone, gbRaised);` `property PanelBorder: TDBCtrlGridBorder;`	Тип рамки вокруг каждой ячейки: `gbNone` — нет рамки; `gbRaised` — выпуклая рамка
`property PanelHeight: Integer;`	Высота ячейки в пикселах
`property PanelWidth: Integer;`	Ширина ячейки в пикселах
`property RowCount: Integer;`	Определяет количество одновременно показываемых строк сетки
`property SelectedColor: TColor;`	Определяет цвет выделенной ячейки
`property ShowFocus: Boolean;`	Разрешает/запрещает обводить пунктирным прямоугольником ячейку с фокусом ввода

Для компонента `TDBCtrlGrid` определен метод `DoKey`:

```
type TDBCtrlGridKey = (gkNull, gkEditMode, gkPriorTab, gkNextTab,
                       gkLeft, gkRight, gkUp, gkDown, gkScrollUp,
                       gkScrollDown, gkPageUp, gkPageDown,
                       gkHome, gkEnd, gkInsert, gkAppend,
                       gkDelete, gkCancel);
procedure DoKey(Key: TDBCtrlGridKey);
```

Действие, выполняемое методом, определяется параметром `Key`:

- `gkNull` — ничего не делать;
- `gkEditMode` — перевести компонент в режим редактирования;
- `gkPriorTab` — передать фокус ввода предыдущей ячейке;
- `gkNextTab` — передать фокус ввода следующей ячейке;
- `gkLeft` — передать фокус ввода ячейке слева;
- `gkRight` — передать фокус ввода ячейке справа;
- `gkUp` — передать фокус ввода ячейке сверху;
- `gkDown` — передать фокус ввода ячейке снизу;
- `gkScrollUp` — прокрутить сетку на строку вверх;
- `gkScrollDown` — прокрутить сетку на строку вниз;
- `gkPageUp` — прокрутить сетку на экран вверх;
- `gkPageDown` — прокрутить сетку на экран вниз;

- gkHome — передать фокус ввода первой ячейке;
- gkEnd — передать фокус ввода последней ячейке;
- gkInsert — перевести компонент в режим редактирования;
- gkAppend — перевести компонент в режим добавления записи;
- gkDelete — перевести компонент в режим удаления записи;
- gkCancel — восстановить режим просмотра.

Для компонента TDBCtrlGrid определены следующие события, аналогичные одноименным событиям сетки TDBGrid: OnClick, OnDblClick, OnDragDrop, OnDragOver, OnEndDrag, OnEnter, OnExit, OnKeyDown, OnKeyPress, OnKeyUp, OnStartDrag. Дополнительно введено событие, которое наступает для каждой ячейки TDBCtrlGrid перед ее отображением:

```
property OnPaintPanel: TPaintPanelEvent;
TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid;
                             Index: Integer) of object;
```

Обработчик этого события может управлять прорисовкой ячейки. Параметр TDBCtrlGrid показывает, какой именно компонент TDBCtrlGrid нуждается в прорисовке; параметр Index определяет индекс отображаемой ячейки.

Компонент TDBNavigator

Строго говоря, компонент TDBNavigator (навигатор БД) не предназначен для отображения данных. Его назначение — дать пользователю программы удобное средство перемещения по записям НД и облегчить ему такие действия, как вставка новой записи, а также редактирование и удаление существующей записи. Однако навигатор БД используется только совместно с компонентами визуализации, поэтому мне кажется вполне уместным рассмотреть его особенности именно в этом разделе.

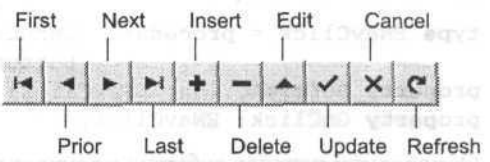


Рис. 3.23. Кнопки навигатора БД

На рис. 3.23 показаны кнопки навигатора.

Назначение кнопок:

- First — устанавливает курсор на первую запись;
- Prior — устанавливает курсор на предыдущую запись;
- Next — устанавливает курсор на следующую запись;
- Last — устанавливает курсор на последнюю запись;
- Insert — переводит НД в режим вставки новой записи;
- Delete — удаляет текущую запись;
- Edit — переводит НД в режим редактирования;
- Post — запоминает изменения, сделанные в текущей записи;
- Cancel — отменяет изменения, сделанные в текущей записи;
- Refresh — обновляет НД (для TQuery — только если запрос обновляемый).

С помощью свойства DataSource компонент связывается с нужным источником данных TDataSource — это все, что необходимо для его нормальной работы. Представленное ниже свойство управляет отображением диалогового окна с просьбой подтвердить удаление записи (значение True этого свойства выводит окно):

```
property ConfirmDelete: Boolean;
```

Если следующее свойство имеет значение True, кнопки плоские, в противном случае — объемные:

```
property Flat: Boolean;
```

С помощью другого свойства можно отображать только те кнопки навигатора, которые действительно необходимы:

```
type TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast,
                     nbInsert, nbDelete, nbEdit, nbPost,
                     nbCancel, nbRefresh);
type TButtonSet = set of TNavigateBtn;
property VisibleButtons: TButtonSet;
```

Например, если НД является результатом выполнения необновляемого запроса, в навигаторе имеет смысл оставить только первые 4 кнопки.

С помощью следующего метода можно имитировать щелчок на нужной кнопке навигатора:

```
procedure BtnClick(Index: TNavigateBtn);
```

Для компонента определены два специфических события:

```
type ENavClick = procedure (Sender: TObject;
                            Button: TNavigateBtn) of object;
property BeforeAction: ENavClick;
property OnClick: ENavClick;
```

Обработчик первого события получает управление перед выполнением действия, связанного со щелчком на кнопке Button, в то время как обработчик OnClick — после выполнения действия.

4 Технология dbGo.NET

Начиная с версии 5, Delphi поддерживает технологию ADO (ActiveX Data Objects — объекты данных, построенные как объекты ActiveX), которая развивалась корпорацией Microsoft и которая в конце концов переросла в технологию ADO.NET (см. главу 2).

В последних версиях Delphi «старая» технология ADO называется dbGo. В Delphi 2005 она приспособлена к работе на платформе .NET и называется dbGo.NET. Все компоненты технологии сосредоточены в категории dbGo палитры компонентов Delphi.

На основе этой технологии созданы соответствующие компоненты-наборы `TADOTable`, `TADOQuery`, `TADOStoredProc`, повторяющие в функциональном отношении компоненты `TTable`, `TQuery`, `TStoredProc`, но не требующие развертывания и настройки BDE на клиентской машине.

Основные особенности использования технологии dbGo.NET не зависят от архитектуры БД — эта технология характерна не только для файл-серверных БД, но также и для клиент-серверных и трехзвенных БД. Однако в этой главе (и в книге вообще) мы не будем рассматривать особенности компонентов `TADOStoredProc` и `TRDSConnection`, предназначенных для поддержки, соответственно, клиент-серверной и трехзвенной архитектур, — если вас интересуют эти вопросы, обратитесь к документации и/или к встроенной справочной службе.

Основным достоинством технологии dbGo.NET является ее естественная ориентация на создание «облегченного» клиента. В рамках этой технологии на машине разработчика БД устанавливаются базовые объекты MS ADO 2.1 и соответствующие компоненты Delphi (рис. 4.1), обеспечивающие использование технологии dbGo.NET (эти установки осуществляются автоматически при развертывании Delphi). На машине сервера данных (это может быть файловый сервер в рамках файл-серверной технологии или машина с сервером данных в технологии

клиент–сервер) устанавливается так называемый *провайдер данных* — некоторая надстройка над специальной технологией OLE DB, «понимающая» запросы объектов ADO и «умеющая» переводить эти запросы в нужные действия с данными. Взаимодействие компонентов ADO и провайдера осуществляется на основе универсальной для Windows технологии ActiveX, причем провайдер реализуется как COM-сервер, а ADO-компоненты — как COM-клиенты.

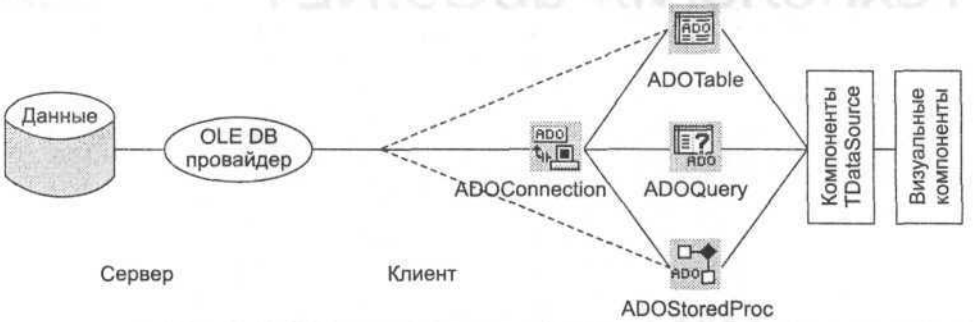


Рис. 4.1. Реализация технологии dbGo.NET в Delphi

На машине сервера создается и размещается источник данных. В случае файл-серверных систем отдельные таблицы типа dBASE, FoxPro, Paradox и т. п. должны управляться соответствующим ODBC-драйвером, а в роли провайдера используется Microsoft OLE DB Provider for ODBC drivers. Если по каким-либо причинам не найден нужный драйвер, файл-серверные таблицы можно перевести в формат MS Access. На их основе создается единый файл, содержащий все необходимые таблицы, индексы, хранимые процедуры и прочие элементы БД. Такой файл управляется машиной баз данных Microsoft Jet 4.0 Database Engine, а в роли провайдера используется Microsoft Jet 4.0 OLE DB Provider.

Если применяется промышленный сервер данных Oracle или MS SQL Server, данные не нуждаются в какой-либо предварительной подготовке, а в роли провайдера используется, соответственно, Microsoft OLE DB Provider for Oracle или Microsoft OLE DB Provider for SQL Server. Нетрудно обнаружить и явный недостаток такой технологии: dbGo.NET не может использоваться, если для соответствующей структуры данных (в частности, для БД многих популярных серверов — InterBase, Informix, DB2 и пр.) не создан нужный провайдер или ODBC-драйвер.

На машине клиента располагаются связные компоненты `TADOConnection` и компоненты-наборы данных `TADOTable`, `TADOQuery`, `TADOStoredProc`, а также не показанные на рисунке компоненты-наборы `TADODataSet` и командные компоненты `TADOCommand`. Каждый из этих компонентов может связываться с провайдером данных либо с помощью связного компонента `TADOConnection`, либо минуя его и используя собственное свойство `ConnectionString`. Таким образом, компонент `TADOConnection` играет роль концентратора соединений с источником данных компонентов-наборов, и в этом смысле он подобен компоненту `TDatabase` в архитектуре BDE.NET.

Командные компоненты TADOCommand предназначены для реализации запросов на языке определения данных (Data Definition Language, DDL), то есть для реализации SQL-запросов, которые не возвращают данные (запросы типа CREATE, DROP, UPDATE и т. п.). Специальный компонент RDSConnection (не показан на рисунке) создан для упрощения связи с MS Internet Explorer и при разработке интранет-приложений.

Компоненты-наборы с помощью компонентов-источников TDataSource и визуализирующих компонентов TDBGrid, TDBMemo, TDBEdit и т. п. обеспечивают необходимый интерфейс с пользователем программы. Эти компоненты рассмотрены в главе 3.

Пример простой программы

В этом разделе рассматриваются основные особенности использования технологии dbGo.NET. Описанный далее пример (проект Ch04\Nakls\Project1.dpr) в функциональном плане повторяет пример, рассмотренный в разделе «Пример простой программы» главы 1: в нем иллюстрируется реляционная связь главный—детальный между таблицами NAKLS и MOVES:

1. Итак, начните новый VCL-проект и поместите на форму компонент ADOConnetion, 5 компонентов ADOTable (категория dbGo палитры компонентов) и 2 компонента DataSource (категория Data Access). Назовите таблицы именами Nakls, Moves, Firms, Books, Types и свяжите источник данных DataSource1 с таблицей Nakls, а источник данных DataSource2 — с таблицей Moves.
2. Поскольку предполагается использовать файл-серверные таблицы типа Paradox, такие таблицы должны управляться драйвером ODBC. Этот драйвер следует предварительно настроить на работу с нужным каталогом. Для этого в главном меню выберите команду Пуск ▸ Настройка ▸ Панель управления ▸ Администрирование и щелкните на значке Источники данных (ODBC).
3. На вкладке Пользовательский DSN открывшегося окна выберите в списке Источники данных пользователя пункт Paradox files, как показано на рис. 4.2, и щелкните на кнопке Настройка.

ВНИМАНИЕ

В некоторых конфигурациях Delphi устанавливает не все драйверы ODBC. В этом случае в списке на вкладке Пользовательский DSN вы можете не увидеть пункта Paradox files. Если это так, щелкните на кнопке Добавить и в списке открывшегося окна выберите пункт Microsoft Paradox Driver. В конечном счете будет открыто окно, показанное на рис. 4.2, и вы сможете настроить драйвер. Мне не удалось добиться нормальной работы драйвера Driver do Microsoft Paradox (версия 4.00.6304.00). Зато работают два других драйвера — Microsoft Paradox Driver и Microsoft Paradox Driver-Treiber.

4. В открывшемся окне (рис. 4.3) сбросьте флажок Использовать текущий каталог, щелкните на кнопке Выбор каталога и с помощью нового диалогового окна

выберите каталог размещения файлов, после чего закройте все окна, щелкая на кнопке OK. При назначении каталога в реальной программе очень часто бывает необходимо сослаться на сетевой серверный диск. В этом случае щелкните на кнопке Сеть в окне выбора каталога и введите сетевое имя диска, например: `\\SRV\C`.

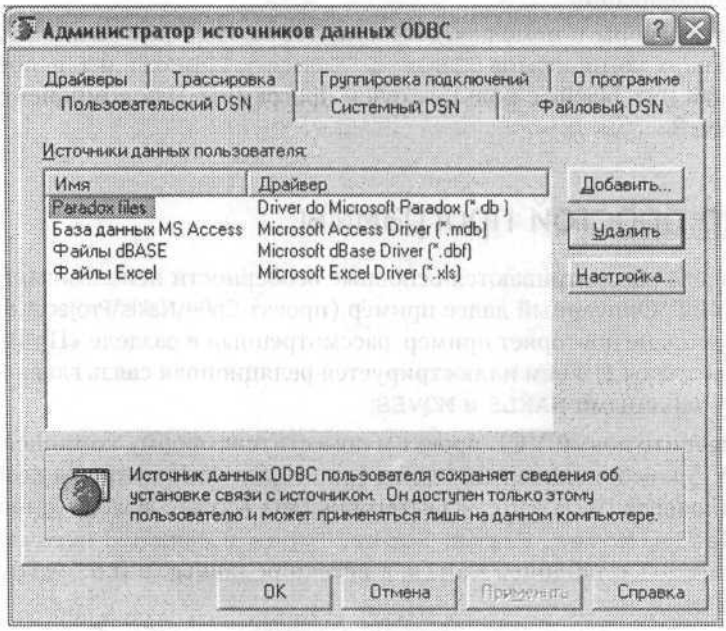


Рис. 4.2. Выбор драйвера ODBC

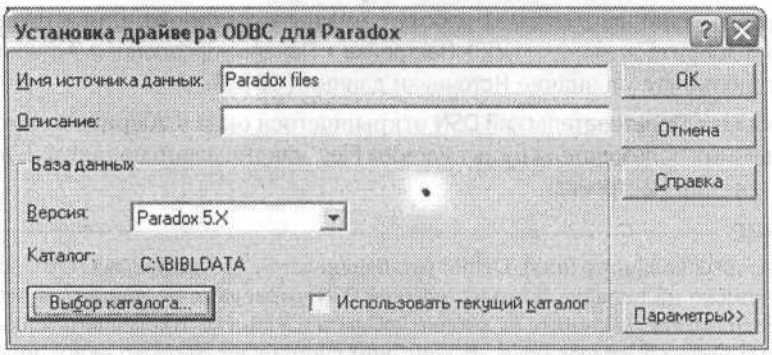


Рис. 4.3. Выбор источника данных для драйвера ODBC

5. Теперь следует настроить связь объектов dbGo.NET с соответствующим провайдером. Для этого дважды щелкните на компоненте `AdoConnection1` или выделите его в окне инспектора объектов и щелкните на кнопке с многоточием в строке свойства `ConnectionString`, чтобы вызвать окно настройки связи (рис. 4.4).

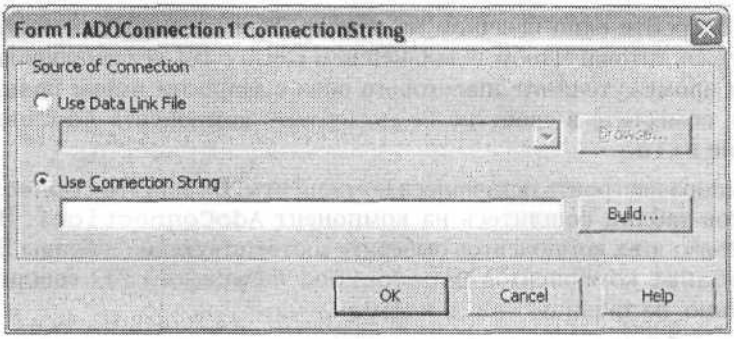


Рис. 4.4. Окно настройки связи

6. Щелкните на кнопке Build и в списке на вкладке Поставщик данных нового окна выберите пункт Microsoft OLE DB Provider for ODBC Drivers (рис. 4.5).

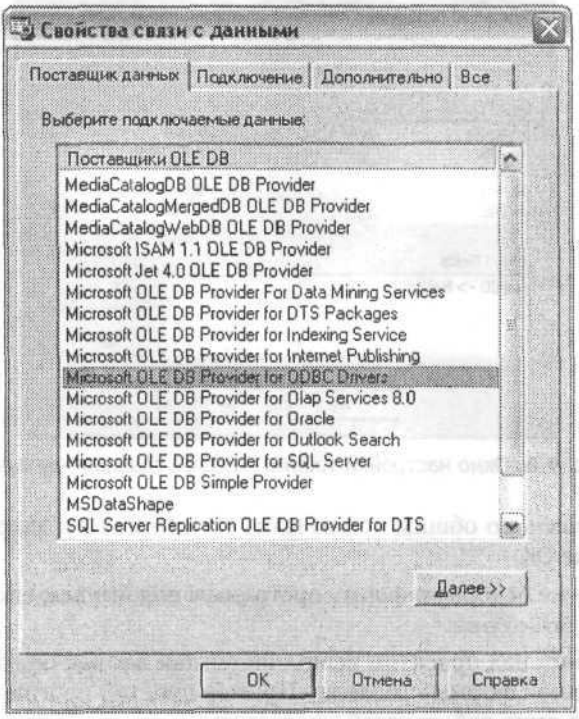


Рис. 4.5. Настройка связи: выбор провайдера

7. Щелкните на кнопке Далее, в новом окне перейдите на вкладку Подключение и, установив переключатель Использовать имя источника данных, выберите в раскрывающемся списке тип используемых данных — пункт Paradox Files.
8. Поскольку ранее мы уже настроили драйвер OBDC, можно щелкнуть на кнопке Проверить подключение, чтобы убедиться в нормальном функционировании

связи. Закройте окно щелчком на кнопке OK и щелкните на кнопке OK еще раз — связь готова! Чтобы в дальнейшем связь с БД устанавливалась без открытия промежуточного диалогового окна с запросом имени пользователя и пароля, поместите в свойство `LoginPrompt` компонента `AdoConnection1` значение `False`.

9. Теперь пора настроить оставшиеся компоненты. В свойстве `Connection` компонентов-таблиц сошлитесь на компонент `AdoConnection1`. В свойстве `TableName` этих компонентов выберите соответствующие таблицы БД. В свойстве `DataSet` компонентов `DataSource1` и `DataSource2` сошлитесь, соответственно, на таблицы `Nakls` и `Moves`.
10. Свяжем таблицы отношением главная—детальная: установите в свойство `MasterSource` таблицы `MOVES` ссылку на источник данных `DataSource1` и щелкните на кнопке с многоточием в строке свойства `MasterFields`, чтобы вызвать окно настройки (рис. 4.6).

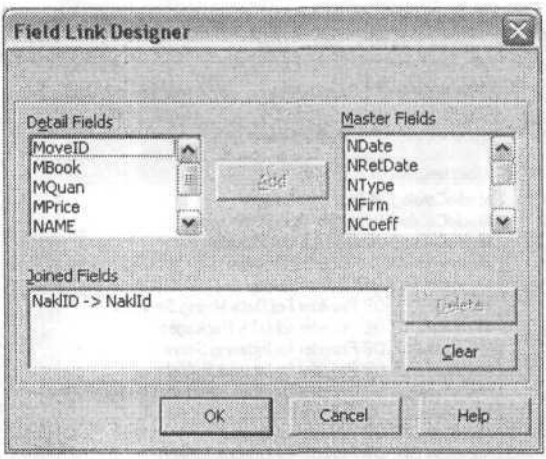


Рис. 4.6. Окно настройки таблиц на связь главная—детальная

11. Установите связь по общим полям `NaklID` и `NaklID` и закройте окно щелчком на кнопке OK.

Сохраните на диске основную форму программы под именем `fmdbGoU` и сам проект под именем `dbGoTest`.

Дальнейшая работа над проектом выполняется так же, как описано в подразделе «Разработка главной формы» раздела «Пример простой программы» главы 1. На рис. 4.7 показано окно работающей программы.

Установление связи с объектом ADO

Установление связи с объектом ADO является ключевым моментом всей технологии. Как уже отмечалось, каждый компонент НД имеет два свойства, с помощью которых он может установить связь с объектом ADO: `Connection` и `Connec-`

tionString. В первое помещается ссылка на специальный связной компонент TAdoConnection, играющий роль концентратора соединения с объектом ADO, во вторую — собственно строка связи. Эти свойства взаимоисключающие, то есть установка значения в одно из них ведет к очистке второго.

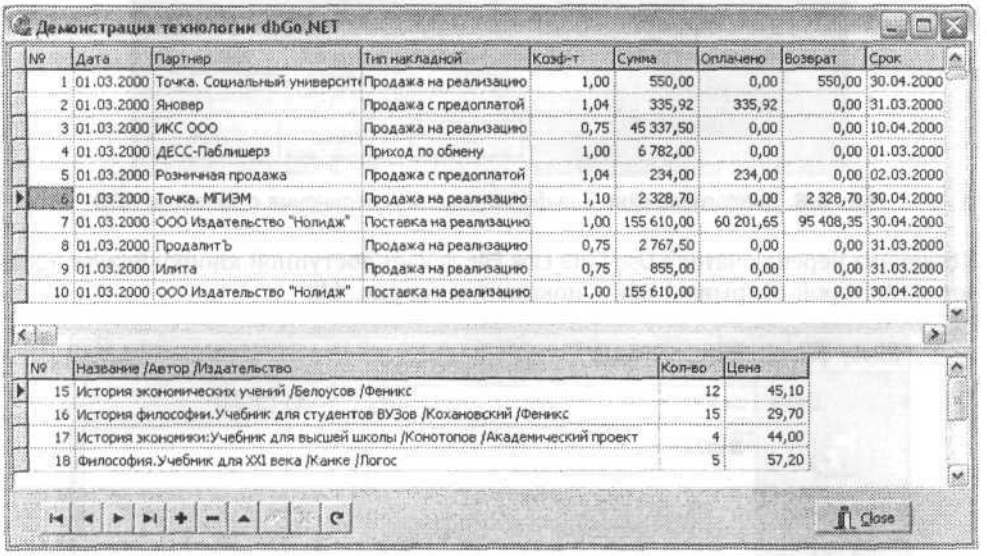


Рис. 4.7. Окно работающей программы

Структура строки связи

Создаваемая тем или иным способом строка связи несет в себе множество (не менее двух) параметров, отделенных друг от друга точкой с запятой. Собственно в технологии dbGo.NET используются лишь четыре из них, остальные нужны для идентификации пользователя при доступе к серверным БД и для настройки некоторых параметров ODBC-драйверов.

Формирование строки связи

При щелчке на кнопке с многоточием в строке свойства ConnectionString компонента TAdoConnection или компонентов-наборов появляется окно, показанное на рис. 4.8.

У программиста есть две возможности: сослаться на специальный связной файл в первой строке или сформировать описание связи во второй. В первом случае можно использовать один и тот же файл сразу для нескольких соединений, поэтому изменение файла отразится на многочисленных связях, причем, возможно, не в одной программе. Изменение содержимого во второй строке окна влияет только на соответствующий компонент-набор или только на те компоненты, которые будут ссылаться на данный связной компонент и только внутри одной программы.

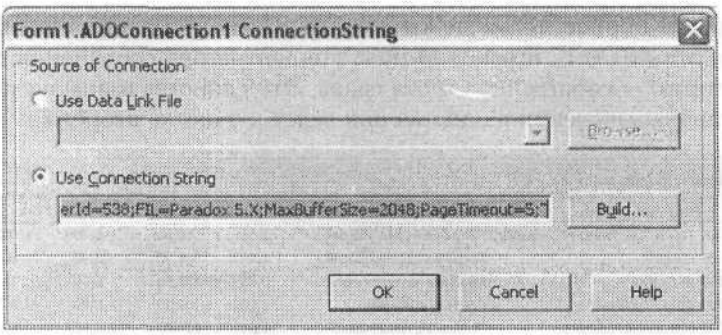


Рис. 4.8. Диалоговое окно для формирования соединения с объектом ADO

Установка переключателя Use Data Link File делает доступной кнопку Browse, щелчок на которой открывает окно, показанное на рис. 4.9.

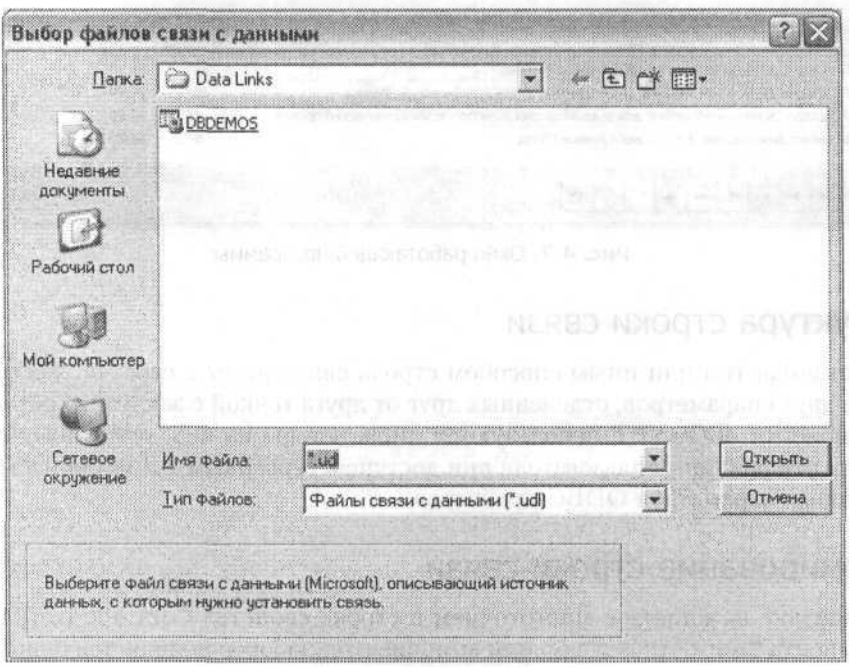


Рис. 4.9. Окно выбора связного файла

Установка переключателя Use Connection String позволяет с помощью кнопки Build воспользоваться специальным диалоговым окном. Вначале рассмотрим диалоговый способ формирования связи.

Диалоговый способ формирования связи

Для диалогового формирования связи установите переключатель Use Connection String и щелкните на кнопке Build. На экране появится диалоговое окно с четырьмя

вкладками. Вкладка Поставщик данных (см. рис. 4.5) используется для выбора механизма, который будет непосредственно взаимодействовать с данными, получая их от клиента и посылая их ему. Фактически в списке этой вкладки отображается состав провайдеров OLE DB, установленных на вашей машине. Выбор провайдера является определяющим фактором. Для разных типов данных должны использоваться только строго определенные провайдеры. Например, в файл-серверных БД должен использоваться провайдер Microsoft OLE DB Provider for ODBC driver с предварительной настройкой соответствующего драйвера. Для работы с БД Access выбирается провайдер Microsoft Jet 4.0 OLE DB Provider. Если применяется сервер Oracle или MS SQL Server, БД работает совместно, соответственно, с провайдером Microsoft OLE DB Provider for Oracle или Microsoft OLE DB Provider for SQL Server и т. д. Некоторые типы баз данных (например, БД InterBase или Informix SQL Server) не имеют провайдеров и поэтому (пока на будут созданы нужные провайдеры) не могут использоваться в технологии dbGo.NET. Поскольку большинство других связных параметров зависят от провайдера, содержимое трех других вкладок также определяется этим фактором. Здесь описывается содержимое вкладок при выборе провайдера Microsoft OLE DB Provider for ODBC driver.

Вкладка Подключение определяет необходимые связные параметры для выбранного провайдера (рис. 4.10).

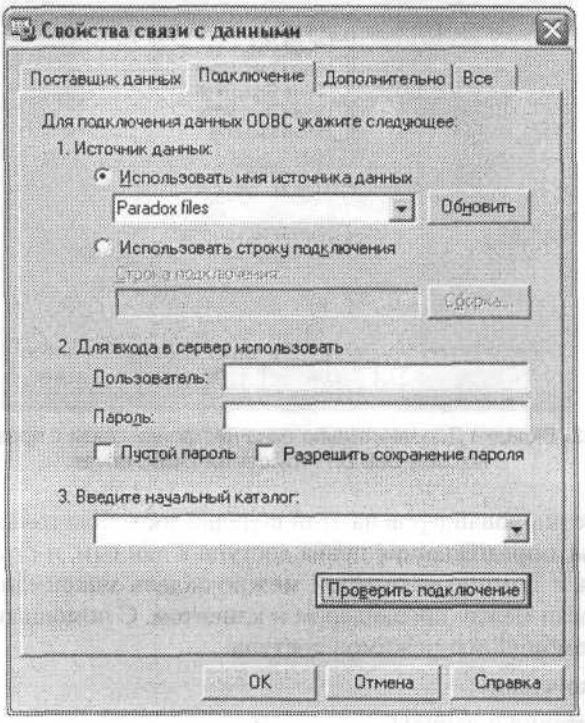


Рис. 4.10. Вкладка Подключение окна настройки связи с провайдером Microsoft OLE DB Provider for ODBC driver

Раскройте список Использовать имя источника данных и выберите в нем нужный источник данных. В качестве дополнительной информации на этой вкладке указываются входное имя пользователя и пароль. Для простейшего варианта связи с Microsoft OLE DB Provider for ODBC driver этих параметров вполне достаточно, поэтому с помощью кнопки Проверить подключение можно протестировать созданную связь. Для других провайдеров может потребоваться задать ряд дополнительных параметров, например имя используемой базы данных, входное имя и пароль для доступа к серверу БД и т. п.

Вкладка Дополнительно позволяет задать некоторые дополнительные свойства связи (рис. 4.11).

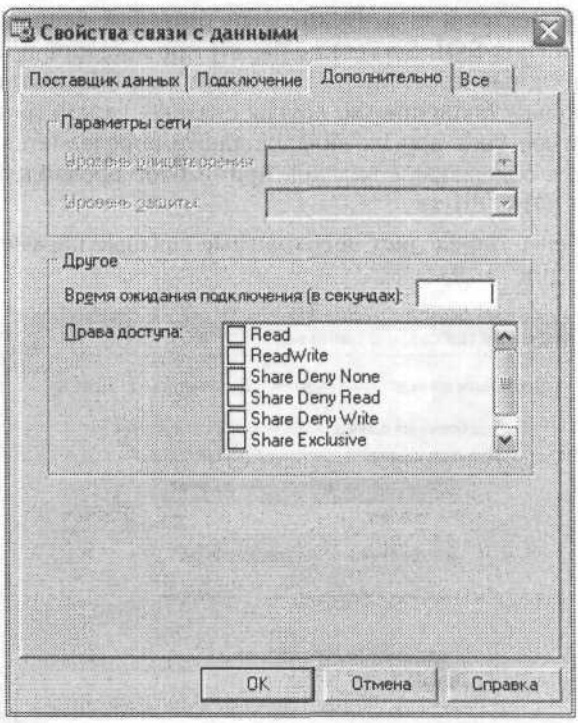


Рис. 4.11. Вкладка Дополнительно окна настройки связи с провайдером Microsoft OLE DB Provider for ODBC driver

Для большей части провайдеров на этой вкладке доступны только флажки в списке Права доступа, определяющие права доступа к данным, и строка Время ожидания подключения, с помощью которой можно задать максимальную паузу при установлении связи между провайдером и клиентом. С помощью флажков можно задать любую комбинацию режимов доступа:

- Read — только чтение;
- ReadWrite — чтение и запись;
- Share Deny None — режим совместной работы невозможен;

- Share Deny Read — нельзя совместно использовать данные, открытые в режиме чтения;
- Share Deny Write — нельзя совместно использовать данные, открытые в режиме записи;
- Share Exclusive — нельзя совместно использовать данные, открытые в режиме чтения и/или записи;
- Write — только запись.

На вкладке Все приводятся все параметры связи — как заданные явно, так и назначенные по умолчанию (рис. 4.12). Щелкнув на кнопке Изменить значение, можно отредактировать значение любого параметра.

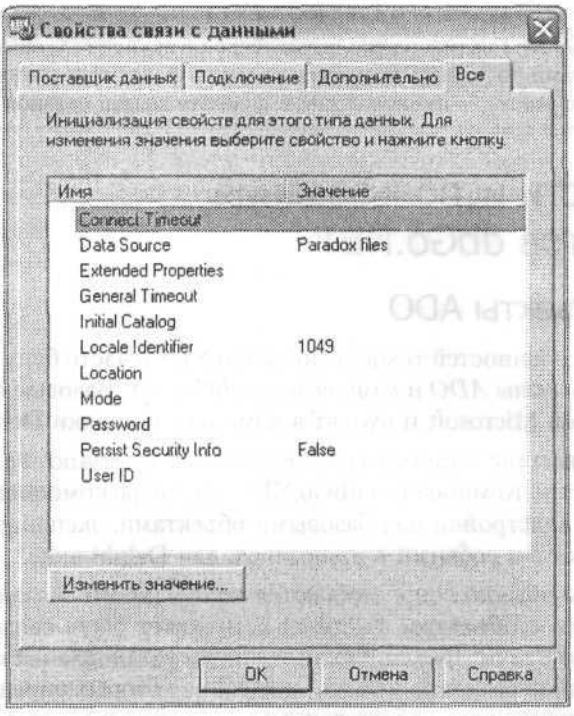


Рис. 4.12. Вкладка Все окна настройки связи с провайдером Microsoft OLE DB Provider for ODBC driver

После щелчка на кнопке OK диалоговое окно формирования параметров связи будет закрыто, и в нижней строке окна формирования соединения с объектом ADO появится соответствующий текст (см. рис. 4.8).

Формирование связного файла

Создать и отредактировать связной файл можно с помощь Проводника Windows (в нерусифицированной версии — Windows Explorer). Вызовите Проводник, выберите папку, в которой будет размещаться связной файл, щелкните на

содержимом папки правой кнопкой мыши и в подменю Создать контекстного меню выберите команду Microsoft Data Link. К сожалению, в ряде систем эта команда недоступна (у меня, например, ее нет в Windows 2000/XP). В этом случае можно порекомендовать не совсем корректный прием. Дело в том, что для демонстрационных целей с Delphi поставляется связной файл DBDEMOS.udl, который по умолчанию располагается в папке Program Files\Common Files\System\ole db\Data Links. Этот файл рассчитан на связь с демонстрационной БД DBDEMOS.mdb, работающей с СУБД Access. Скопируйте этот файл в нужную папку, переименуйте и отредактируйте так, чтобы он описывал связь с вашей БД. Для редактирования связного файла используется окно, показанное на рис. 4.9: дважды щелкните на файле — и вы увидите диалоговое окно с уже знакомыми вкладками (см. рис. 4.10).

ПРИМЕЧАНИЕ

Поскольку dbGo.NET активно использует технологию ActiveX, обмен текстовыми данными с объектами dbGo.NET возможен только в формате `WideString` (по два байта на каждый символ) — именно в таком формате создан связной файл DBDEMOS.udl.

Особенности использования компонентов dbGo.NET

Базовые объекты ADO

При описании особенностей технологии dbGo.NET я часто буду употреблять термины *базовые объекты ADO* и *компоненты dbGo.NET*. Базовые объекты произведены корпорацией Microsoft и входят в комплект поставки Delphi.

Базовых объектов семь: `Connection`, `Recordset`, `Command`, `Parameter`, `Field`, `Error` и `Property`. Компоненты dbGo.NET в палитре компонентов Delphi представляют собой надстройки над базовыми объектами, экспонируя большинство их свойств, методов и событий в привычном для Delphi виде.

Базовые объекты обычно сопровождаются коллекциями связанных с ними объектов. Например, с объектом `Connection` может быть связан один или несколько объектов `Error`, фиксирующих ошибки установления связи; с объектом `Recordset` — набор объектов `Field`, каждый из которых определяет единственное поле результирующего набора данных; с объектом `Command` — один или несколько объектов `Parameter`, конкретизирующих выполнение метода `Execute` этого объекта, и коллекция объектов `Error` и т. д. В связи с этим говорят об основных (`Connection`, `Recordset`, `Command`) и вспомогательных (`Error`, `Field`, `Parameter`, `Property`) объектах ADO.

- Объект `Connection` предназначен для установления соединения с данными — это его главная задача. Кроме того, объект поддерживает механизм транзакций. На него может ссылаться произвольное количество объектов `Command` и `Recordset`. В этом случае `Connection` управляет транзакциями этих объектов. С объектом связан набор объектов `Error`, в котором фиксируются все ошибки, связанные с работой объекта `Connection`.

- Объект Recordset представляет собой текущий набор данных. Он может быть получен только после выполнения метода Execute какого-либо объекта Command. С объектом автоматически связывается набор объектов Field, в которых описываются все поля НД. Объект Recordset способен хранить нужные записи, перемещаться по ним, добавлять, удалять и редактировать записи как в обычном (при одновременном изменении физических ТБД), так и в пакетном режиме (то есть при кэшировании данных). При создании объекта автоматически создается и связанный с ним курсор, обеспечивающий просмотр, редактирование и изменение записей.
- С помощью объекта Command пользователь может выполнить над данными любую SQL-команду. С ним может быть связан набор объектов Parameter, в котором описываются соответствующие параметры, необходимые для выполнения запроса. Характерной особенностью объекта является возможность асинхронного выполнения связанной с ним команды. При обнаружении ошибки с объектом связывается своя коллекция объектов Error.

ПРИМЕЧАНИЕ

Объекту Command соответствуют целых четыре компонента Delphi: TADOCommand, TADOQuery, TADOTable и TADOStoredProc. Сделано это в целях унификации с компонентами категории BDE, хотя даже в этом случае возможности TADOCommand нельзя считать уникальными по сравнению с возможностями TADOQuery и наоборот.

- Объект Parameter определяет единственный параметр, который будет использоваться при выполнении метода Execute объекта Command, — его (параметра) тип, размер и способ применения (входной, выходной, входной и выходной или только для чтения). При необходимости с объектом Command можно связать коллекцию объектов Parameter для указания множества параметров.
- Коллекция объектов Error хранит все ошибки, связанные с работой остальных объектов, и прежде всего объектов Connection, Command и Recordset.
- Объект Field хранит всю необходимую информацию об одном поле НД. Поскольку обычно НД содержит несколько полей, с объектом Recordset связана коллекция объектов Field. С любым полем Field можно связать произвольную коллекцию объектов Property, определяющих индивидуальные характеристики поля.
- Объект Property может быть связан с любым другим объектом ADO, кроме объектов Connection и Error. Он может хранить как статические, так и динамические свойства. Статических (то есть заранее заданных) свойств у объекта всего четыре: Name, Type, Value, Attributes. Остальные свойства динамические и создаются в ходе выполнения программы. В объекте Property задаются некоторые индивидуальные характеристики связанного с ним объекта ADO.

Связной компонент TADOConnection

Компонент TADOConnection осуществляет связь остальных компонентов с данными. Для этих целей у него имеется свойство ConnectionString. После того

как с помощью этого свойства связь с данными установлена, на компонент могут ссылаться другие компоненты dbGo.NET, разделяя установленную им связь.

Однако роль компонента TADOConnection может быть гораздо шире, чем просто концентрация соединений. С помощью своих свойств и методов он может осуществлять точную настройку соединения, обеспечивать необходимый уровень изоляции транзакций, управлять транзакциями и т. д.

Для установления связи нужно с помощью свойства ConncetionString сформировать связные параметры и затем установить значение True в свойство Active или вызвать метод Open. Для разрыва связи выполняется метод Close компонента или в его свойство Active устанавливается значение False.

Компонент содержит в свойстве ConnectionObject ссылку на базовый ADO-объект, с помощью которого и работает сам компонент. Это свойство открывает возможности детального управления связью, если, разумеется, программист хорошо знаком с техникой ADO.

В свойствах CommandCount и DataSetCount содержится количество соответствующих объектов, которые обслуживаются данным компонентом. В сочетании со свойствами Commands и DataSets программист может получить доступ к любому интересующему его объекту. Например:

```
var
  i: Integer;
begin
  for i := 0 to ADOConnection1.DataSetCount - 1 do
    ADOConnection1.DataSets[i].Open;
end;
```

С помощью методов GetProcedureNames и GetTableNames можно получить список всех хранимых процедур и таблиц. Например:

```
AdoConnection1.Open;
AdoConnection1.GetTableNames(ListBox1.Items)
```

Важной особенностью компонента является возможность управления с его помощью транзакциями (подробнее о транзакциях см. главу 1). Для этого в состав компонента добавлены соответствующие методы и события.

С помощью метода BeginTrans стартует новая транзакция, методы CommitTrans и RollbackTrans подтверждают или откатывают транзакцию. Разрешается произвольная глубина вложенности транзакций, то есть после старта одной транзакции может немедленно стартовать следующая и т. д. Уровни разграничения транзакций (свойство IsolationLevel) несколько отличаются от аналогичных уровней BDE.NET и в некоторых случаях могут не поддерживаться сервером БД. Транзакция, стартующая с помощью компонента TADOConection, разделяется всеми другими связанными с ним компонентами. С помощью свойства InTransaction программа может определить, завершилась ли ранее начатая транзакция.

В следующем разделе работа с транзакциями в технологии dbGo.NET рассматривается более подробно.

Работа с транзакциями

Как уже отмечалось, механизм транзакций существенно защищает целостность и непротиворечивость данных в условиях многопользовательской работы. Этот механизм практически бесполезен, если используется файл-серверная БД. Наоборот, все современные серверы поддерживают транзакции, поэтому для иллюстрации особенностей работы с транзакциями в технологии dbGo.NET нам понадобится клиент-серверная БД. Использовавшаяся в предыдущих главах БД «Книголюб» имеется в варианте для сервера InterBase, однако в рамках технологии этот сервер не поддерживается.

Вспомним, что с Delphi 2005 поставляется мощный сервер MS SQL Server. Я прекрасно понимаю — чтобы детально описать его, потребуется отдельная толстая книга. Тем не менее далее описываются некоторые приемы работы с этим сервером. Возможно, этого окажется достаточно для того, чтобы вы смогли самостоятельно создать и наполнить данными собственную БД для этого сервера.

Итак, прежде всего убедитесь в том, что сервер запущен: в системной области панели задач (*systray*) должен быть значок с зеленым треугольником, всплывающая подсказка которого содержит такую строку:

```
Runnung - \\CompName - MSSQLServer
```

Здесь CompName — имя вашего компьютера.

Если значок вместо зеленого треугольника содержит красный квадратик (в этом случае всплывающая подсказка начинается словом Stopping), щелкните на нем правой кнопкой мыши и выберите в контекстном меню команду MSSQLServer — Start. Вся дальнейшая работа идет с утилитой SQL Server Enterprise Manager.

Щелкните на кнопке Пуск и в левой части появившегося меню выберите команду Enterprise Manager, открывающую окно утилиты SQL Server Enterprise Manager (рис. 4.13).

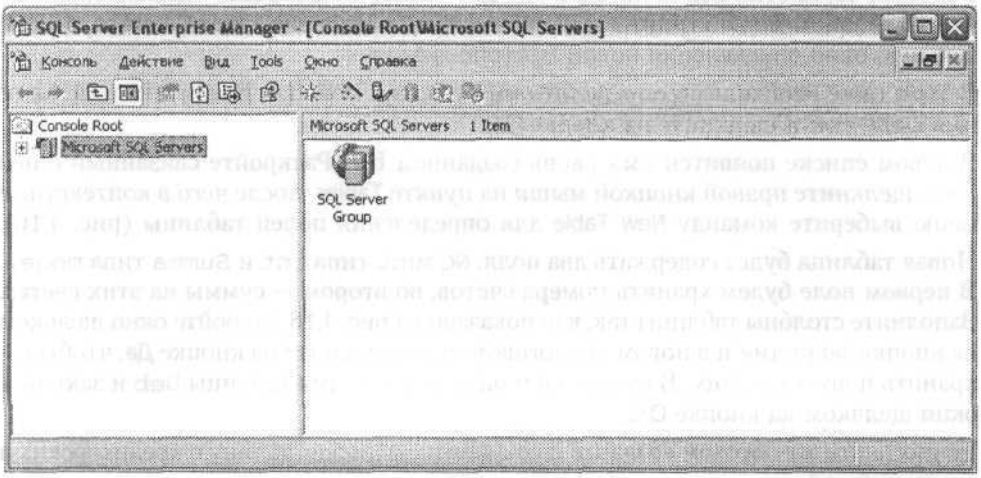


Рис. 4.13. Окно утилиты SQL Server Enterprise Manager

ПРИМЕЧАНИЕ

При установке сервера в главное меню Windows обычно записывается ссылка на утилиту SQL Server Enterprise Manager. Если этой ссылки нет, ищите ее в группе Все программы ▸ MS SQL Server.

В левой части окна последовательно раскройте узлы MS SQL Servers ▸ SQL Server Group ▸ (local) (Windows NT) ▸ Databases. Вы увидите стандартные БД, поставляемые вместе с сервером (рис. 4.14).

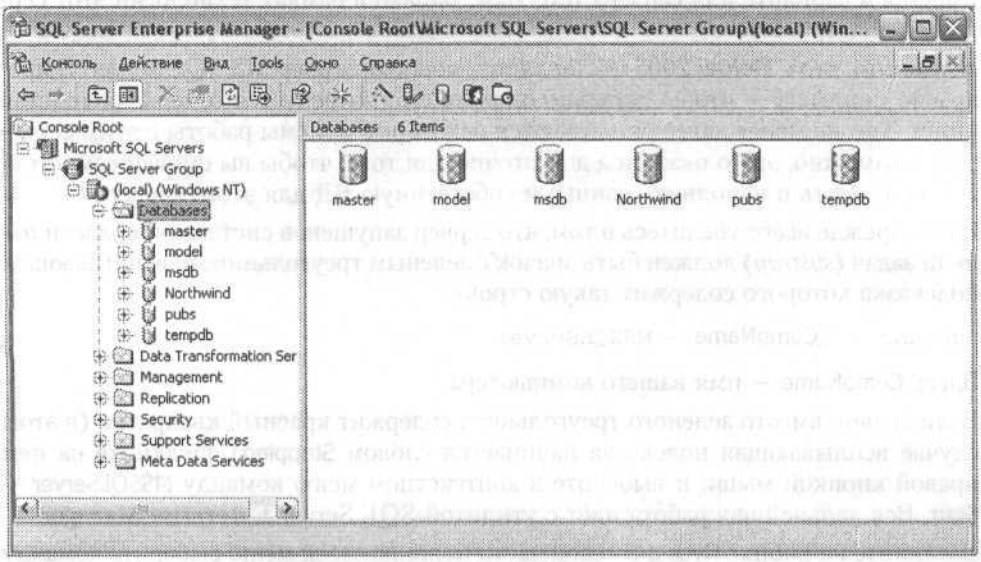


Рис. 4.14. Стандартные базы данных

Бóльшая часть этих БД служебные, но две — Northwind и pubs — демонстрационные. В главном меню утилиты выберите команду Действие ▸ New Database, открывающую окно определения новой БД (рис. 4.15).

В этом окне необходимо определить имя создаваемой БД. Введите в поле Name имя `DebCred` и щелкните на кнопке OK.

В левом списке появится имя вновь созданной БД. Раскройте связанный с ней узел, щелкните правой кнопкой мыши на пункте Tables, после чего в контекстном меню выберите команду New Table для определения полей таблицы (рис. 4.16).

Новая таблица будет содержать два поля: `NCount` типа `int` и `Summa` типа `money`. В первом поле будем хранить номера счетов, во втором — суммы на этих счетах. Заполните столбцы таблицы так, как показано на рис. 4.16, закройте окно щелчком на кнопке закрытия и в новом диалоговом окне щелкните на кнопке Да, чтобы сохранить новую таблицу. В следующем окне введите имя таблицы `Deb` и закройте окно щелчком на кнопке OK.

Точно таким же образом создайте и сохраните на диске таблицу с кредиторскими счетами `Cred`.

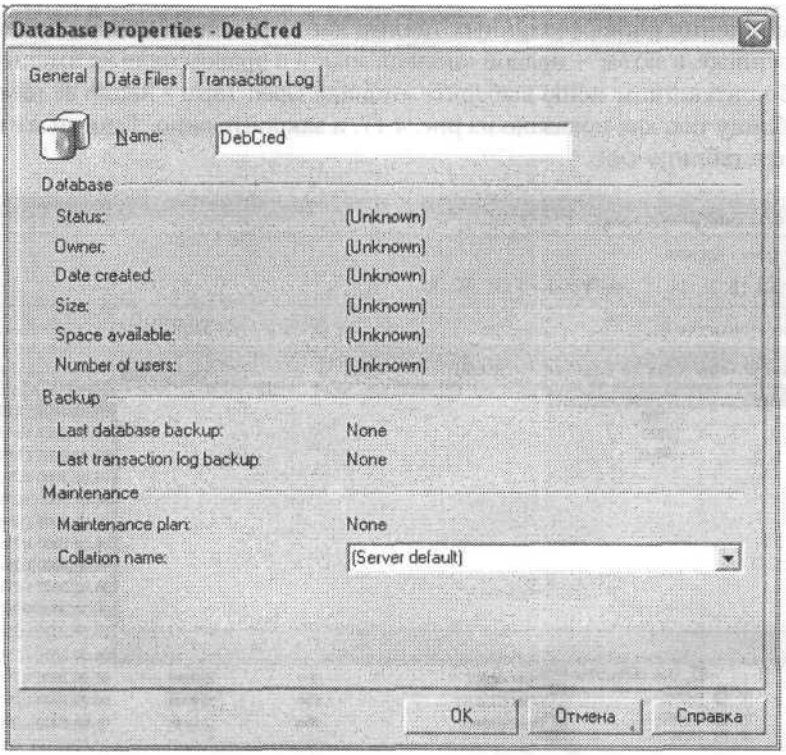


Рис. 4.15. Окно определения новой БД

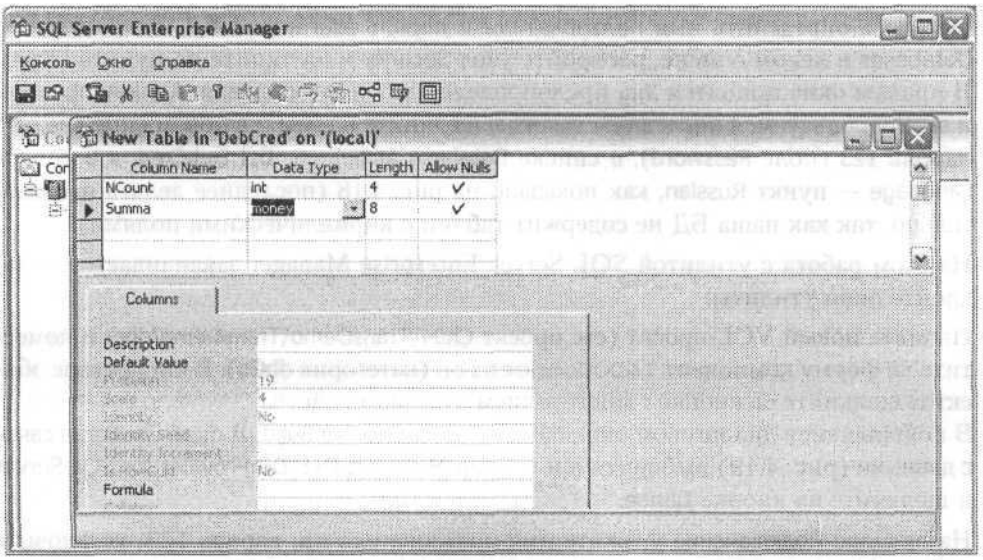


Рис. 4.16. Окно определения полей таблицы

Для наполнения вновь созданных таблиц данными щелкните на пункте Tables в левом списке, а затем — правой кнопкой мыши в правом окне на имени таблицы Cred. В контекстном меню выберите команду Open Table ▸ Return all rows. Заполните таблицу так, как показано на рис. 4.17, и закройте окно. Такими же данными заполните таблицу Deb.

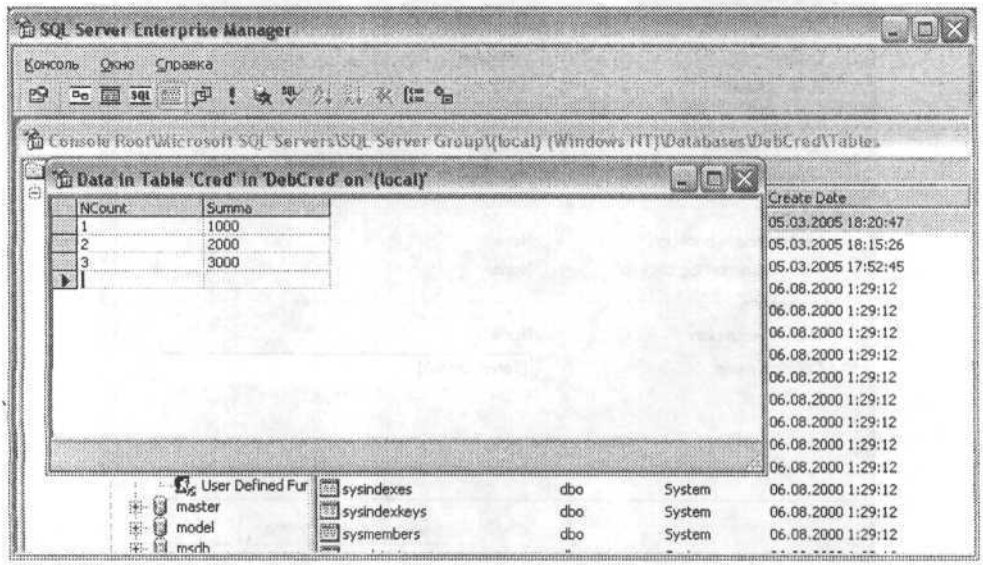


Рис. 4.17. Наполнение таблицы данными

Осталось определить имя пользователя и пароль для новой БД. Закройте узел Databases в левом списке, раскройте узел Security и щелкните на пункте Logins. В правом окне появятся два предопределенных пользователя: Администраторы и sa. Воспользуемся последним: дважды щелкните на нем и в новом окне введите пароль 123 (поле Password), в списке Database выберите БД DebCred, а в списке Language — пункт Russian, как показано на рис. 4.18 (последнее делать необязательно, так как наша БД не содержит таблиц с кириллическими полями).

На этом работа с утилитой SQL Server Enterprise Manager заканчивается — закройте окно утилиты.

Начните новый VCL-проект (см. проект Ch04\TransDemo\TransDemo/dpr) и поместите на форму компонент TADOConnection (категория dbGo). В инспекторе объектов щелкните на кнопке с многоточием в строке свойства ConnectionString. В появившемся диалоговом окне щелкните на кнопке Build. В окне Свойства связи с данными (рис. 4.19) выберите провайдер Microsoft OLE DB Provider for SQL Server и щелкните на кнопке Далее.

На вкладке Подключение укажите имя пользователя sa, пароль 123, установите флажок Разрешить сохранение пароля, раскройте список Выберите базу данных на сервере и выберите БД DebCred (рис. 4.20).

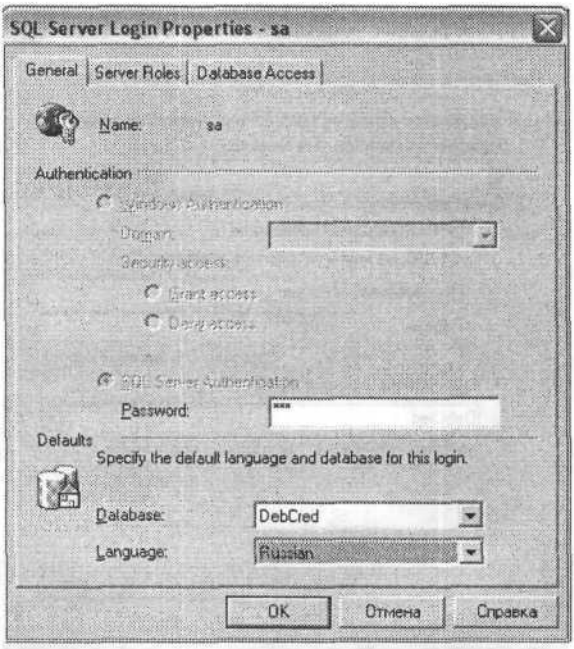


Рис. 4.18. Имя пользователя и пароль для новой БД

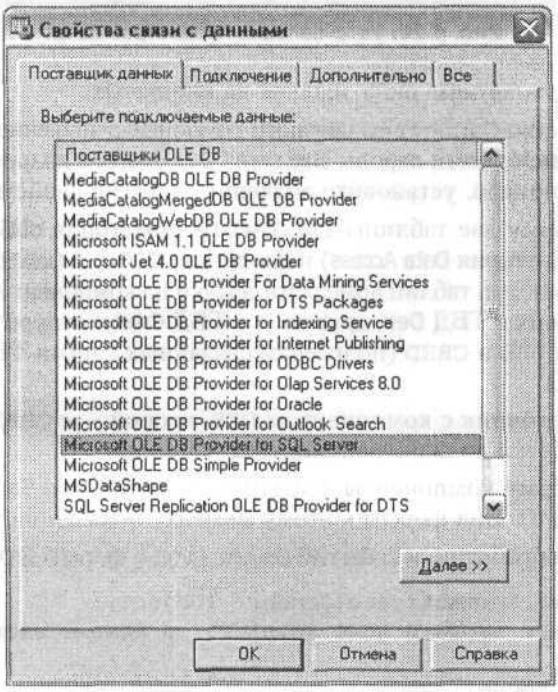


Рис. 4.19. Выбор провайдера

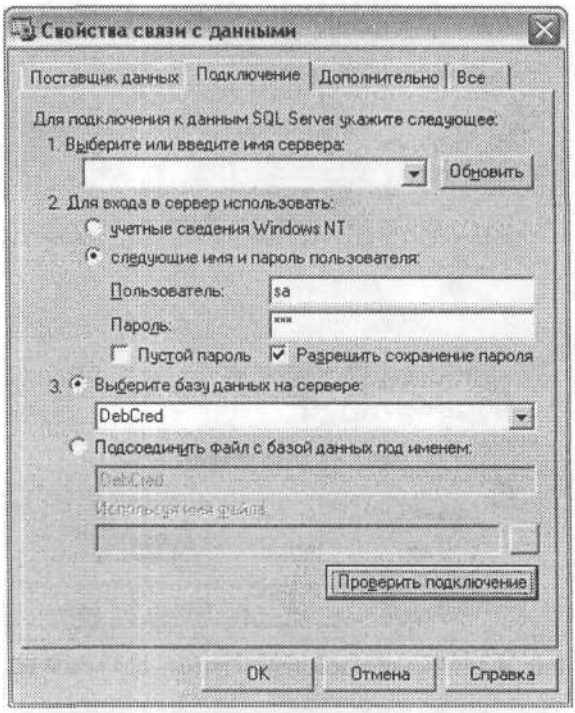


Рис. 4.20. Установление параметров связи

Закрывайте все диалоговые окна, щелкая на кнопке OK.

Поместите в свойство `Connected` связного компонента значение `True` — появится окно с предложением ввести пароль. Введите `123`. Чтобы в дальнейшем окно запроса пароля не возникало, установите значение `False` в свойство `LoginPrompt`.

Поместите на форму две таблицы `TADOTable` (категория dbGo), два источника `TDataSource` (категория Data Access) и две сетки `TDBGrid` (категория Data Controls). В свойства `Connection` таблиц поместите ссылки на компонент `ADOConnection1`, свяжите одну из них с ТБД Deb, другую — с ТБД Cred и откройте таблицы. Назовите компоненты `DEB` и `CRED` (по именам связанных с ними ТБД). Создайте для них объекты-поля.

Свяжите один источник с компонентом `DEB`, второй — с `CRED`. Свяжите сетки с источниками. Сетки должны наполниться данными.

Поместите на форму компоненты `TSpinEdit` (категория Samples) и `TButton` (категория Standard). Вид формы к этому моменту показан на рис. 4.21.

Напишите такие обработчики событий `OnActivate` формы и `OnClick` кнопки:

```
procedure TForm1.FormActivate(Sender: TObject);
// Передает фокус ввода в поле SpinEdit1 в момент активизации формы
begin
  SpinEdit1.SetFocus;
end;
```

Рис. 4.21. Вид формы на этапе конструирования

```
procedure TForm1.Button1Click(Sender: TObject);
// Осуществляет перевод суммы под защитой транзакции
var
  Com: TADOCommand;
  ComStr: String;
  Sum: Integer;
begin
  // Проверяем сумму перевода:
  if SpinEdit1.Value <=0 then
    Exit;                          // Нет ввода или сумма отрицательная
  if SpinEdit1.Value > DEBSumma.Value then
  begin                            // Слишком большая сумма
    SpinEdit1.Value := Round(DEBSumma.Value);
    SpinEdit1.SetFocus;
    Exit;
  end;
  Sum := SpinEdit1.Value;
  // Создаем объект TADOCommand:
  Com := TADOCommand.Create(Self);
  // Указываем ему связной компонент:
  Com.Connection := AdoConnection1;
  // Формируем строку для снятия денег:
  ComStr := Format('UPDATE DEB SET SUMMA=%d WHERE NCOUNT=%d',
                   [Round(DEBSumma.Value - Sum), DEBNCount.Value]);
  // Помещаем ее в ADOCommand:
  Com.CommandText := ComStr;
  // Формируем строку для начисления денег:
  ComStr := Format('UPDATE CRED SET SUMMA=%d WHERE NCOUNT=%d',
                   [Round(CREDSumma.Value + Sum), CREDNCount.Value]);
  ADOConnection1.BeginTrans;  // Стартуем транзакцию
  try
    Com.Execute;                   // Снимаем деньги с дебиторского счета
    Com.CommandText := ComStr;  // Изменяем команду
```

```
    Com.Execute;                    // Начисляем на кредиторский счет
    AdoConnection1.CommitTrans;     // Подтверждаем транзакцию
  except                            // Ошибка!
    AdoConnection1.RollbackTrans;   // Откатываем транзакцию
    ShowMessage('No!');
  end;
  // Обновляем наборы данных:
  DEB.Active := False;
  DEB.Active := True;
  CRED.Active := False;
  CRED.Active := True;
  // Готовим поле ввода:
  SpinEdit1.Value := 0;
  SpinEdit1.SetFocus;
end;
```

Компонент TADOCommand

Компонент TADOCommand предназначен в основном для реализации SQL-запросов, не возвращающих никаких данных. К предложениям DDL (Data Definition Language — язык определения данных) относятся практически все запросы, которые не начинаются зарезервированным словом **SELECT**.

ПРИМЕЧАНИЕ

Хотя конкретная реализация транслятора SQL зависит от выбранного провайдера, в целом компоненты dbGo.NET следуют промышленному стандарту SQL92.

Исполнение подобного рода запросов идет несколько иначе, чем запросов **SELECT**. В BDE-ориентированных компонентах TQuery для реализации запросов **SELECT** используется метод Open (или свойство Active), в то время как DDL-запросы выполняются методом ExecSQL. В dbGo.NET для этих целей выделен специальный компонент. Хотя, как мы увидим дальше, он способен при некоторых обстоятельствах возвращать наборы данных, а компонент TADOQuery имеет в своем составе метод ExecSQL, позволяющий ему выполнять DDL-запросы. Иными словами, одни и те же запросы в рамках dbGo.NET можно выполнять с помощью двух разных компонентов, TADOCommand и TADOQuery, вернее, даже трех компонентов, поскольку рассмотренный ранее связной компонент TADOConnection также способен выполнять команду, например:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  AdoConnection1.ConnectionString :=
    'Provider=Microsoft.Jet.OLEDB.4.0; Data Source='+
    'C:\Program Files\Common Files\Borland Shared\DATA\dbdemos.mdb';
  AdoConnection1.Connected := True;
  AdoDataSet1.RecordSet :=
    AdoConnection1.Execute('SELECT * FROM CUSTOMER')
end;
```

Текст исполняемой команды хранится в свойстве CommandText компонента. Компонент способен за один прием исполнять одну и только одну команду. Особенностью компонента TADOCommand является специализированный текстовый редактор, с помощью которого можно сформировать команду (рис. 4.22). Этот редактор вызывается щелчком на кнопке с многоточием в строке свойства CommandText инспектора объектов.

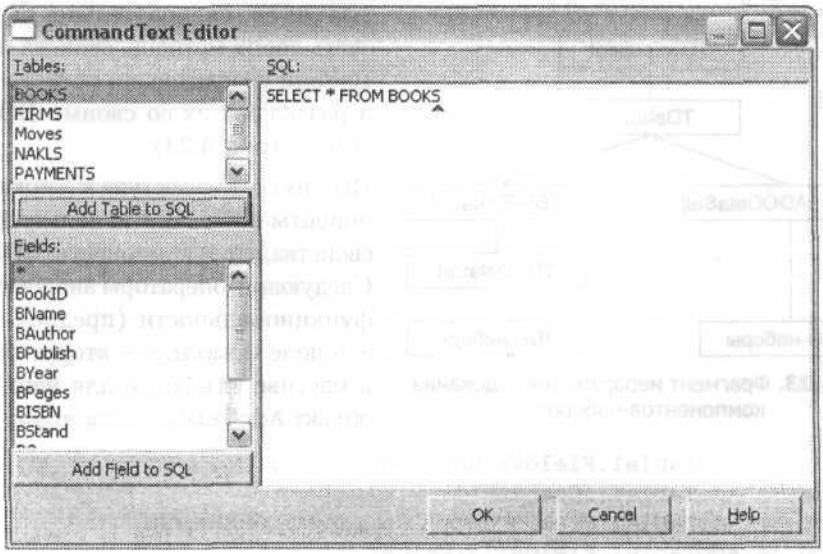


Рис. 4.22. Редактор свойства CommandText компонента TADOCommand

Все окно редактора разделено на три части. В левой верхней части отображается список таблиц БД, с которой связан компонент, в нижней левой — список полей для выделенной таблицы, всю остальную часть занимает собственно текстовый редактор. Справочные окна в левой части лишь облегчают набор текста, который в основном формируется вручную в правом поле. Например, необходимо вручную ввести слово **SELECT**, выделить в нижнем списке строку * и щелкнуть на кнопке Add Field to SQL. Затем вновь вручную вводится слово FROM, в верхнем списке выделяется таблица BOOKS, выполняется щелчок на кнопке Add Table to SQL и т. д.

Как уже упоминалось, компонент TADOCommand способен возвращать записи. Для этого в него включены целых три реализации метода Execute, две из которых как раз и предназначены для создания наборов записей. Использовать возвращаемый НД можно с помощью компонента-посредника TADODataSet по следующей схеме:

```
AdoDataSet1.RecordSet := AdoCommand1.Execute;
```

Для создания НД множество ExecuteOptions не должно содержать значения eoExecuteNoRecords.

Свойства, методы и события компонентов-наборов

Общие с компонентами BDE свойства

В состав компонентов dbGo.NET входят 4 компонента-набора: `TADODataSet`, `TADOTable`, `TADOQuery` и `TADOStoredProc`. Как и аналогичные BDE-ориентированные компоненты, они имеют общего родителя — абстрактный класс `TDataSet`, следовательно, большую часть своих методов, свойств и событий они унаследовали от этого класса и разделяют их со своими BDE-аналогами (рис. 4.23).

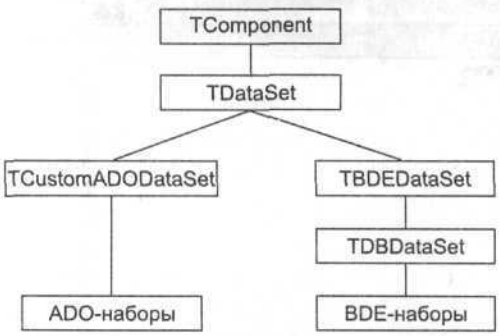


Рис. 4.23. Фрагмент иерархии наследования компонентов-наборов

Например, для доступа к данным компоненты dbGo.NET используют те же свойства, что и компоненты BDE.NET. Следующие операторы аналогичны по функциональности (предполагается, что поле `Company` — второе по счету в массиве `Fields` и для него создан объект `AdoTable1Company`):

```
Caption := AdoTable1.FieldValues['Company'];
Caption := AdoTable1['Company'];
Caption := AdoTable1.FieldByName('Company').AsString;
Caption := AdoTable1.Fields[1].Value;
Caption := AdoTable1Company.AsString;
```

Разумеется, к моменту использования этих операторов компонент `AdoTable1` должен быть связан через провайдер с физической таблицей БД и открыт.

Модификация данных и навигация по ним также не отличаются от описанных ранее:

```
AdoTable1.Edit;
AdoTable1Company.Value := 'Новая компания';
AdoTable1.Post;
```

Или:

```
AdoTable1.Open;
while not AdoTable1.EOF do
begin
  ...
  AdoTable1.Next
end;
```

Наборы dbGo.NET визуализируют данные так же, как BDE.NET-ориентированные наборы: через компоненты-посредники `TDataSource` (категория Data Access) и компоненты категории Data Controls палитры компонентов Delphi.

Специфические свойства

Наборы dbGo.NET имеют непосредственного родителя `TCustomAdoDataSet`, от которого они унаследовали многие специфические свойства. Здесь дается общий обзор особенностей наборов dbGo.NET, а также описываются наиболее важные свойства, методы и события класса `TCustomADODataSet`.

Прежде всего — это свойства `Connection` и `ConnectionString`, с помощью которых набор dbGO.NET может самостоятельно связаться с источником данных.

С помощью свойства `BlockReadSize` программист может организовать такой режим работы, когда навигация по данным не приводит к изменениям в связанных с набором визуализирующих компонентах. Для этого в свойстве `BlockReadSize` следует поместить любое ненулевое значение. В этом случае свойство `State` автоматически примет значение `dsBlockRead`, и навигация по НД пойдет значительно быстрее. Чтобы отменить этот режим, достаточно поместить в свойство `BlockReadSize` значение 0.

В отличие от BDE.NET, наборы dbGo.NET могут создавать курсоры двух типов: на стороне клиента и на стороне сервера (свойство `CursorConnection`). В сочетании со свойством `CursorType` это дает программисту гибкий инструмент влияния на скоростные качества передачи данных и разграничение доступа к данным.

С помощью свойства `EnabledBCD` программист может заменить вещественные данные форматом двоично-десятичных данных (`BCD`). Этот формат позволяет избежать ошибок округления вещественных чисел.

Новым по сравнению с BDE.NET-ориентированными компонентами является также свойство `LockType`, позволяющее повысить скорость работы в многопользовательской среде за счет учета специфики БД.

С помощью свойства `MarshalOptions` можно несколько снизить нагрузку на сеть при создании курсора на клиентской машине.

Важным отличием является то, что компоненты dbGo.NET имеют свойство `RecordSet`, содержащее ссылку на одноименный базовый объект. Если программист хорошо знаком с базовыми объектами ADO, он может напрямую обратиться к этому свойству для более гибкого доступа к данным.

Компоненты dbGo.NET способны работать с кэшированными данными, но делают это по-своему. Во-первых, в терминологии dbGo.NET вместо термина «кэширование данных» используется понятие «пакетная обработка данных», после завершения которой все сделанные в пакете изменения либо подтверждаются, либо отвергаются. Для работы в пакетном режиме набор dbGo.NET должен:

- в свойстве `CursorType` содержать значение `ctKeySet`, заданное по умолчанию, или значение `ctStatic`;
- в свойстве `LockType` содержать значение `ltBatchOptimistic`.

Кроме того, сами данные должны быть получены с помощью SQL-запроса **`SELECT`**.

Вот как, например, готовится компонент TADODataSet к работе в пакетном режиме:

```
with ADODataSet1 do
begin
  CursorLocation := clUseServer;
  CursorType := ctKeyset;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Books';
  Open;
end;
```

Замечу, что, поскольку компонент TADOTable не имеет свойств CommandType и CommandText, его перевод в режим пакетной обработки данных имеет свою специфику:

```
AdoTable1.LockType := ltBatchOptimistik;
AdoTable1.Open;
```

В пакетном режиме можно по этой же схеме выполнять кэшированные изменения в хранимых процедурах, которые возвращают набор записей:

```
AdoStoredProc1.ProcedureName := 'MyBatchQuery';
AdoStoredProc1.LockType := ltBatchOptimistic;
AdoStoredProc1.Open;
```

После получения пакетного набора записей с помощью свойства RecordStatus можно проверить статус конкретной записи, а с помощью свойства FilterGroup — отфильтровать записи. Подтвердить изменения можно с помощью метода UpdateBatch; прекратить режим пакетной обработки и отказаться от сделанных изменений — с помощью метода CancelBatch. При этом в передаваемых этим методам параметрах можно отменить или подтвердить изменения для всех или только для отфильтрованных записей.

Есть определенная специфика в механизме сортировки записей набора dbGo.NET. Для сортировки в свойство Sort помещают список полей сортировки, разделенных запятыми. Каждое поле может дополнительно снабжаться признаком ASCENDING (ASC) или DESCENDING (DESC) для указания, соответственно, восходящего или нисходящего порядка сортировки (если ни одно из этих слов не указано, реализуется восходящая сортировка), например:

```
ADOQuery1.Sort := 'LastName ASC, DateDue DESC';
```

Если в свойство Sort помещена пустая строка, записи не сортируются и предъявляются в том виде, в котором они получены из БД. Если для НД создан клиентский курсор и нет соответствующих индексов для сортировки, на клиентской машине создается временный индекс, который уничтожается при изменении свойства Sort.

Методы класса TCustomADODataSet

Любопытной особенностью наборов dbGo.NET являются инкапсулированные в них методы SaveToFile и LoadFromFile. В dbGo.NET эти методы используются

в качестве одного из возможных механизмов обмена данными между разными компьютерами, а также для отложенной обработки данных. Перед вызовами методов набор dbGo.NET должен быть закрыт. После успешного вызова LoadFromFile набор автоматически открывается в том состоянии, в котором он был сохранен методом SaveToFile.

Наборы dbGo.NET можно сортировать по закладкам (метод FilterOnBookmarks). Например:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BM1, BM2: TBookmarkStr;
begin
  with ADODataSet1 do
  begin
    BM1 := Bookmark;
    MoveBy(3);
    BM2 := Bookmark;
    FilterOnBookmarks([BM1, BM2]);
  end;
end;
```

Для работы в многопользовательском режиме может оказаться полезным метод Clone, с помощью которого уже созданный НД дублируется в вызывающий. Например:

```
AdoTable1.Open;                                  // Создан исходный НД
AdoTable2.Clone(AdoTable1, ltReadOnly); // Создан дубликат в НД
                                                 // AdoTable2
```

Для наборов dbGo.NET определен метод Requery, с помощью которого программист может обновить НД. Его действие фактически эквивалентно последовательному вызову методов Close и Open. Особенностью является то обстоятельство, что влияющие на НД свойства, такие как CursorLocation, CursotType, LockTable и т. п., остаются неизменными, в то время как в промежутке между использованием методов Close и Open эти свойства можно изменить и тем самым повлиять на результирующий НД.

Весьма необычным является метод Seek, осуществляющий поиск записи по заданному значению (значениям) ключевого поля (полей). В принципе, он реализует те же действия, что и метод Locate, но отличается от него несколькими важными особенностями. Во-первых, для Locate не имеет значения, индексировано ли поле (поля), по которому (которым) ищутся записи. В то же время Seek ищет только по индексированным полям, которые к тому же должны быть указаны в свойстве IndexNames или IndexFields. Далее, для Locate курсор не перемещается, если поиск неудачен, для Seek — перемещается всегда в зависимости от указанного параметра SeekOption (если запись не найдена, курсор перемещается в начало НД, в его конец, в место непосредственно перед искомой записью или сразу за ней). И наконец, Seek не может вести поиск с игнорированием возможной разницы в регистре букв или по частичному совпадению поисковых

значений. К сожалению, многие провайдеры OLE DB не допускают установки значений свойства IndexNames или IndexFields и, таким образом, не поддерживают метод Seek.

С помощью метода Supports можно проверить различные характеристики набора, в частности, поддерживает ли он метод Seek:

```
AdoTable1.Open;
if Supports([coSeek]) then
  AdoTable1.Seek(90001,soAfterEQ) else
if Supports([coLocate]) then
  AdoTable1.Locate('Number',90001,[]);
```

События класса TCustomADODataSet

Некоторые события используют следующие типы данных, определяющие статус и смысл события:

```
type TEventStatus = (esOK, esErrorsOccured, esCantDeny,
                     esCancel, esUnwantedEvent);
```

Здесь:

- esOK — нет ошибок;
- esErrorsOccured — событие вызвало ошибки;
- esCantDeny — ожидаемую связь нельзя разорвать;
- esCancel — ожидаемую связь можно разорвать до ее активизации;
- esUnwantedEvent — блокирует связанные события.

```
type TEventReason = (erAddNew, erDelete, erUpdate,
                     erUndoUpdate, erUndoAddNew, erUndoDelete,
                     erRequery, erResynch, erClose, erMove,
                     erFirstChange, erMoveFirst, erMoveNext,
                     erMovePrevious, erMoveLast);
```

Здесь:

- erAddNew — добавлена запись;
- erDelete — удалена запись;
- erUpdate — запись изменена;
- erUndoUpdate — произошел откат изменений записи;
- erUndoAddNew — произошел откат вставки записи;
- erUndoDelete — произошел откат удаления записи;
- erRequery — к НД был применен метод Requere;
- erResynch — к НД был применен метод Resync;
- erClose — НД был закрыт;
- erMove — курсор сместился к другой записи;
- erFirstChange — произошло первое изменение НД;
- erMoveFirst — курсор сместился к первой записи;

- erMoveNext — курсор сместился к следующей записи;
- erMovePrevious — курсор сместился к предыдущей записи;
- erMoveLast — курсор сместился к последней записи.

Особенностью события OnEndOfRecordset является то, что оно предшествует событиям BeforeScroll и AfterScroll. В обработчике события нельзя проверить конец НД с помощью функции EOF, так как сам НД может быть частично не наполнен, если наполнение его записями идет в асинхронном режиме.

Обработчик события OnFetchProgressEvent обычно используется при длительном обращении к БД в асинхронном режиме для индикации процесса. Например:

```
procedure TForm1.ADODataSet1FetchProgress(DataSet:
          TCustomADODataSet; Progress, MaxProgress: Integer;
          var EventStatus: TEventStatus);
begin
  Caption := 'Процент выполнения: ' +
             IntToStr(Trunc(Progress / MaxProgress * 100)) + '%';
  Application.ProcessMessages;
end;
```

Компонент TADODataSet

Компонент TADODataSet обеспечивает доступ к одной или нескольким таблицам БД с помощью запроса типа **SELECT**. Компонент рассчитан на возвращение набора данных, поэтому его нельзя использовать для выполнения подмножества DDL-операторов. (В компоненте есть свойство CommandText, однако в него можно поместить только оператор **SELECT**. Для выполнения DDL-предложений языка SQL можно использовать метод Execute компонента TADOCommand или метод ExecSQL компонента TADOQuery.)

В отличие от компонента TADOTable, компонент TADODataSet может обращаться не только к одной, но сразу к нескольким таблицам. На рис. 4.24, например, показан результат такого запроса к двум таблицам, выполненного с помощью компонента ADODataSet.

Рис. 4.24. Связывание двух таблиц по полю CustNo

Текст запроса:

```
SELECT
  c.CustNo, c.company, o.SaleDate
FROM
  customer c, orders o
WHERE
  o.CustNo=c.CustNo
```

Компонент TADODataSet — единственный компонент-набор, с помощью которого можно установить связь с удаленным источником данных TRDSConnaction при создании трехзвенной архитектуры. Для этого у него определено следующее свойство:

```
property RDSConnection: TRDSConnection;
```

Если компонент связан с единственной таблицей, можно воспользоваться его методом GetIndexNames, чтобы получить список всех имен табличных индексов. Например:

```
AdoDataSet1.GetIndexNames(Memo1.Lines);
```

Остальные свойства, методы и события компонент наследует от своих предков TCustomAdoDataSet и TDataSet.

Компонент TADOTable

Компонент TADOTable является прямым аналогом популярного компонента TTable технологии BDE.NET. Так же как TTable, он способен получать и обслуживать НД, состоящий из записей единственной физической таблицы БД, имя которой содержит его свойство TableName. Являясь, как и все остальные компоненты-наборы, надстройкой к базовому объекту Command, компонент TADOTable имеет свойство CommandText, которое недоступно программисту. Значение этого свойства у него формируется автоматически по имени связанной таблицы:

```
Command.CommandString := 'SELECT * FROM TableName';
```

Здесь TableName — имя связанной таблицы.

Это, однако, не мешает компоненту работать в режиме кэшированных изменений (в режиме пакетной обработки): свойство связанного с ним базового объекта CommandType у него всегда имеет значение cmdText, так что остается изменить лишь единственное свойство LockType (свойство CursorType у компонента автоматически имеет значение ctKeySet):

```
AdoTable1.LockType := ltBatchOptimistic;
AdoTable1.Open;
```

После этого компонент готов к режиму пакетной обработки.

Компонент TADOQuery

В отличие от компонента TADOCommand, компонент TADOQuery преимущественно предназначен для получения набора записей из одной или нескольких таблиц БД. Фактически, он целиком повторяет функциональность компонента TQuery, так как в него включен специфический метод ExecSQL, с помощью которого компонент может выполнять DDL-предложения языка SQL. Сам запрос формируется в свойстве SQL.

Единственным отличием от аналога технологии BDE.NET является следующее свойство, с помощью которого программист может узнать или указать максимальное количество обновлений, которое сделано (или разрешается сделать) при выполнении запроса:

```
property RowsAffected: Integer;
```

Так же как TQuery, компонент TADOQuery имеет свойство DataSource, позволяющее передать параметры запроса от одного компонента к другому.

5 Технология dbExpress.NET

Впервые введенная в Delphi 6, технология dbExpress обеспечивает прямой доступ к некоторым промышленным серверам (InterBase, MySQL, Oracle, Informix, MS SQL Server и DB2) без BDE или другого подобного механизма. Она реализуется в виде набора соответствующих драйверов, учитывающих специфику серверов и обеспечивающих клиенту единый формат взаимодействия с ними. В Delphi 2005 она приспособлена к работе на платформе .NET и называется dbExpress.NET.

Характерной особенностью технологии является создаваемый ее средствами *однонаправленный* курсор НД, что является следствием специфики получения данных от сервера: фактически любой сервер возвращает данные по записям в цикле **FOR...SUSPEND** (см. главу 7), а двунаправленный курсор создается средствами BDE.NET (dbGO.NET, ADO.NET) путем буферизации записей.

Однонаправленный курсор обеспечивает более быстрый доступ к данным и экономит ресурсы клиента, но вместе с тем в большой части практически важных случаев менее удобен. Ниже перечислены ограничения, предъявляемые им к действиям клиента:

- Навигация по НД возможна только от первой записи к последней. В НД могут использоваться только навигационные методы `First` и `Next`. Попытка обращения к любому другому методу (`Prior`, `Last`) вызывает исключение.
- НД не могут сортироваться или фильтроваться (точнее, сортировка и фильтрация реализуются SQL-запросом, а не свойствами `Filter`, `Filtered`, `IndexName` НД).
- Клиент не может визуализировать данные в сетке `TDBGrid`.
- НД не могут редактироваться (их свойство `CanModify` всегда имеет значение `False`).
- К НД нельзя присоединить подстановочные поля.
- В НД запрещено использовать закладки и поиск записей методами `Locate` и `Lookup`.

- Наконец, по результатам выборки нельзя создать отчет главный—детальный, используя технологию Rave Reports.

К счастью, разработчики Delphi создали компонент `TSimpleDataSet`, который буферизует получаемые от сервера данные и создает двунаправленный курсор. Наличие этого компонента практически снимает все перечисленные ограничения и делает технологию dbExpress.NET не менее удобной, чем BDE.NET или dbGo.NET.

Пример простой программы

В следующем примере (проект Ch05\Nakls\dbExpress.dpr) средствами технологии dbExpress.NET воспроизводится уже рассматривавшийся в одноименных разделах предыдущих глав пример отображения связи главный—детальный таблиц `Nakls` и `Moves` из БД «Книголюб»:

1. Начните новый VCL-проект и поместите на форму три панели `TPanel` и вешку разбивки `TSplitter`: две нижние панели и вешку разбивки разместите в нижней части формы (`Align = alBottom`), а верхнюю — в оставшемся пространстве формы (`Align = alClient`).
2. Разместите на форме компонент `TSQLConnection`, два компонент `TSimpleDataSet`, компонент `TSQLQuery` (категория dbExpress палитры компонентов) и два компонента `TDataSource` (категория Data Access). Компоненты `TSimpleDataSet` предназначены для отображения данных, связанных отношением главный—детальный, компонент `TSQLQuery` — для удаления данных из детальной таблицы.
3. Настройте компонент `SQLConnection1` на работу с БД IB_Bibl.gdb: в списке свойства `ConnectionName` выберите значение `IBConnection` — при этом автоматически настроятся некоторые другие свойства компонента; в свойство `LoginPrompt` поместите значение `False`. Щелчком на кнопке с многоточием откройте окно редактора свойства `Params` (рис. 5.1) и в строке `Database` укажите путь доступа к файлу БД (путь должен включать имя хоста, на котором размещен сервер InterBase; если сервер запущен на вашем компьютере, путь должен выглядеть так: `localhost:c:\bibldata\ib_bibl.gdb`).
4. Настройте свойства компонента `TSimpleDataSet1` (он представляет собой главный НД): `Name = Nakls`, `SQLConnection = SQLConnection1`. Раскройте сложное свойство `DataSet`, щелчком на кнопке свойства `CommandText` откройте окно редактора свойства и разместите в нем следующий запрос (рис. 5.2):

```
SELECT
  NaklID, NDate, FName, TName, NSum, NPayedSum, NCoeff, NRetSum
FROM
  Nakls, Firms, TypeNakl
WHERE
  FirmID=NFirm AND TypeID=NType
ORDER BY
  NaklID
```

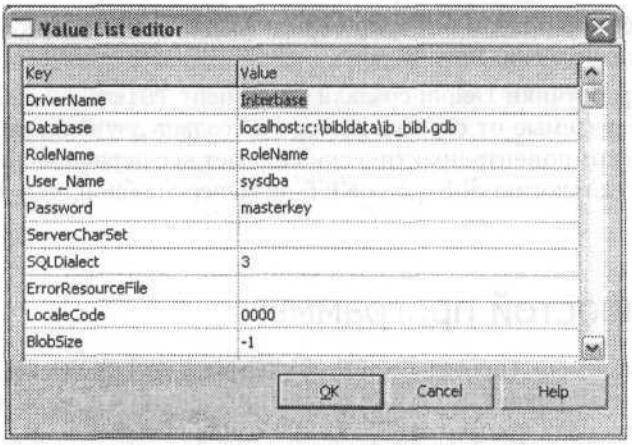


Рис. 5.1. Окно редактора свойства Params

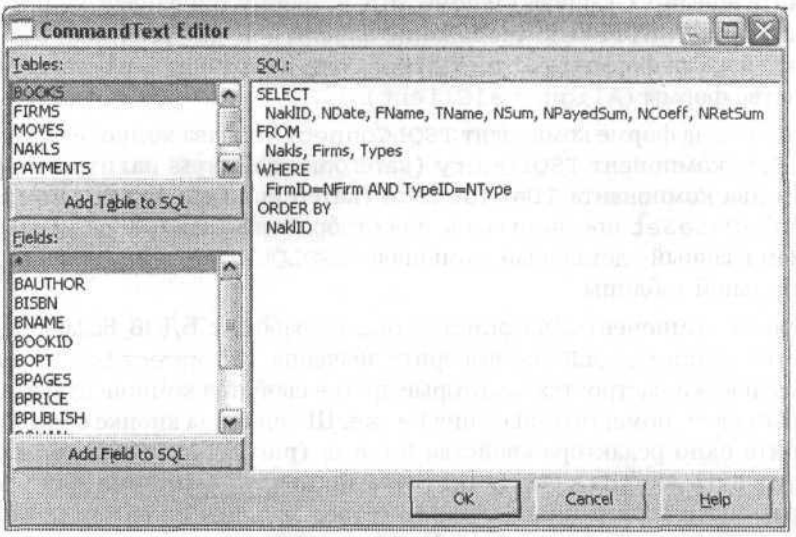


Рис. 5.2. Окно редактора свойства CommandText

5. Откройте НД, создайте для него объекты-поля и свяжите с ним компонент `DataSource1`.
6. Поместите на верхнюю панель сетку `TDBGrid`. В ее свойстве `Align` укажите значение `alClient`, а в свойстве `DataSource` сошлитесь на компонент `DataSource1`. Сетка должна наполниться данными из главного НД.
7. Настройте свойства второго НД `SimpleDataSet2`: `Name` = `Moves`, `Connection` = `SQLConnection1`. Раскройте сложное свойство `DataSet` и с помощью редактора введите в подсвойство `CommandText` такой запрос:

```
SELECT
  MBook, BName, BAuthor, BPublish, MQuan, MPrice, NaklID
```

```
FROM
  Books, Moves
WHERE
  NaklID =: NaklID AND BookID = MBook
ORDER BY BName
```

8. В подсвойстве DataSource свойства DataSet сошлитесь на компонент DataSource1. Обратите внимание: на следующем шаге мы еще раз свяжем детальный НД с главным.
9. В свойство MasterSource НД Moves поместите ссылку на источник DataSource1, вызовите редактор свойства MasterField и установите реляционную связь по столбцам NaklID и NaklID (рис. 5.3).

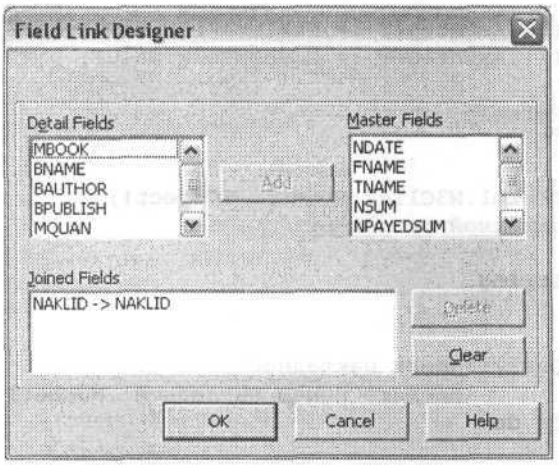


Рис. 5.3. Установление реляционной связи

10. Свяжите с компонентом источник DataSource2, откройте НД и создайте для него объекты-поля.
11. Поместите на среднюю панель сетку TDBGrid, укажите в ее свойстве Align значение alClient, а в свойстве DataSource сошлитесь на источник DataSource2. Сетка должна наполниться данными из детального НД.
12. Для отображения сумм в денежном формате напишите соответствующие обработчики OnGetText для объектов-полей. Например:

```
procedure TForm1.NaklsNSUMGetText(Sender: TField;
                                  var Text: String;
                                  DisplayText: Boolean);
begin
  Text := FloatToStrF(NaklsNSum.Value, ffCurrency, 15, 2)
end;
```

13. Для «листания» накладных разместите на нижней панели компонент TDBNavigator и свяжите его с источником DataSource1.

14. Чтобы продемонстрировать, как можно удалять и искать записи в НД с однонаправленным курсором, создайте два контекстных меню: одно с командой Искать, второе с командой Удалить книгу. Свяжите первое с верхней сеткой, а второе — с нижней. Напишите такие обработчики OnClick выбора команд меню:

```
procedure TForm1.N1Click(Sender: TObject);
// Удаление книги
begin
  if MessageDlg('Вы действительно хотите удалить из накладной '+
                NaklsNaklID.AsString+
                ' книгу '+ MoveBNAME.Value+'?',
                mtWarning, [mbYes, mbNo], 0) = mrYes then
  with DelBook do
  begin
    Params[0].AsInteger := NaklsNAKLID.AsInteger;
    Params[1].AsInteger := MovesMBook.Value;
    ExecSQL;
    Moves.Refresh
  end
end;
procedure TForm1.N3Click(Sender: TObject);
// Поиск накладной по номеру
var
  Number: String;
  N: Integer;
begin
  if InputQuery('Поиск накладной',
                'Введите номер накладной',Number) then
  with Nakls do
  begin
    try
      N := StrToInt(Number);
    except
      ShowMessage('Ошибка в записи числа');
      Exit
    end;
    DataSource1.Enabled := False;
    Locate('NaklID', N, []);
    DataSource1.Enabled := True
  end
end;
```

15. Чтобы реализовать удаление книги, измените следующим образом свойства оставшегося в модуле данных компонента TSQLQuery:
    - Name = DelBook;
    - SQLConnection = SQLConnection1;
    - SQL:

```
DELETE FROM Moves
WHERE NaklID = :ID AND MBook = :Book
```

16. Создайте для компонента два параметра:

```
Params.ID.DataType = ftInteger
Params.Book.DataType = ftInteger
```

17. Вид окна работающей программы показан на рис. 5.4.

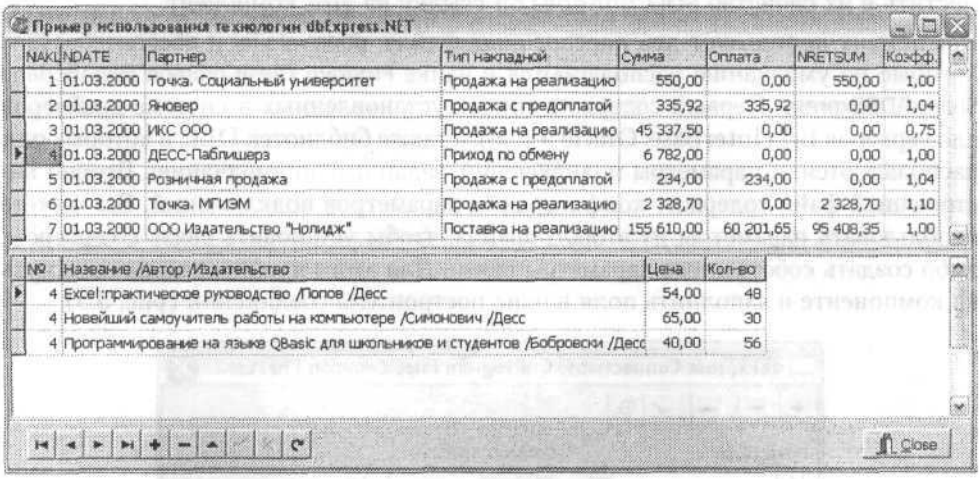


Рис. 5.4. Окно работающей программы

Подведем некоторые итоги. Важнейшим отличием компонента `TSimpleDataSet` от НД технологий BDE.NET и dbGo.NET является буферизация (кэширование) отображаемых им данных. Если вы попробуете с помощью навигатора удалить или изменить любую запись в верхней сетке, будут изменены данные в буфере, но никак не в БД! Реальное изменение данных возможно лишь с помощью соответствующего параметрического запроса `TSQLDataSet`, как это сделано в примере для удаления данных о книге из накладной.

Разумеется, технология dbExpress.NET требует от программиста несколько бóльших усилий, чем рассмотренные ранее технологии BDE.NET и dbGo.NET. Однако в ряде последних заявлений представителей Borland отчетливо прозвучала мысль, что в будущем эта технология станет доминирующей, так как именно она создает «легкое» клиентское место (без громоздкого аппарата BDE и/или необходимости распространения вместе с приложением базовых объектов ADO).

Компоненты для реализации технологии

В основе технологии dbExpress.NET лежат три компонента: связной компонент `TSQLConnection`, универсальный НД `TSQLDataSet` и клиентский НД `TSimpleDataSet`. Компоненты-наборы `TSQLQuery`, `TSQLTable` и `TSQLStoredProc` лишь повторяют свойства аналогичных компонентов других технологий, но не решают главной проблемы — не создают двунаправленных курсоров.

Компонент TSQLConnection

Компонент `TSQLConnection` решает две основные проблемы: выбирает нужный драйвер для связи с сервером и указывает путь доступа к файлу БД. Созданное с его помощью соединение может разделяться многочисленными НД, если поместить в их свойство `SQLConnection` ссылку на этот компонент.

Компонент использует два настроечных файла: dbdrivers.ini и dbconnections.ini, которые по умолчанию располагаются в папке Program Files\Common Files\Borland Shared\DBExpress. Первый содержит список установленных в системе драйверов для серверов БД (InterBase, Oracle и т. д.), а также библиотек DLL, в которых они располагаются, и параметры подключений, заданные по умолчанию. Второй настроечный файл содержит конфигурации параметров подключений. Вы можете использовать параметры из второго файла, чтобы установить связь с сервером, либо создать собственные параметры связи. Для этого нужно дважды щелкнуть на компоненте и заполнить поля в окне построителя соединений (рис. 5.5).

Рис. 5.5. Окно построителя соединений

Свойства компонента `TSQLConnection` перечислены в табл. 5.1.

Таблица 5.1. Свойства компонента TSQLConnection

Свойство	Описание
`property ActiveStatements: Cardinal;`	Содержит общее количество активных запросов для данного подключения. Активным запрос становится после установления в его свойство `Prepared` значения `True` или в момент выполнения неподготовленного запроса. Значение этого свойства не может превышать значения свойства `MaxStmtsPerConn`, если `AutoClone = False`

Свойство	Описание
`property AutoClone: Boolean;`	Разрешает/запрещает автоматически создавать новые соединения, если исчерпано количество активных соединений, указанное в свойстве `MaxStmtsPerConn`
`property Connected: Boolean;`	Открывает/закрывает связь с сервером
`property ConnectionName: String;`	Определяет имя конфигурации параметров соединения
`type TConnectionState = (csStateClosed, csStateOpen, csStateConnecting, csStateExecuting, csStateFetching, csStateDisconnecting);` `property ConnectionState: TConnectionState;`	Определяет состояние соединения: `csStateClosed` — соединение установлено; `csStateOpen` — соединение разорвано; `csStateConnecting` — компонент устанавливает соединение; `csStateExecuting` — компонент передает серверу запрос; `csStateFetchin` — компонент получает информацию от сервера; `csStateDisconnecting` — компонент разрывает связь, но остались незавершенные запросы
`property DataSetCount: Integer;`	Указывает количество обслуживаемых компонентом НД
`property DataSets[Index: Integer]: TCustomSQLDataSet;`	Открывает индексированный доступ ко всем НД, разделяющим данное соединение
`property DriverName: String;`	Определяет имя драйвера для связи с сервером БД
`property GetDriverFunc: String;`	Определяет имя функции, возвращающей интерфейс dbExpress верхнего уровня. Как правило, это свойство содержит имя функции, которое определено в файле dbxdrivers.ini
`property InTransaction: Boolean;`	Содержит `True`, если стартовавшая транзакция не завершена
`property KeepConnection: Boolean;`	Если содержит `True`, соединение с сервером сохраняется, даже если закрыты все обслуживаемые компонентом НД
`property LibraryName: String;`	Содержит имя DLL с драйвером
`property LoadParamsOnConnect: Boolean;`	Определяет, загружает ли компонент конфигурацию, указанную в свойстве `ConnectionName`, непосредственно перед соединением с сервером
`type TLocaleCode = Integer;` `property LocaleCode: TLocaleCode;`	Определяет язык локализации (для России — 1049)
`property LoginPrompt: Boolean;`	Указывает, будет ли появляться диалоговое окно перед установкой соединения с БД

продолжение

Таблица 5.1 (продолжение)

Свойство	Описание
`property MaxStmtsPerConn: Cardinal;`	Определяет максимальное количество активных запросов на одно соединение
`property Params: TStrings;`	Содержит список параметров соединения
`property ParamsLoaded: Boolean;`	Если содержит `True`, параметры соединения будут загружаться из файла dbxconnections.ini
`property SQLHourGlass: Boolean;`	Если содержит `True`, форма курсора будет изменяться во время длительных операций
`Type TTableScope = (tsSynonym, tsSysTable, tsTable, tsView); TTableScopes = set of TTableScope;` `property TableScope: TTableScopes;`	Определяет тип таблиц, которые будут отображаться при выполнении метода `GetTableNames`: `tsSynonym` — синонимы, `tsSysTable` — системные таблицы, `tsTable` — обычные таблицы, `tsView` — представления
`property TransactionsSupported: LongBool;`	Указывает, будет ли сервер базы данных поддерживать транзакции
`property VendorLib: String;`	Содержит имя библиотеки DLL, созданной производителем сервера для поддержки клиентов

Методы компонента `TSQLConnection` перечислены в табл. 5.2.

Таблица 5.2. Методы компонента TSQLConnection

Метод	Описание
`function CloneConnection: TSQLConnection;`	Создает копию соединения
`procedure CloseDataSets;`	Закрывает все обслуживаемые компонентом НД, но не разрывает связь с сервером
`procedure Commit(TransDesc: TTransactionDesc);`	Подтверждает изменения, сделанные в транзакции `TransDesc`, и закрывает ее
`function Execute(const SQL: String; Params: TParams): Integer;`	Выполняет на сервере SQL-команду без использования каких-либо НД. Если в команде не требуются параметры, удобнее использовать метод `ExecuteDirect`
`function ExecuteDirect(const SQL: String ): Integer;`	Выполняет на сервере SQL-команду без параметров
`procedure GetFieldNames(const TableName: String; List: TStrings);`	Заполняет список `List` именами столбцов таблицы `TableName`
`procedure GetIndexNames(const TableName: String; List: TStrings);`	Заполняет список `List` именами индексов таблицы `TableName`

Метод	Описание
`procedure GetProcedureNames (List: TStrings);`	Заполняет список `List` именами хранимых процедур
`procedure GetProcedureParams (ProcedureName: String; List: TList);`	Заполняет список `List` именами параметров хранимой процедуры `ProcedureName`
`procedure Rollback(TransDesc: TTransactionDesc);`	Отменяет изменения, сделанные в транзакции `TransDesc`, и закрывает ее
`procedure SetTraceCallbackEvent (Event: TSQLCallbackEvent; IClientInfo: Integer);`	Связывает с соединением функцию обратного вызова, которая будет вызываться при каждом обмене информацией между клиентом и сервером
`procedure StartTransaction(TransDesc: TTransactionDesc);`	Стартует транзакцию

В листинге 5.1 показан пример использования метода `Execute` для ввода данных о новой книге в таблицу `Books`.

Листинг 5.1. Использование метода Execute для ввода данных о новой книге

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    SQLConnection1.Connected := True;
    stmtParams.CreateParam(ftString, 'Name', ptInput);
    stmtParams.ParamByName('Name').AsString := edName.Text;
    stmtParams.CreateParam(ftString, 'Author', ptInput);
    stmtParams.ParamByName('Author').AsString := edAuthor.Text;
    stmtParams.CreateParam(ftString, 'Publish', ptInput);
    stmtParams.ParamByName('Publish').AsString := edPublish.Text;
    stmtParams.CreateParam(ftString, 'Year', ptInput);
    stmtParams.ParamByName('Year').AsString := edYear.Text;
    stmtParams.CreateParam(ftString, 'Pages', ptInput);
    stmtParams.ParamByName('Pages').AsString := edPage.Text;
    stmtParams.CreateParam(ftString, 'ISBN', ptInput);
    stmtParams.ParamByName('ISBN').AsString := edISBN.Text;
    SQLstmt := 'INSERT INTO Books '+
      '(BName, BAuthor, BPublish, bYear, BPages, BISBN) ' +
      'VALUES (:Name, :Author, :Publish, :Year, :Pages, :ISBN)';
    SQLConnection1.Execute(SQLstmt, stmtParams);
  finally
    stmtParams.Free;
  end
end;
```

Для демонстрации механизма транзакций создадим проект, в котором деньги с дебиторского счета переводятся на кредиторский (проект Ch05\TransDemo\TransDemo.dpr). В качестве таблиц счетов используем таблицы DEB и CRED БД IB_BIBL.GDB.

На рис. 5.6 показано окно формы на этапе конструирования.

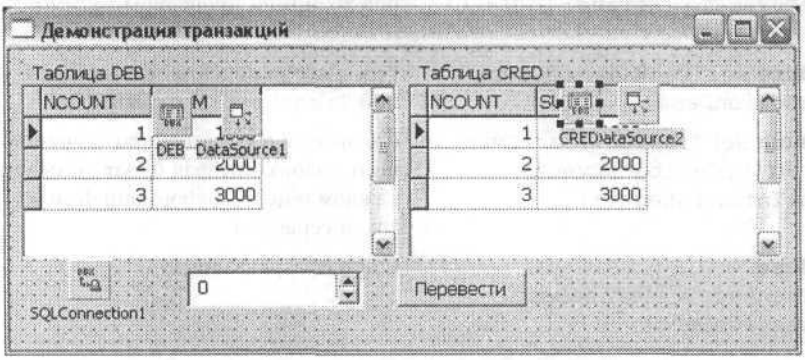


Рис. 5.6. Окно формы на этапе конструирования

На форме располагается связной компонент SQLConnection1, настроенный на связь с БД IB_BIBL:

- ConnectionName = IBConnection;
- DriverName = Interbase;
- Params.Database = localhost:C:\BIBLDATA\IB_BIBL.GDB;
- LoginPrompt = False;
- Connected = True.

Компоненты DEB и CRED — НД класса TSimpleDataSet, связанные с соединением SQLConnection1 и соответствующими таблицами БД. Оба НД открыты, и для них созданы объекты-поля. Сетки DBGrid1 и DBGrid2 через источники DataSource1 и DataSource2 связаны с НД DEB и CRED соответственно.

В нижней части формы располагается поле ввода класса TSpinEdit (категория Samples) и кнопка Button1.

В листинге 5.2 показаны обработчики событий OnActivate формы и OnClick кнопки.

Листинг 5.2. Использование транзакций для перевода денег

```
procedure TForm1.FormActivate(Sender: TObject);
// Передает фокус ввода полю SpinEdit1 в момент активизации окна
begin
  SpinEdit1.SetFocus;
end;

procedure TForm1.Button1Click(Sender: TObject);
// Осуществляет перевод суммы под защитой транзакции
var
  Com: TSQLQuery;
```

```
  ComStr: String;
  Sum: Integer;
  TD: TTransactionDesc;
begin
  // Проверяем допустимость введенной суммы:
  if SpinEdit1.Value <= 0 then
    Exit;                          // Нет ввода
  if SpinEdit1.Value > DEBSumm.Value then
  begin                            // Слишком большая сумма
    SpinEdit1.Value := Round(DEBSumm.Value);
    SpinEdit1.SetFocus;
    Exit;
  end;
  Sum := SpinEdit1.Value;    // Сумма перевода
  // Создаем и настраиваем компонент TSQLQuery:
  Com := TSQLQuery.Create(Self);
  Com.SQLConnection := SQLConnection1;
  // Команда снятия денег со счета:
  ComStr := Format('UPDATE DEB SET SUMM=%d WHERE NCOUNT=%d',
                   [Round(DEBSumm.Value - Sum), DEBNCount.Value]);
  Com.CommandText := ComStr;
  // Команда начисления на кредиторский счет:
  ComStr := Format('UPDATE CRED SET SUMM=%d WHERE NCOUNT=%d',
                   [Round(CREDSumm.Value + Sum), CREDNCount.Value]);
  TD.TransactionID := 1;
  SQLConnection1.StartTransaction(TD);
  try
    Com.ExecSQL;
    Com.CommandText := ComStr;
    Com.ExecSQL;
    SQLConnection1.Commit(TD);
  except
    SQLConnection1.Rollback(TD);
    ShowMessage('No!');
  end;
  // Обновляем данные в сетках:
  DEB.Active := False;
  DEB.Active := True;
  CRED.Active := False;
  CRED.Active := True;
  // Готовим поле ввода:
  SpinEdit1.Value := 0;
  SpinEdit1.SetFocus;
end;
```

Характерной особенностью транзакций в технологии dbExpress.NET является возможность вложения транзакций. Для этого каждая транзакция снабжается уникальным идентификатором и набором дополнительных параметров в экземпляре записи следующего класса:

```
type TTransactionDesc = packed record
     TransactionID    : LongWord;  // Идентификатор транзакции
```

```
    GlobalID          : LongWord;  // Идентификатор транзакции
    IsolationLevel    : TTransIsolationLevel; // Уровень изоляции
    CustomIsolation   : LongWord;  // Умалчиваемый уровень изоляции
    end;
```

Поле TransationID определяет уникальный идентификатор транзакции для всех серверов, кроме Oracle, который для этих целей использует поле GlobalID. Поле IsolationLevel устанавливает уровень изоляции транзакции:

```
type TTransIsolationLevel = (xilREADCOMMITTED, xilREPEATABLEREAD,
                             xilDIRTYREAD, xilCUSTOM);
```

Если это поле имеет значение xilCUSTOM, изоляция транзакции определяется полем CustomIsolation, однако не все серверы поддерживают это значение. При вызове метода StartTransaction в транзакцию передается ее уникальный идентификатор в поле TransactionID (или GlobalID для сервера Oracle). Это позволяет указывать, какую именно транзакцию подтверждает или откатывает программа.

Помимо событий, связанных с установкой и разрывом соединения (AfterConnect, AfterDisconnect, BeforeConnect, BeforeDisconnect), для компонента TSQLConnection определено следующее событие, возникающее перед установкой соединения и до события BeforeConnect, если свойство LoginPrompt имеет значение True:

```
type TSQLConnectionLoginEvent = procedure(
     Database: TSQLConnection; LoginParams: TStrings) of object;
property OnLogin: TSQLConnectionLoginEvent;
```

В обработчике программа может указать параметры User_Name, Password и Database.

Компонент TSQLDataSet

Компонент TSQLDataSet определяет набор данных с однонаправленным курсором или команду обновления данных, в зависимости от содержимого его свойства CommandText.

Компонент унаследовал от класса TDataSet многие свойства, методы и события, *типичные для наборов данных BDE.NET* (см. главу 3). В этом разделе рассматриваются только специфические для компонента свойства, методы и события.

Свойства компонента TSQLDataSet перечислены в табл. 5.3.

Таблица 5.3. Свойства компонента TSQLDataSet

Свойство	Описание
`property CommandText: String;`	Содержит текст SQL-запроса к серверу БД
`type TSQLCommandType = (ctQuery, ctTable, ctStoredProc);` `property CommandType: TSQLCommandType;`	Определяет тип запроса: ctQuery — обычный SQL-запрос; ctTable — запрос, возвращающий все записи из единственной таблицы; ctStoredProc — обращение к хранимой процедуре

Свойство	Описание
`property DataSource: TDataSource;`	Сдержит ссылку на главный НД в отношении главный–детальный
`property DesignerData: String;`	Не имеет определенного назначения. Может использоваться в специализированных наследниках
`property IndexDefs: TIndexDefs;`	Содержит описания всех индексов, определенных для данного НД
`property MaxBlobSize: Integer;`	Определяет максимальный размер буфера (в байтах), в который будут считываться данные из BLOB-столбцов
`property NoMetadata: Boolean;`	Если содержит `True`, компонент не получает от сервера связанных с НД метаданных (индексов), что может повысить производительность работы. Однако для отношений главный–детальный, для редактирования записей и в некоторых других случаях это свойство должно иметь значение `False`, заданное по умолчанию
`property ParamCheck: Boolean;`	Если содержит `False`, компонент не будет автоматически обновлять свойство `Params` при изменении содержимого свойства `CommandText`
`property Params: TParams;`	Содержит параметры запроса
`property Prepared: Boolean;`	Если содержит `True`, компонент требует от сервера подготовить выполнение запроса. Выделенные для этого ресурсы сервера будут освобождены при установке в свойство значения `False`
`property RecordCount: Integer;`	Содержит количество записей в НД. Чтение этого свойства вызовет исключение, если компонент реализует обращение к хранимой процедуре, выполняет параметрический запрос или объединяет данные из нескольких таблиц
`property SortFieldNames: String;`	Содержит перечень столбцов сортировки, если свойство `CommandType` имеет значение `ctTable`
`property SQLConnection: TSQLConnection;`	Содержит ссылку на связной объект `TSQLConnection`
`property TransactionLevel: SmallInt;`	Указывает уровень транзакции, в которой будет выполняться запрос. Только InterBase разрешает вложенные транзакции. Серверы Oracle и DB2 не допускают вложение транзакций, а сервер MySQL их не поддерживает

В табл. 5.4 перечислены лишь наиболее важные методы компонента TSQLDataSet.

Таблица 5.4. Методы компонента TSQLDataSet

Метод	Описание
`type TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);`	Создает поток данных для доступа к BLOB-столбцам
`function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream; override;`	
`function ExecSQL(ExecDirect: Boolean = False): Integer; override;`	Выполняет запрос, который не возвращает НД. Параметр ExecDirect определяет, будет ли запрос подготавливаться сервером перед его исполнением
`type TBlobByteData = array of Byte;`	Читает в буфер Buffer данные из BLOB-столбца текущей записи
`function GetBlobFieldData (FieldNo: Integer; var Buffer: TBlobByteData): Integer; override;`	
`function ParamByName(const Value: String): TParam;`	Открывает доступ к параметру по его имени
`type TSchemaType = (stNoSchema, stTables, stSysTables, stProcedures, stColumns, stProcedureParams, stIndexes );`	Управляет схемой отображаемых данных (см. далее)
`procedure SetSchemaInfo( SchemaType: TSchemaType; SchemaObjectName, SchemaPattern: String );`	

Элементы типа TSchemaType:

- stNoSchema — данные отображаются по запросу в свойстве CommandText;
- stTables — отображается информация о таблицах БД;
- stSysTables — отображается информация о системных таблицах БД;
- stProcedures — отображается информация о хранимых процедурах;
- stColumns — отображается информация о столбцах таблицы, имя которой содержит параметр SchemaObjectName;
- stProcedureParams — отображается информация о параметрах хранимой процедуры, имя которой содержит параметр SchemaObjectName;
- stIndexes — отображается информация об индексах таблицы БД, имя которой содержит параметр SchemaObjectName.

Параметр SchemaPattern метода SetSchemaInfo определяет SQL-маску отображаемой информации. Отображаемая запись должна соответствовать маске или она не будет отображаться. В маске используются два специальных символа %

(на месте этого символа может находиться произвольная строка) и _ (на месте этого символа может находиться любой одиночный символ).

Компонент TSQLDataSet не имеет специфических событий по сравнению с событиями, рассмотренными в главе 3.

Компонент TSQLQuery

Класс TSQLQuery является непосредственным потомком класса TSQLDataSet и наследует от него большинство свойств и методов. «Своими» у компонента являются свойства SQL и Text. Первое определяет текст многострочного запроса, второе — однострочного. Метод ExecSQL выполняет запрос, не возвращающий записи, а метод PrepareStatment готовит запрос к выполнению.

Подобно своему родителю, компонент возвращает набор данных с однонаправленным курсором.

Компонент TSQLTable

Компонент TSQLTable используется для доступа к одной таблице БД. Он имеет свойства MasterSource и MasterFields, с помощью которых можно установить реляционную связь главный—детальный. Свойства IndexFields и IndexNames позволяют подключать к НД нужный индекс и тем самым сортировать данные. Свойство TableName определяет имя отображаемой таблицы БД.

Компонент TSQLMonitor

Компонент TSQLMonitor используется для отладки программы: он перехватывает все сообщения, которыми обмениваются соединение и сервер, и помещает их в текстовый список или сохраняет в файле.

Примеры перехваченных сообщений:

```
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT * FROM TypeNakl
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME,
B.RDB$FIELD_NAME, B.RDB$FIELD_POSITION, '', 0, A.RDB$INDEX_TYPE, '',
A.RDB$UNIQUE_FLAG, C.RDB$CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE
FROM RDB$INDICES A, RDB$INDEX_SEGMENTS B FULL OUTER JOIN
RDB$RELATION_CONSTRAINTS C ON A.RDB$RELATION_NAME =
C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY'
WHERE (A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG IS NULL) AND
(A.RDB$INDEX_NAME = B.RDB$INDEX_NAME) AND (A.RDB$RELATION_NAME =
UPPER('TypeNakl')) ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
```

```
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_allocate_statement
select count(*) from TypeNakl
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
```

Свойства компонента TSQLMonitor перечислены в табл. 5.5.

Таблица 5.5. Свойства компонента TSQLMonitor

Свойство	Описание
`property Active: Boolean;`	Включает/выключает работу компонента
`property AutoSave: Boolean;`	Если содержит True, все перехваченные сообщения помещаются в файл, указанный свойством FileName
`property FileName: String;`	Содержит имя файла, возможно, с путем доступа. В этом файле сохраняются все перехваченные компонентом сообщения, если свойство AutoSave содержит True. Если AutoSave = False, файл является целевым файлом, заданным по умолчанию для методов LoadFromFile и SaveToFile
`property MaxTraceCount: Integer;`	Содержит максимальное количество сохраненных сообщений. Если содержит –1, ограничений нет, если 0, компонент не перехватывает сообщения. Переустановка значения этого свойства в работающей программе влияет только на новые сообщения и не влияет на уже сохраненные
`property SQLConnection: TSQLConnection;`	Содержит ссылку на связной компонент
`property TraceCount: Integer;`	Указывает количество сохраненных сообщений
`property TraceList: TStrings;`	Хранилище сообщений

Компонент TSQLMonitor имеет два специфических метода, с помощью которых программа может загрузить из файла или сохранить в файле протокол сообщений:

```
procedure LoadFromFile(AFileName: String);
procedure SaveToFile(AFileName: String);
```

Если при обращении к любому из этих методов указана пустая строка AFileName, используется имя файла из свойства FileName.

Для компонента TSQLMonitor определены два события. Первое событие:

```
type
  SQLTRACEDesc = packed record
  pszTrace     : array [0..1023] of Char;
  eTraceCat    : TRACECat;
  uTotalMsgLen : Word;
  end;
TTraceLogEvent = procedure (Sender: TObject;
                            var CBInfo: SQLTRACEDesc) of object;
property OnLogTrace: TTraceLogEvent;
```

Второе событие:

```
type TTraceEvent = procedure(Sender: TObject;
                             var CBInfo: SQLTRACEDesc;
                             var LogTrace: Boolean) of object;
property OnTrace: TTraceEvent;
```

Первое событие возникает сразу после того, как очередное сообщение помещено в список TraceList, но до сохранения его в файле, если AutoSave = True. Второе событие возникает непосредственно перед сохранением сообщения в списке или файле.

Параметр CBInfo определяет запись типа SQLTRACEDesc со следующими полями:

- pszTrace — текст сообщения;
- eTraceCat — категория сообщения;
- uTotalMsgLen — длина сообщения в буфере pszTrace.

Если в параметр LogTrace обработчика OnTrace поместить значение False, сообщение не будет сохранено в списке (файле).

Компонент TSimpleDataSet

Компонент TSimpleDataSet буферизует записи и создает двунаправленный курсор. Он имеет две особенности. Во-первых, с помощью свойства ConnectionName он может создать собственное соединение с сервером БД. Во-вторых, только с его помощью в технологии dbExpress можно кэшировать изменения.

Следует заметить, что создавать с помощью компонента TSimpleDataSet собственное соединение с сервером следует лишь в исключительных случаях, так как это соединение нельзя разделить с другими компонентами, а количество одновременно создаваемых соединений с сервером обычно лимитируется условиями лицензионного соглашения (например, поставляемый с Delphi 2005 сервер InterBase 7.5 ограничивает количество соединений пятью).

Наиболее важные свойства компонента представлены в табл. 5.6.

Таблица 5.6. Свойства компонента TSimpleDataSet

Свойство	Описание
`property AutoCalcFields: Boolean;`	Если содержит `True`, будет возбуждаться событие `OnCalcFields`
`property Connection: TSQLConnection;`	Содержит ссылку на связной компонент или описывает собственное соединение с БД
`property DataSet: TDataSet;`	Содержит ссылку на внешний НД или описывает собственный SQL-запрос
`property DisableStringTrim: Boolean;`	Если содержит `True`, будут отсекаться ведомые пробелы при подтверждении изменений в столбцах
`property FetchOnDemand: Boolean;`	Если содержит `True` (по умолчанию), данные кэшируются по мере надобности (например, при прокрутке сетки отображения)
`property FileName: string;`	Содержит имя файла, в котором сохраняются данные (см. далее)
`property MasterFields: string;`	Содержит ссылку на столбцы, реализующие реляционную связь главный–детальный
`property MasterSource: TDataSource;`	Содержит ссылку на главный НД в связи главный–детальный
`property PacketRecords: Integer;`	Указывает количество записей, получаемых компонентом из БД в одном пакете. Если содержит 0, компонент получает из БД метаданные
`property Params: TParams;`	Аналогично одноименному свойству компонента `TQuery` (см. раздел «Запросы» в главе 3)

Свойство `FileName` компонента позволяет сохранять данные из буфера в дисковом файле и читать из файла в буфер. Таким способом реализуется *отложенная обработка записей*. Пользователь с помощью ноутбука подключается к локальной сети, переписывает интересующие его данные в файл и затем отправляется на удаленную презентацию, конференцию, деловую встречу или распродажу. Там он оперирует файловыми данными, вносит в них необходимые изменения. После возвращения на предприятие он вновь подключается к сети и подтверждает сделанные им изменения в БД. Если свойство содержит правильный маршрут доступа к файлу, компонент автоматически записывает в него данные при закрытии НД и так же автоматически читает из файла при его открытии.

Чтобы проиллюстрировать технику отложенной обработки, внесем в пример, рассмотренный в начале этой главы (см. раздел «Пример простой программы»), следующие изменения:

1. Поместите на нижнюю панель две кнопки и диалоговые окна `TOpenDialog` и `TSaveDialog`. Назовите кнопки (свойства `Caption`) **В файл** и **Из файла**.

2. Напишите такие обработчики событий OnClick кнопок:

```
procedure TForm1.Button1Click(Sender: TObject);
// Сохранение данных в файле
begin
  if SaveDialog1.Execute then
  begin
    Nakls.FileName := SaveDialog1.FileName;
    Screen.Cursor := crHourGlass;
    DataSource1.Enabled := False;
    Nakls.Close;
    Nakls.Open;
    DataSource1.Enabled := True;
    Screen.Cursor := crDefault
  end
end;
procedure TForm1.Button2Click(Sender: TObject);
// Чтение данных из файла
begin
  if OpenDialog1.Execute then
  begin
    DataSource1.Enabled := False;
    Screen.Cursor := crHourGlass;
    Nakls.Close;
    Nakls.FileName := OpenDialog1.FileName;
    Nakls.Open;
    DataSource1.Enabled := True;
    Screen.Cursor := crDefault
  end
end;
```

Замечу, что для связи главный—детальный нужно создавать два файла. Если бы в нашем примере мы попытались сохранить детальный НД, были бы сохранены данные только по одной накладной. Чтобы данные по всем накладным были сохранены в файле, нужно предварительно разорвать связь главный—детальный, затем сохранить данные в файле, после чего восстановить реляционную связь.

Наиболее важные методы компонента TSimpleDataSet перечислены в табл. 5.7.

Таблица 5.7. Методы компонента TSimpleDataSet

Метод	Описание
`procedure AddIndex(const Name, Fields: String; Options: TIndexOptions; const DescFields: String = ''; const CaseInsFields: String = ''; const GroupingLevel: Integer = 0 );`	Создает новый индекс (см. далее)
`procedure AppendData(const Data: OleVariant; HitEOF: Boolean);`	Добавляет данные Data к существующему локальному НД. Вместо этого метода обычно используется GetNextPacket

продолжение

Таблица 5.7 (продолжение)

Метод	Описание
`procedure ApplyRange;`	Включает механизм учета диапазона фильтрации записей
`function ApplyUpdates (MaxErrors: Integer); Integer;`	Обновляет записи в БД: `MaxErrors` — максимально допустимое количество ошибок. Возвращает истинное количество ошибок
`function BookmarkValid (Bookmark: TBookmark): Boolean;`	Возвращает `True`, если закладка `Bookmark` содержит правильную ссылку
`procedure Cancel; override;`	Откатывает все неподтвержденные изменения текущей записи
`procedure CancelRange;`	Отменяет режим фильтрации по ключевым столбцам
`procedure CancelUpdates;`	Отменяет все неподтвержденные изменения из пакета `Delta`
`procedure CloneCursor(Source: TCustomClientDataSet; Reset: Boolean; KeepSettings: Boolean = False);`	Позволяет разделять данные двумя и более компонентами (см. далее)
`function CompareBookmarks (Bookmark1, Bookmark2: TBookmark): Integer;`	Сравнивает две закладки и возвращает –1, 0 или +1, если первая закладка, соответственно, меньше, равна или больше второй
`function ConstraintsDisabled: Boolean;`	Разрешает или запрещает компоненту учитывать ограничения на значения столбцов
`function CreateBlobStream (Field: TField; Mode: TBlobStreamMode): TStream; override;`	Создает поток данных для BLOB-столбца: `Field` — имя столбца; `Mode` — режим потока: для чтения (`bmRead`), записи (`bmWrite`) или обновления (`bmReadWrite`) данных
`procedure CreateDataSet;`	Создает пустой набор данных с параметрами по умолчанию
`procedure DeleteIndex(const Name: String);`	Удаляет индекс с именем `Name`
`procedure DisableConstraints;`	Временно прекращает действие ограничений, наложенных на значения столбцов
`procedure EditKey;`	Переводит НД в режим `dsEditKey` и сохраняет поисковый буфер. Этот метод полезно вызывать для комплексного индекса, когда между двумя поисками меняются лишь некоторые столбцы (см. далее примечание)
`procedure EditRangeEnd;`	Переводит НД в режим редактирования нижней границы диапазона фильтрации
`procedure EditRangeStart;`	Переводит НД в режим редактирования верхней границы диапазона фильтрации

Метод	Описание
`procedure EmptyDataSet;`	Очищает буфер от всех записей
`procedure EnableConstraints;`	Отменяет режим, установленный методом `DisableConstraints`
`procedure Execute; virtual;`	Выполняет запрос из свойства `CommandText`
`procedure FetchBlobs;`	Получает от сервера приложений текущий BLOB-столбец
`procedure FetchDetails;`	Получает от сервера приложений вложенную таблицу Oracle
`function FetchParams;`	Получает от сервера приложений текущие параметры компонентов `TQuery` или `TStoredProc`
`function FindKey(const KeyValues: array of const): Boolean;`	Аналог одноименного метода `TTable` (см. далее примечание)
`procedure FindNearest(const KeyValues: array of const);`	Аналог одноименного метода `TTable` (см. далее примечание)
`function GetCurrentRecord (Buffer: IntPtr): Boolean; override;`	Копирует текущую запись в буфер `Buffer`
`function GetFieldData(Field: TField; Buffer: IntPtr): Boolean; override;`	Копирует столбец `Field` текущей записи в буфер `Buffer`

`type TGroupPosInd = (gbFirst, gbMiddle, gbLast);`
`TGroupPosInds = set of TGroupPosInd;`
`function GetGroupState(Level: Integer): TGroupPosInds;`
	Возвращает положение текущей записи в группе агрегирования `Level`: `gbFirst` — первая в группе минимум из двух записей; `gbMiddle` — ни первая, ни последняя; `gbLast` — последняя в группе минимум из двух записей

`procedure GetIndexInfo(IndexName: String);`	Возвращает информацию об индексе с именем `IndexName`
`function GetNextPacket: Integer;`	Требует от сервера приложений передать следующий пакет записей. Длина пакета определяется свойством `PacketRecords`. Функция возвращает количество записей, добавленных в свойство `Data`
`function GetOptionalParam(const ParamName: String): OleVariant;`	Возвращает из пакета данных пользовательскую информацию с именем `ParamName`, которая помещена туда методом `SetOptionalParam`
`procedure GotoCurrent(DataSet: TClientDataset);`	Синхронизирует курсор текущего набора с курсором набора `DataSet`

продолжение

Таблица 5.7 (продолжение)

Метод	Описание
`function GotoKey: Boolean;`	Ищет запись, определенную предварительным вызовом `SetKey` или `EditKey` с последующим вызовом `FieldByName`, и возвращает `True`, если запись найдена (см. далее примечание)
`procedure GotoNearest;`	Ищет запись, либо точно соответствующую поисковому значению, либо следующую за ней в порядке индекса. Поисковое значение задается методом `SetKey` или `EditKey` с последующим вызовом `FieldByName` (см. далее примечание)
`procedure LoadFromFile(const FileName: String = '');`	Загружает набор данных из файла `FileName`. Совместно с `SaveToFile` дает возможность отложенного обновления данных
`procedure LoadFromStream (Stream: TStream);`	Загружает набор данных из потока `Stream`
`type TLocateOption = (loCaseInsensitive, loPartialKey); TLocateOptions = set of TLocateOption;` `function Locate(const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean; override;`	Ищет запись по значениям `KeyValues` столбцов `KeyFields` и возвращает `True` в случае успеха. Параметр `Option` уточняет условия поиска: `loCaseInsensitive` — игнорировать разницу в регистре букв; `loPartialKey` — искать по части ключа
`function Lookup(const KeyFields: String; const KeyValues: Variant; const ResultFields: String): Variant; override;`	Ищет запись по значениям `KeyValues` столбцов `KeyFields` и возвращает значения столбцов `ResultFields` найденной записи в случае успеха
`procedure MergeChangeLog;`	Объединяет данные из протокола изменений с реальными данными вызовом `ApplayUpdates`
`procedure Post; override;`	Подтверждает изменения в текущей записи
`function Reconcile(const Results: OleVariant): Boolean;`	Вызывается методом `ApplayUpdates` провайдера в случае возникновения ошибки `Results` — пакет ошибочных записей. Метод возбуждает событие `OnReconcileError` для каждой записи пакета `Results` (см. далее)
`procedure RefreshRecord;`	Обновляет текущую запись, считывая ее у провайдера. Протокол изменений записи не меняется
`procedure RevertRecord;`	Восстанавливает текущую запись по протоколу ее изменений
`procedure SaveToFile(const FileName: String = '');`	Сохраняет НД в файле `FileName`

Метод	Описание
`procedure SaveToStream (Stream: TStream; Format: TDataPacketFormat = dfBinary);`	Сохраняет данные в потоке
`procedure SetAltRecBuffers (Old, New, Cur: PChar);`	Устанавливает новые значения в столбцы OldValue, NewValue и CurValue текущей записи
`procedure SetKey;`	Переводит НД в режим dsSetKey для установки нового поискового значения, которое задается следующим вызовом FieldByName (см. далее примечание)
`procedure SetOptionalParam(const ParamName: String; const Value: OleVariant; IncludeInDelta: Boolean = False);`	Помещает в пакет данных пользовательскую информацию: ParamName — имя данных; Value — данные; IncludeInDelta содержит True, если данные помещаются в пакет Delta
`procedure SetRange(const StartValues, EndValues: array of const);`	Устанавливает диапазон поиска по ключам (см. далее примечание)
`procedure SetRangeEnd;`	Устанавливает нижнюю границу диапазона поиска (см. далее примечание)
`procedure SetRangeStart;`	Устанавливает верхнюю границу диапазона поиска (см. далее примечание)
`function UndoLastChange (FollowChange: Boolean): Boolean;`	Отменяет последнюю операцию изменения, удаления или вставки записи. Если FollowChange = True, после отмены текущей становится восстановленная запись
`function UpdateStatus: TUpdateStatus; override;`	Возвращает статус обновления текущей записи

ПРИМЕЧАНИЕ

Подробнее об индексном поиске с помощью методов SetKey, EditKey, SetRange и т. п. см. в главе 3.

Параметры метода AddIndex перечислены ниже:

- Name — имя индекса;
- Fields — список индексных столбцов;
- Options — может включать 0, 1 или 2 флага ixDescending (сортировка по убыванию) и ixCaseInsensitve (сортировка с учетом регистра букв); остальные значения типа TIndexOptions вызовут исключение;
- DescFields — список столбцов, сортируемых по убыванию; игнорируется, если установлен флаг ixDescending;
- CaseInsFields — список столбцов, сортируемых с учетом регистра букв; игнорируется, если установлен флаг ixCaseInsensitve;
- GroupingLevel — уровень вложенности индекса, заданный по умолчанию.

С помощью метода CloneCursor компонент переводится в режим совместного владения данными с другим компонентом, ссылку на который содержит параметр Source. Параметры Reset и KeepSettings определяют, как будут использоваться следующие свойства:

```
Filter, Filtered, FilterOptions, OnFilterRecord;
IndexName;
MasterSource, MasterFields;
ReadOnly.
```

Эти свойства влияют на совместное владение данными: если оба параметра имеют значение False, перечисленные свойства у текущего компонента будут такими же, как у компонента Source; если Reset = True, свойства очищаются, а если Reset = False и KeepSettings = True, сохраняют те значения, которые они имели до обращения к методу. Поскольку оба компонента владеют одной и той же копией данных, изменения, сделанные в копии одним компонентом, тут же становятся видны другому компоненту.

Метод Reconcile для каждой ошибочной записи генерирует связанное с компонентом специфическое событие:

```
type
  TReconcileAction = (raSkip, raAbort, raMerge,
                      raCorrect, raCancel, raRefresh);

  TReconcileErrorEvent = procedure(DataSet: TClientDataSet;
                         E: EReconcileError;
                         UpdateKind: TUpdateKind;
                         var Action: TReconcileAction) of object;

  property OnReconcileError: TReconcileErrorEvent;
```

Здесь:

- DataSet — клиентский НД, содержащий ошибочные записи (текущая запись этого набора и является ошибочной);
- E — объект исключения;
- UpdateKind — характер обновления ошибочной записи;
- Action — в этом параметре обработчик должен вернуть одно из следующих значений:
  - raSkip — пропустить ошибочную запись;
  - raAbort — прекратить режим корректировки;
  - raMerge — выполнить конкатенацию ошибочной записи с исходной;
  - raCorrect — запись была скорректирована в обработчике;
  - raCancel — восстановить запись по протоколу изменений;
  - raRefresh — восстановить запись, считав ее из БД.

Обработчик может использовать значения OldValue, NewValue и CurValue для каждого столбца записи, но он не должен изменять текущее положение курсора НД.

События компонента TSimpleDataSet перечислены в табл. 5.8.

Таблица 5.8. События компонента TSimpleDataSet

Событие	Описание
`type TRemoteEvent = procedure (Sender: TObject; var OwnerData: OleVariant) of object;`	Возникает после подтверждения изменений, но перед событием OnReconcileError
`property AfterApplyUpdates: TRemoteEvent;`	
`property AfterExecute: TRemoteEvent;`	Возникает после получения НД в результате выполнения метода Execute
`property AfterGetParams: TRemoteEvent;`	Возникает после получения от провайдера значений параметров в результате выполнения метода FetchParams
`property AfterGetRecords: TRemoteEvent;`	Возникает после получения от провайдера пакета данных
`property AfterRowRequest: TRemoteEvent;`	Возникает после получения компонентом детальной информации о текущей записи
`property BeforeApplyUpdates: TRemoteEvent;`	Возникает перед подтверждением изменений
`property BeforeExecute: TRemoteEvent;`	Возникает перед выполнением метода Execute
`property BeforeGetParams: TRemoteEvent;`	Возникает перед выполнением метода FetchParams
`property BeforeGetRecords: TRemoteEvent;`	Возникает перед получением пакета данных от провайдера
`property BeforeRowRequest: TRemoteEvent;`	Возникает перед получением компонентом детальной информации о текущей записи
`type TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel, raRefresh);`	Это событие генерируется методом Reconcile для каждой ошибочной записи (см. ранее описание метода Reconcile)
`type TReconcileErrorEvent = procedure (DataSet: TCustomClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction) of object;`	
`property OnReconcileError: TReconcileErrorEvent;`	

События Before*XXX*–After*XXX* сопровождаются параметром OwnerData, которым программа может распоряжаться по своему усмотрению.

6 Технология IBX.NET

Технология IBX.NET (IBX — InterBase eXpress) рассчитана на создание «облегченного» клиента. С этой целью она предоставляет программисту средства непосредственного обращения к промышленному серверу InterBase версии 5.5 и выше без использования машины баз данных BDE или других подобных средств доступа к данным. Delphi не запрещает и традиционную архитектуру с BDE, однако, если вы собираетесь создавать клиент-серверную СУБД на основе сервера InterBase 5.5 и выше, вряд ли вы откажетесь от более компактных и быстрых средств IBX.NET, описываемых в этой главе.

ПРИМЕЧАНИЕ

Все компоненты технологии IBX.NET сосредоточены в категории InterBase палитры компонентов. В списке категорий, раскрывающемся после щелчка на кнопке Categories, эта категория обычно не видна, так как она следует за категорией Internet. Для доступа к категории InterBase используйте полосу прокрутки в списке категорий.

Пример простой программы

Перед выполнением этого примера (проект Ch06\Nakls\IBXDemo.dpr) на вашей машине (или в доступном вам сетевом окружении) должен быть развернут и запущен сервер InterBase 5.5 или выше (с Delphi 2005 поставляется InterBase 7.5).

Характерной особенностью использования технологии IBX.NET является необходимость создания соединения с БД, что достигается с помощью двух компонентов: `TIBDatasase` и `TIBTransaction`. Только после размещения на форме этих компонентов и их настройки могут получить доступ к данным другие компоненты IBX.

1. Начните новый VCL-проект и разместите на форме компоненты `TIBDatabase`, `TIBTransaction`, два компонента `TIBTable` (все компоненты располагаются в категории InterBase палитры компонентов) и два компонента `TDataSource` (категория Data Access).

2. Раскройте список свойства DatabaseName компонента IBDatabase1 и с его помощью установите ссылку на файл БД IB_BIBL.GDB (он размещается в папке C:\BIBLDATA).
3. Раскройте редактор свойства Params компонента IBDatabase1 и введите регистрационное имя и пароль:

```
USER_NAME = SYSDBA
PASSWORD = masterkey
```

ПРИМЕЧАНИЕ

При использовании технологии IBX.NET параметр USER_NAME пишется с символом подчеркивания. Если используются другие технологии (BDE.NET, dbGo.NET, dbExpress.NET), вместо подчеркивания можно ставить пробел.

4. Установите в свойство LoginPrompt компонента IBDatabase1 значение False.
5. С помощью списка свойства DefaultDatabase свяжите компонент IBTransaction1 с компонентом IBDatabase1.
6. Поместите в свойство Database компонента IBTable1 ссылку на компонент IBDatabase1. В списке свойства TableName выберите таблицу NAKLS. Установите в свойство Name компонента значение Nakls. Откройте таблицу (поместите True в свойство Active). При этом автоматически активизируются компоненты IBDatabase1 и IBTransaction1.
7. Компонент IBTable2 свяжите с компонентом IBDatabase1 и таблицей MOVES, укажите для него имя Moves.
8. Таблицы NAKLS (накладные) и MOVES (списки книг) связаны отношением главная—детальная по полям NaklID. Для реализации связи в свойство MasterSource компонента MOVES поместите ссылку на источник DataSource1, раскройте редактор свойства MasterFields и укажите связь по полям NaklID (рис. 6.1). Откройте НД MOVES.

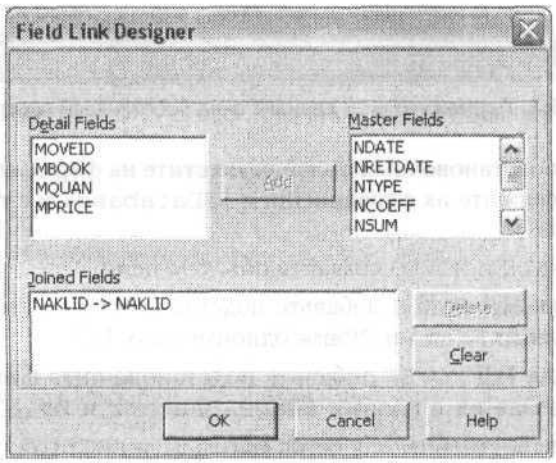


Рис. 6.1. Установление связи по полям NaklID

9. Поместите на форму три панели TPanel (категория Standard). В свойстве Align двух из них укажите значение alBottom, а в третьей — значение alClient.
10. Поместите на верхнюю панель сетку TDBGrid (категория Data Control), в свойстве Align сетки укажите значение alClient и сошлитесь на источник DataSource1 в ее свойстве DataSource. Сетка наполнится записями из главной таблицы.
11. Поместите на среднюю панель еще одну сетку TDBGrid и настройте ее на отображение записей детальной таблицы (Align = alClient, DateSource = DataSource2).
12. Разместите на нижней панели навигатор TDBNavigator (категория Data Control) и свяжите его с верхней сеткой (поместите в свойство DataSource значение DataSource1).

    Вид окна работающей программы для этого момента показан на рис. 6.2.

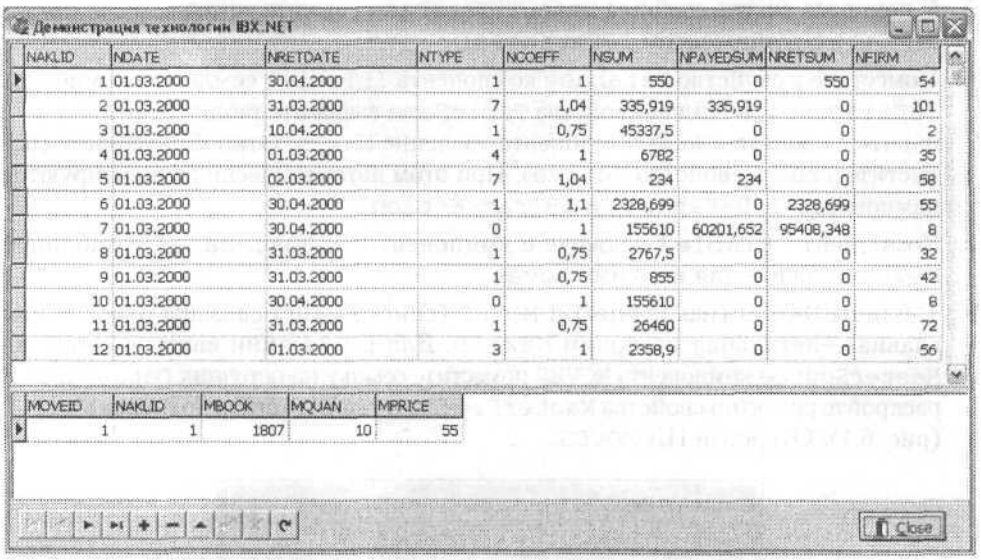


Рис. 6.2. Промежуточный вариант окна работающей программы

13. Для создания подстановочных полей разместите на форме еще три компонента TIBTable, свяжите их с соединением IBDatabase1 и с таблицами FIRMS, BOOKS и TYPES.
14. Для таблиц NAKLS и MOVES создайте объекты-поля.
15. В объекты-поля НД NAKLS добавьте подстановочные поля Firms и Types, связав их с полями FName и TName одноименных НД.
16. В объекты-поля НД MOVES добавьте подстановочные поля Name, Author и Publish, связав их с полями BName, BAuthor и BPublish НД BOOKS.
17. Создайте объекты-столбцы для сеток DBGrid1 и DBGrid2, указав в них подстановочные поля вместо ключевых.

18. Напишите обработчики событий `OnGetText` для всех вещественных полей, чтобы данные были отформатированы нужным образом. Например:

```
procedure TForm1.NAKLSNCOEFFGetText(Sender: TField;
                                    var Text: string;
                                    DisplayText: Boolean);
begin
  Text := FloatToStrF(NAKLSNCoeff.Value, ffNumber, 10, 2);
end;
```

Вид окна работающей программы в окончательном варианте показан на рис. 6.3.

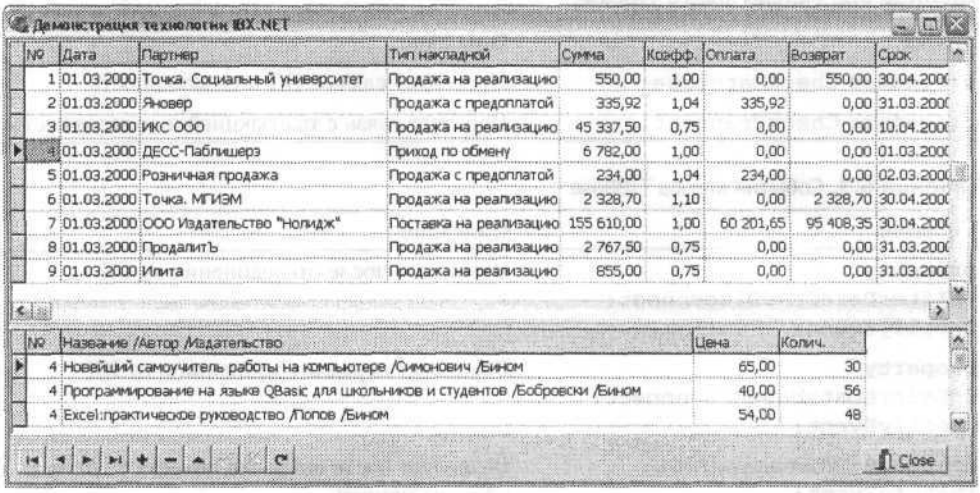


Рис. 6.3. Окончательный вариант окна работающей программы

Компоненты для реализации технологии

Как уже говорилось, особенностью технологии является обязательное использование в ней компонентов `TIBDatabase` и `TIBTransaction`. Эти компоненты имеют общий родительский класс `TIBBase`, с рассмотрения свойств которого мы и начинаем.

Класс TIBBase

Класс `TIBBase` инкапсулирует свойства (табл. 6.1), методы (табл. 6.2) и события (табл. 6.3), общие для компонентов `TIBDatabase` и `TIBTransaction`.

Таблица 6.1. Свойства класса TIBBase

Свойство	Описание
`property Database: TIBDatabase;`	Содержит ссылку на связанный компонент `TIBDatabase`

продолжение

Таблица 6.1 (продолжение)

Свойство	Описание
`property DBHandle: IntPtr;`	Содержит дескриптор базы данных
`property Owner: TObject;`	Содержит ссылку на владельца объекта
`property Transaction: TIBTransaction;`	Содержит ссылку на связанный компонент TIBTransaction
`property TRHandle: IntPtr;`	Содержит дескриптор транзакции

Таблица 6.2. Методы класса TIBBase

Метод	Описание
`procedure CheckDatabase;`	Проверяет связь с БД и ее активность
`procedure CheckTransaction;`	Поверяет связь с транзакцией и ее активность

Таблица 6.3. События класса TIBBase

Событие	Описание
`property OnAfterDatabaseDisconnect: TNotifyEvent;`	Возникает после отсоединения от БД
`property OnBeforeDatabaseDisconnect: TNotifyEvent;`	Возникает перед отсоединением от БД
`property OnDatabaseFree: TNotifyEvent;`	Возникает после выгрузки объекта TIBDatabase из памяти
`property OnAfterTransactionEnd: TNotifyEvent;`	Возникает после окончания транзакции
`Property OnBeforeTransactionEnd: TNotifyEvent;`	Возникает перед окончанием транзакции
`property OnTransactionFree: TNotifyEvent;`	Возникает после выгрузки объекта TIBTransaction из памяти

Компонент TIBDatabase

Компонент TIBDatabase создает соединение с БД InterBase. Свойства компонента TIBDatabase перечислены в табл. 6.4.

Таблица 6.4. Свойства компонента TIBDatabase

Свойство	Описание
`property DatabaseName: String;`	Указывает имя физической БД, с которой связан компонент (см. далее)
`property DBParamByDPB: [const Idx: Integer]: String;`	Позволяет проверить или установить заново какой-либо параметр БД

Свойство	Описание
`property DBSQLDialect: Integer;`	Содержит диалект языка SQL, который используется БД
`property DefaultTransaction: TIBTransaction;`	Содержит ссылку на транзакцию, заданную по умолчанию
`property Handle: IntPtr;`	Содержит дескриптор соединения. Используется для прямого обращения к API-функциям InterBase
`property HandleIsShared: Boolean;`	Содержит `True`, если используется разделяемый дескриптор соединения
`property IdleTimer: Integer;`	Указывает время в миллисекундах, по истечении которого автоматически разрывается соединение с БД, если за это время к ней не было обращения
`property IsReadOnly: Boolean;`	Содержит `True`, если БД работает только в режиме чтения
`property Params: TStrings;`	Содержит список параметров, которые передаются серверу в момент соединения
`property SQLDialect: Integer;`	Содержит диалект языка SQL, который используется компонентом
`property SQLObjectCount: Integer;`	Содержит количество SQL-объектов, связанных с данным соединением. К SQL-объектам относятся IBSQL и BLOB наборов данных InterBase
`property SQLObjects[Index: Integer]: TIBBase;`	Открывает индексированный доступ к SQL-объектам
`property TraceFlags: TTraceFlags;`	Содержит флаги, определяющие различные состояния БД, которые должны прослеживаться в SQL-мониторе (см. далее)
`property TransactionCount: Integer;`	Указывает количество транзакций, связанных в данный момент с компонентом
`property Transactions [Index: Integer]: TIBTransaction;`	Открывает индексированный доступ к связанным с компонентом транзакциям

Свойство `DatabaseName` для локального сервера содержит имя файла БД. При связи с удаленной БД формат имени следующий:

- по протоколу TCP/IP:

```
имя_сервера:имя_файла_БД
```

- по протоколу NetBIOS:

```
\\имя_сервера\имя_файла_БД
```

- по протоколу SPX:

```
имя_сервера@имя_файла_БД
```

Компонент может управлять несколькими транзакциями одновременно. Свойство DefaultTransaction ссылается на транзакцию, которой компонент управляет в данный момент.

Определение типа для свойства TraceFlags:

```
type TTraceFlag = (tfQPrepare, tfQExecute, tfQFetch,
                   tfError, tfStmt, tfConnect, tfTransact,
                   tfBlob, tfService, tfMisc);
TTraceFlags = set of TTraceFlag;
```

Здесь перечислены следующие флаги:

- tfQPrepare — готовится выполнение запроса;
- tfQExecute — запрос выполняется;
- tfQFetch — идет наполнение набора данных;
- tfError — возникла ошибка при выполнении запроса;
- tfStmt — мониторинг всех SQL-предложений;
- tfConnect — устанавливается связь с БД;
- tfTransact — мониторинг начала, подтверждения и отката транзакций;
- tfBlob — мониторинг операций над данными типа BLOB;
- tfService — мониторинг служебных операций;
- tfMisc — мониторинг любых других предложений, не перекрывающихся этими флагами.

Методы компонента TIBDatabase перечислены в табл. 6.5.

Таблица 6.5. Методы компонента TIBDatabase

Метод	Описание
`function AddTransaction(TR: TIBTransaction): Integer;`	Устанавливает связь между компонентом и новой транзакцией
`procedure ApplyUpdates(const DataSets: array of TIBCustomDataSet);`	Подтверждает кэшированные изменения
`procedure CheckActive;`	Проверяет активность компонента и возвращает ошибку, если связь не установлена
`procedure CheckDatabaseName;`	Возбуждает ошибку, если свойство DatabaseName не заполнено
`procedure CheckInactive;`	Возбуждает ошибку, если связь с БД установлена
`procedure Close;`	Разрывает соединение с БД и закрывает все связанные с компонентом НД. Повторное открытие компонента не открывает заново НД
`procedure CloseDataSets;`	Закрывает все связанные с компонентом наборы данных, но в отличие от метода Close, не разрывает связь с сервером и БД

Метод	Описание
`procedure CreateDatabase;`	Создает новую БД, используя свойство `Params` как перечень параметров новой БД
`procedure DropDatabase;`	Удаляет существующую БД
`function FindTransaction (TR: TIBTransaction): Integer;`	Возвращает индекс указанной транзакции
`procedure ForceClose;`	Закрывает БД, даже если в результате операции возникла ошибка
`procedure GetFieldNames(const TableName: String; List: TStrings);`	Возвращает список имен всех столбцов указанной таблицы. К моменту обращения к методу связь с БД должна быть установлена
`procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);`	Возвращает список всех таблиц БД. Если параметр `SystemTables` имеет значение `True`, список дополняется именами системных таблиц
`function IndexOfDBConst(st: String): Integer;`	Возвращает индекс параметра по его имени (если возвращается –1, параметр с нужным именем не найден)
`procedure Open;`	Устанавливает связь с БД
`procedure RemoveTransaction (Idx: Integer);`	Отсоединяет указанную транзакцию от компонента
`procedure RemoveTransactions;`	Отсоединяет все транзакции от компонента
`procedure SetHandle;`	Устанавливает дескриптор для БД
`procedure TestConnected: Boolean;`	Проверяет связь с БД и возвращает `True`, если связь установлена

С помощью метода `CreateDatabase` можно создать новую базу данных. Для этого в свойство `DatabaseName` помещается информация о месторасположении (компьютер, маршрут доступа) и имя файла создаваемой БД, а в набор `Params` — параметры БД, например, такие:

```
USER "SYSDBA"
PASSWORD "masterkey"
PAGE_SIZE=4096
DEFAULT CHARACTER SET WIN1251
```

После выполнения метода будет создана полноценная БД с файлом по адресу, *указанному в свойстве* `DatabaseName`.

События компонента `TIBDatabase` перечислены в табл. 6.6.

Таблица 6.6. События компонента TIBDatabase

Событие	Описание
`property OnDialectDowngradeWarning: TNotifyEvent;`	Возникает в момент открытия БД, если номер используемой в ней версии диалекта SQL меньше указанного в свойстве `SQLDialect`

продолжение

Таблица 6.6 (продолжение)

Событие	Описание
`property OnIdleTimer: TNotifyEvent;`	Возникает, если время «простоя» БД превысит указанное в свойстве `IdleTimer`
`TDatabaseLoginEvent = procedure (Database: TIBDatabase; LoginParams: TStrings) of object;`	Возникает, если свойство `LoginPrompt` содержит `True`. В обработчике события регистрационное имя и пароль необходимо поместить во временный список `LoginParams`
`property OnLogin: TDatabaseLoginEvent;`	

Компонент TIBTransaction

Для установления связи с InterBase в технологии IBX.NET хотя бы один компонент `TIBTransaction` должен быть связан с компонентом `TIBDatabase`. В многозвенных приложениях каждый SQL-запрос исполняется в отдельной транзакции.

Свойства компонента `TIBTransaction` перечислены в табл. 6.7.

Таблица 6.7. Свойства компонента TIBTransaction

Свойство	Описание
`property Active: Boolean;`	Содержит `True`, если компонент активен
`property DatabaseCount: Integer;`	Содержит количество БД, обслуживаемых в данной транзакции
`property Databases[Index: Integer]: TIBDatabase;`	Дает индексный доступ к БД, обслуживаемым транзакцией
`type TTransactionAction = (taRollback, taCommit, taRollbackRetaining, taCommitRetaining);`	Указывает способ завершения транзакции по истечении времени, заданного в свойстве `IdleTimer`: `taRollback` — откатить транзакцию; `taCommit` — подтвердить транзакцию; `taRollbackRetaining` — откатить транзакцию, но сохранить ее контекст (для InterBase версии 6 и выше); `taCommitRetaining` — подтвердить транзакцию, но сохранить ее контекст
`property DefaultAction: TTransactionAction;`	
`property DefaultDatabase: TIBDatabase;`	Содержит текущую БД
`property Handle: IntPtr;`	Содержит дескриптор транзакции
`property HandleIsShared: Boolean;`	Содержит `True`, если дескриптор транзакции разделяемый
`property IdleTimer: Integer;`	Указывает, сколько времени в миллисекундах должно пройти перед автоматическим выполнением свойства `DefaultAction`

Свойство	Описание
`property InTransaction: Boolean;`	Указывает, находится ли БД в состоянии незавершенной транзакции
`property Params: TStrings;`	Содержит набор параметров для компонента
`property SQLObjectCount: Integer;`	Содержит количество обслуживаемых транзакцией активных SQL-компонентов (наборов данных, IBSQL, BLOB)
`property SQLObjects[Index: Integer]: TIBBase;`	Открывает индексный доступ к обслуживаемым транзакцией SQL-компонентам
`property TPB: IntPtr;`	Содержит ссылку на буфер параметров транзакции (только в режиме чтения). Для записи параметров служит свойство `Params`
`Property TPBLength: Short;`	Указывает длину буфера параметров компонента

Методы компонента `TIBTransaction` перечислены в табл. 6.8.

Таблица 6.8. Методы компонента TIBTransaction

Метод	Описание
`function AddDatabase(db: TIBDatabase): Integer;`	Добавляет к компоненту новый компонент `TIBDatabase`
`procedure CheckDatabasesInList;`	Проверяет список связанных компонентов `TIBDatabase`. Если список пуст, возбуждается исключение
`procedure CheckInTransaction;`	Проверяет активность транзакции и список обслуживаемых ею компонентов `TIBDatabase`. Если транзакция неактивна или список пуст, возбуждается исключение
`procedure CheckNotInTransaction;`	Проверяет активность транзакции и список обслуживаемых ею компонентов `TIBDatabase`. Если транзакция активна или список не пуст, возбуждается исключение
`procedure Commit;`	Подтверждает текущую транзакцию и завершает ее
`procedure CommitRetaining;`	Аналогичен методу `Commit`, но сохраняет контекст транзакции
`function FindDatabase (db: TIBDatabase): Integer;`	Возвращает индекс связанного компонента `TIBDatabase`
`procedure RemoveDatabase(Idx: Integer);`	Удаляет компонент `TIBDatabase` с индексом `Idx` из списка связанных с компонентом `TIBTransaction`

продолжение

Таблица 6.8 (продолжение)

Метод	Описание
`procedure RemoveDatabases;`	Удаляет все компоненты `TIBDatabase` из списка связанных с компонентом `TIBTransaction`
`procedure Rollback;`	Осуществляет откат текущей транзакции
`procedure RollbackRetaining;`	Аналогичен методу `Rollback`, но сохраняет контекст транзакции
`Procedure StartTransaction;`	Начинает очередную транзакцию

В момент открытия компонента `TIBTransation` автоматически стартует связанная с ним транзакция. Пример неправильного кода:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Nakls.Open;
  IbTransaction1.StartTransaction; // Ошибка!
  // Транзакция уже стартовала!
  ...
end;
```

В момент открытия НД `Nakls` автоматически открывается и компонент `IBTransaction1`, поэтому во втором операторе делается попытка открыть вложенную транзакцию, что недопустимо. В листинге 6.1 (проект Ch06\TransDemo\TransDemo.dpr) приводится правильное решение классической задачи перевода денег с дебиторского счета на кредиторский (рис. 6.4).

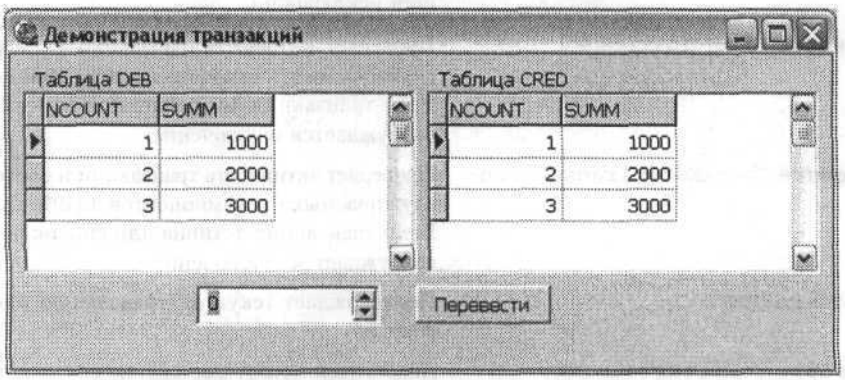


Рис. 6.4. Окно программы с примером использования транзакции

Листинг 6.1. Программа перевода денег с дебиторского счета на кредиторский

```
procedure TForm1.FormActivate(Sender: TObject);
// Передает фокус ввода полю SpinEdit1 в момент активизации формы
begin
  SpinEdit1.SetFocus;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
// Осуществляет перевод суммы под защитой транзакции
var
  Com: TIBQuery;
  ComStr: String;
  Sum: Integer;
begin
  // Проверяем сумму перевода:
  if SpinEdit1.Value <= 0 then
    Exit;                    // Нет ввода
  if SpinEdit1.Value > DEBSumm.Value then
  begin                      // Слишком большая сумма
    SpinEdit1.Value := Round(DEBSumm.Value);
    SpinEdit1.SetFocus;
    Exit;
  end;
  Sum := SpinEdit1.Value;    // Сумма перевода
  // Создаем и настраиваем компонент класса TIBQuery:
  Com := TIBQuery.Create(Self);
  Com.Database := IBDatabase1;
  Com.Transaction := IBTransaction1;
  // Формируем команду снятия денег с дебиторского счета:
  ComStr := Format('UPDATE DEB SET SUMM=%d WHERE NCOUNT=%d',
                   [Round(DEBSumm.Value - Sum), DEBNCount.Value]);
  Com.Text := ComStr;
  // Формируем команду начисления на кредиторский счет:
  ComStr := Format('UPDATE CRED SET SUMM=%d WHERE NCOUNT=%d',
                   [Round(CREDSumm.Value + Sum), CREDNCount.Value]);
  try
    Com.ExecSQL;   // Снимаем деньги
    Com.Text := ComStr;
    Com.ExecSQL;   // Начисляем деньги
    IBTransaction1.Commit;   // Подтверждаем транзакцию
  except                     // Ошибка!
    IBTransaction1.Rollback; // Откатываем транзакцию
    ShowMessage('No!');
  end;
  // Обновляем данные в сетках:
  DEB.Active := False;
  DEB.Active := True;   // Одновременно стартует новая транзакция
  CRED.Active := False;
  CRED.Active := True;
  // Готовим поле для нового ввода:
  SpinEdit1.Value := 0;
  SpinEdit1.SetFocus;
end;
```

Обратите внимание: при открытии любого НД автоматически стартует новая транзакция, поэтому в приведенном примере нет команды

```
IBTahsaction1.StartTransaction;
```

С компонентом связано единственное событие, которое возникает после завершения транзакции по истечении времени, указанного в свойстве IdleTimer:

```
property OnIdleTimer: TNotifyEvent;
```

Компонент TIBTable

Компонент TIBTable повторяет в функциональном отношении компонент TTable технологии BDE.NET, поэтому их основные свойства, методы, события, а также приемы работы с ними аналогичны (см. главу 3). В табл. 6.9 описываются лишь специфические для компонента свойства.

Таблица 6.9. Свойства компонента TIBTable

Свойство	Описание
`property BufferChunks: Integer;`	Указывает, на сколько записей будет увеличиваться буфер таблицы по мере его наполнения
`property UniDirectional: Boolean;`	Если содержит False, с таблицей связывается двунаправленный курсор и навигация по ней разрешается как сверху вниз, так и снизу вверх. Если содержит True, навигация по таблице разрешается только сверху вниз
`property UpdateObject;`	Ссылается на объект, который будет реализовывать изменения в таблицах, открытых только для чтения

В отличие от своего аналога из технологии BDE.NET, компонент TIBTable может создавать однонаправленный курсор (компонент TTable всегда создает двунаправленный курсор). Однонаправленные курсоры экономят ресурсы компьютера и реализуют более быструю навигацию по записям. Свойство UniDirectional объявлено в секции **publish**, а не в секции **published**, поэтому его нет в окне инспектора объектов, и программист не может изменить его на этапе разработки программы (но может — при прогоне программы). Если таблица с однонаправленным курсором отображается в сетке TDBGrid, перемещение курсора вверх возможно только в пределах сетки. Перед изменением свойства UniDirectional компонент должен быть закрыт.

В документации сказано, что компонент может отображать не только реальные таблицы БД, но и представления (*view*). Это действительно так, только при ссылке на таблицу не стоит искать представление в раскрывающемся списке свойства TableName, а следует просто вручную записать имя представления в это свойство. Поскольку представления могут быть необновляемыми (например, они могут содержать записи из нескольких таблиц), в компоненте появилось свойство UpdateObject. В это свойство чаще всего помещается имя компонента TIBUpdateSQL, с помощью которого реализуются изменения отображаемых данных, если представление «не живое».

Компонент не имеет специфических по сравнению с компонентом TTable методов и событий.

Компонент TIBQuery

Компонент TIBQuery является функциональным аналогом компонента TQuery, описанного в главе 3. В этом разделе представлены только специфические для компонента свойства (табл. 6.10).

Таблица 6.10. Свойства компонента TIBQuery

Свойство	Описание
`property GenerateParamNames: Boolean;`	Если содержит True, компонент создает список параметров запроса (в компоненте TQuery этот список создается всегда)
`property StmtHandle: IntPtr;`	Дескриптор БД, который можно использовать при непосредственном обращении к API-функциям InterBase

По-моему, свойству GenerateParamNames трудно найти практическое применение, разве что оно дает программисту возможность немного сэкономить ресурсы компьютера. Кроме того, мне не удалось найти разницу в поведении компонента при изменении значения этого свойства. Например, в следующем фрагменте список ListBox1 всегда наполнялся именами параметров независимо от значения свойства GenerateParamNames:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  k: Integer;
begin
  IBQuery1.GenerateParamNames := False;
  for k := 0 to IBQuery1.Params.Count-1 do
    ListBox1.Items.Add(IBQuery1.Params[k].Name);
end;
```

Компонент не имеет специфических методов и свойств.

Компонент TIBDataSet

Компонент TIBDataSet предназначен для просмотра и модификации набора записей, указанного в его свойстве SelectSQL. Особенностью компонента является то обстоятельство, что НД не становится обновляемым до тех пор, пока не определены модифицирующие его SQL-операторы в его строковых свойствах InsertSQL, DeleteSQL и ModifySQL. Если, например, в его свойстве InsertSQL записан правильный оператор **INSERT**, НД будет способен вставлять записи, но только так, как определено в этом операторе. Точно так же, если в свойстве DeleteSQL определен оператор **DELETE**, НД будет способен удалить запись или записи, определенные в этом операторе, и т. д. Действия пользователя являются своеобразным

«спусковым крючком», заставляя компонент автоматически выполнить нужный оператор. При этом если эти действия не совпадают с действиями, определенными дополнительными SQL-операторами, они отвергаются. Например, пусть в свойстве ModifySQL записан следующий оператор:

```
UPDATE Credit SET Summ=1 WHERE N_Count=2
```

При наличии этого оператора какие бы изменения в какие бы записи ни вносил пользователь, фактически один раз будет выполнен указанный выше оператор и запись (записи) из таблицы БД Credit со значением столбца N_Count = 2 получит в столбце Summ значение 1. Причем в сетке, отображающей НД, изменения будут видны так, как они сделаны пользователем, но фактически в таблице БД изменения окажутся другими — они «проявятся» после повторного открытия НД. Замечу, что, если в свойство RefreshSQL поместить такой же оператор, как и в основное свойство SelectSQL, все изменения в реальных данных станут видимыми немедленно, поэтому в сетке пользователь увидит не свои изменения, а те, что фактически сделаны соответствующими SQL-операторами.

Свойства компонента TIBDataSet перечислены в табл. 6.11.

Таблица 6.11. Свойства компонента TIBDataSet

Свойство	Описание
`property BufferChunks: Integer;`	Указывает, на сколько записей будет увеличиваться буфер компонента по мере его наполнения
`property Database: TIBDatabase;`	Содержит ссылку на связанный компонент TIBDatabase
`property DBHandle: IntPtr;`	Содержит дескриптор набора данных
`property DeleteSQL: TStrings;`	Содержит SQL-предложение для удаления записей. Если свойство содержит пустую строку, в НД нельзя удалять записи
`property InsertSQL: TStrings;`	Содержит SQL-предложение для вставки записей. Если свойство содержит пустую строку, в НД нельзя вставлять записи
`property ModifySQL: TStrings;`	Содержит SQL-предложение для изменения записей. Если свойство содержит пустую строку, в НД нельзя редактировать записи
`property Params: TIBXSQLDA;`	Содержит параметры для параметрического запроса
`property Prepared: Boolean;`	Если содержит True, осуществляется подготовка перед исполнением запроса
`property QDelete: TIBSQL;`	Содержит ссылку на объект, который инкапсулирует предложение **DELETE**
`property QInsert: TIBSQL;`	Содержит ссылку на объект, который инкапсулирует предложение **INSERT**

Свойство	Описание
`property QModify: TIBSQL;`	Содержит ссылку на объект, который инкапсулирует предложение **UPDATE**
`property QRefresh: TIBSQL;`	Содержит ссылку на объект, который инкапсулирует предложение **REFRESHSQL**
`property QSelect: TIBSQL;`	Содержит ссылку на объект, который инкапсулирует предложение **SELECT**
`property RefreshSQL: TStrings;`	Содержит SQL-предложение, позволяющее обновить НД. Обычно оно просто совпадает с предложением `SelectSQL`
`property SelectSQL: TStrings;`	Содержит предложение **SELECT**, создающее НД
`property StatementType: TIBSQLTypes;`	Содержит тип SQL-предложения (см. далее)
`property Transaction: TIBTransaction;`	Содержит ссылку на связанный компонент `TIBTransaction`
`property TRHandle: IntPtr;`	Содержит дескриптор транзакции
`property UpdateObject: TIBDataSetUpdateObject;`	Содержит ссылку на компонент `TIBUpdateSQL`, с помощью которого можно обновить НД только для чтения
`type TIBUpdateRecordTypes = set of (cusModified, cusInserted, cusDeleted, cusUnmodified, cusUninserted);` `property UpdateRecordTypes: TIBUpdateRecordTypes;`	Указывает, какие записи в кэшированном наборе видны при обновлении: `cusModified` — модифицированные записи; `cusInserted` — вставленные записи; `cusDeleted` — удаленные записи; `cusUnmodified` — только неизмененные записи; `cusUninserted` — только не вставленные записи
`property UpdatesPending: Boolean;`	Указывает, содержит ли буфер обновления записи

Определение типа для свойства `StatementType`:

```
type TIBSQLTypes = set of (SQLUnknown, SQLSelect, SQLInsert,
                           SQLUpdate, SQLDelete, SQLDDL,
                           SQLGetSegment, SQLPutSegment,
                           SQLExecProcedure, SQLStartTransaction,
                           SQLCommit, SQLRollback,
                           SQLSelectForUpdate, SQLSetGenerator);
```

Здесь:

- `SQLUnknown` — тип неизвестен;
- `SQLSelect` — предложение **SELECT**;
- `SQLInsert` — предложение **INSERT**;
- `SQLUpdate` — предложение **UPDATE**;
- `SQLDelete` — предложение **DELETE**;

- SQLDDL — предложение на языке DDL;
- SQLGetSegment — читает часть открытого BLOB-столбца;
- SQLPutSegment — записывает часть BLOB-столбца;
- SQLExecProcedure — выполняет хранимую процедуру;
- SQLStartTransaction — стартует новую транзакцию;
- SQLCommit — подтверждает транзакцию;
- SQLRollback — откатывает транзакцию;
- SQLSelectForUpdate — хранимая процедура представляет собой набор предложений для обновления данных;
- SQLSetGenerator — устанавливает новое значение для генератора.

Специфические методы компонента TIBDataSet перечислены в табл. 6.12.

Таблица 6.12. Методы компонента TIBDataSet

Метод	Описание
`procedure ApplyUpdates;`	Переписывает изменения кэшированных данных в таблицу БД, но не завершает транзакцию
`type TCachedUpdateStatus = (cusUnmodified, cusModified, cusInserted, cusDeleted, cusUninserted);`	Возвращает статус кэшированных изменений: cusUnmodified — данные не редактировались; cusModified — данные редактировались; cusInserted — были вставлены записи; cusDeleted — были удалены записи; cusUninserted — записи не вставлялись
`function CachedUpdateStatus: TCachedUpdateStatus;`	
`procedure CancelUpdates;`	Очищает буфер изменений
`procedure FetchAll;`	Получает все записи от текущей до конца файла и сохраняет их локально
`function GetFieldData(FieldNo: Integer; Buffer: IntPtr): Boolean;`	Читает в буфер Buffer данные из столбца по его номеру FieldNo или из объекта-поля Field
`function GetFieldData(Field: TField; Buffer: IntPtr): Boolean;`	
`function LocateNext(const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean`	Позиционирует курсор на запись ниже текущей: KeyFields — имя ключевого столбца; KeyValues — искомое значение; Options — дополнительные условия поиска
`procedure Prepare;`	Подготавливает компонент к выполнению запроса
`procedure UnPrepare;`	Освобождает ресурсы, занятые при выполнении метода Prepare

В отличие от стандартного метода Locate, который, как и Lookup, доступен компоненту, метод LocateNext ищет запись от текущей к концу НД.

Специфические события компонента TIBDataSet перечислены в табл. 6.13.

Таблица 6.13. События компонента TIBDataSet

Событие	Описание
`property DatabaseDisconnected: TNotifyEvent;`	Возникает после разрыва соединения с БД
`property DatabaseDisconnecting: TNotifyEvent;`	Возникает перед разрывом соединения с БД
`property DatabaseFree: TNotifyEvent;`	Возникает при разрушении связанного компонента TIBDatabase

Компонент TIBSQL

Компонент TIBSQL предназначен для выполнения SQL-запросов с минимальными затратами времени. Он не может быть связан с компонентом TDataSource и, следовательно, с визуализирующими компонентами. В отличие от других компонентов-наборов, компонент TIBSQL происходит непосредственно от компонента TComponent, поэтому все его свойства, методы и события «свои», хотя многие из них в функциональном отношении повторяют одноименные свойства, методы и события других компонентов-наборов.

Для ускорения доступа к данным компонент использует специальные классы записей и полей. Он создает однонаправленный курсор, так что перемещение по НД возможно только сверху вниз. Наконец, он лишен многочисленных событий After*XXXX*, Befor*XXXX*, а также OnCalcField, так что в НД нельзя создать вычисляемые поля или реализовать бизнес-правила на основе обработчиков этих событий.

Специфические свойства компонента TIBSQL перечислены в табл. 6.14.

Таблица 6.14. Свойства компонента TIBSQL

Свойство	Описание
`property Bof: Boolean;`	Содержит True, если курсор стоит перед первой записью НД
`property Database: TIBDatabase;`	Содержит ссылку на связанный компонент TIBDatabase
`property DBHandle: IntPtr;`	Содержит дескриптор БД
`property Eof: Boolean;`	Содержит True, если курсор стоит после последней записи НД
`property FieldIndex: [FieldName: String]: Integer;`	Возвращает индекс столбца по его имени
`property Fields[const Idx: Integer]: TIBXSQLVAR;`	Открывает индексный доступ к столбцам НД в формате TIBXSQLVAR (см. далее)

продолжение

Таблица 6.14 (продолжение)

Свойство	Описание
`property GenerateParamNames: Boolean;`	Если содержит `True`, компонент создает имена параметров для параметрического запроса
`property GoToFirstRecordOnExecute: Boolean;`	Если содержит `True`, курсор сразу после открытия НД устанавливается на его первую запись
`property Handle: IntPtr;`	Содержит дескриптор запроса
`property Open: Boolean;`	Содержит `True`, если НД открыт
`property ParamCheck: Boolean;`	Если содержит `True`, параметры параметрического запроса проверяются перед его выполнением
`property Params: TIBXSQLDA;`	Содержит параметры запроса в формате `TIBXSQLDA` (см. далее)
`property Plan: String;`	Содержит план выполнения запроса после его подготовки
`property Prepared: Boolean;`	Содержит `True`, если запрос был подготовлен методом `Prepare`
`property RecordCount: Integer;`	Содержит количество записей в запросе
`property RowsAffected: Integer;`	Возвращает количество записей, которые были изменены в результате выполнения запроса
`property SQL: TStrings;`	Содержит текст запроса
`property SQLType: TIBSQLTypes read FSQLType;`	Содержит тип запроса (тип `TIBSQLTypes` см. в описании свойства `StatementType` компонента `TIBDataSet` в разделе «Компонент TIBDataSet»)
`property Transaction: TIBTransaction;`	Содержит ссылку на связанный компонент `TIBTransaction`
`property TRHandle: IntPtr;`	Содержит дескриптор транзакции
`property UniqueRelationName: String;`	Используйте это свойство, чтобы указать уникальное реляционное имя для запроса, касающегося одной таблицы

Тип `TIBXSQLVAR` содержит практически те же методы приведения типов `AS`*XXXX*, как и «обычный» тип `TField`, а его свойство `Value` имеет тип `Variant`. Таким образом, работа со значениями столбцов компонента `TIBSQL` практически не отличается от работы со значениями столбцов других компонентов-наборов. Это же замечание относится к типу `TIBXSQLDA` и работе со значениями параметров. Указанные типы поддерживают специфику скоростной работы компонента.

Методы компонента `TIBSQL` перечислены в табл. 6.15.

Таблица 6.15. Методы компонента TIBSQL

Метод	Описание
`procedure BatchInput (InputObject: TIBBatchInput);`	Выполняет параметрический запрос для объекта InputObject
`procedure BatchOutput (OutputObject: TIBBatchOutput);`	Выполняет параметрический запрос для объекта OutputObject
`procedure CheckClosed;`	Возбуждает исключение, если запрос не закрыт
`procedure CheckOpen;`	Возбуждает исключение, если запрос закрыт
`procedure CheckValidStatement;`	Возбуждает исключение, если SQL-предложение ошибочно
`procedure Close;`	Закрывает запрос
`function Current: TIBXSQLDA;`	Возвращает дескриптор текущей записи
`procedure ExecQuery;`	Выполняет запрос
`function FieldByName[FieldName: String]: TIBXSQLVAR;`	Возвращает столбец по его имени
`procedure FreeHandle;`	Освобождает ресурсы InterBase, связанные с запросом
`function Next: TIBXSQLDA;`	Смещает курсор к следующей записи
`procedure Prepare;`	Готовит запрос к выполнению

Нетрудно обнаружить, что единственным навигационным методом является метод Next. Таким образом, последовательный просмотр записей НД возможен только от начала набора к его концу.

Компонент имеет единственное событие, которое возникает при изменении SQL-запроса:

```
property OnSQLChanging: TNotifyEvent;
```

Компонент TIBDatabaseInfo

Компонент TIBDatabaseInfo предназначен для предоставления программе дополнительной служебной информации о присоединенной базе данных. Компонент не имеет специфических методов и событий. Свойства компонента TIBDatabaseInfo перечислены в табл. 6.16.

Таблица 6.16. Свойства компонента TIBDatabaseInfo

Свойство	Описание
`property Allocation: LongIntInt;`	Возвращает количество распределенных в памяти страниц БД
`property BackoutCount: TStringList;`	Возвращает количество удаленных версий страниц

продолжение

Таблица 6.16 (продолжение)

Свойство	Описание
`property BaseLevel: SmallInt;`	Возвращает 2-байтный номер версии БД
`property CurrentMemory: LongIntInt;`	Возвращает объем памяти (в байтах), используемый сервером
`property Database: TIBDatabase;`	Содержит ссылку на связанный компонент TIBDatabase
`property DBFileName: String;`	Возвращает имя БД
`property DBImplementationClass: Integer;`	Возвращает номер реализуемого класса БД
`property DBImplementationNo: Integer;`	Возвращает номер реализации БД
`property DBSiteName: String;`	Возвращает имя сайта БД (обычно совпадает с именем компьютера, на котором она расположена)
`property DBSQLDialect: Integer;`	Возвращает используемый в БД диалект SQL
`property DeleteCount: TStringList;`	Возвращает список БД, которые были удалены с момента последнего присоединения к ней пользователя
`property ExpungeCount: TStringList;`	Возвращает количество удаленных записей
`property Fetches: Integer;`	Возвращает количество записей, прочитанных в локальный буфер
`property ForcedWrites: Integer;`	Возвращает номер режима записи в БД: 0 — асинхронный режим; 1 — синхронный режим
`property InsertCount: TStringList;`	Возвращает количество вставленных записей
`property Marks: Integer;`	Возвращает количество записей в буфер кэша
`property MaxMemory: Integer;`	Возвращает максимальный объем памяти в байтах, который использовался с момента первого присоединения к БД
`property NoReserve: Integer;`	Указывает, резервировалась ли память для сохранения устаревших записей (0 — резервировалась)
`property NumBuffers: Integer;`	Возвращает количество распределенных буферов кэша
`property PageSize: Integer;`	Возвращает размер страницы в байтах
`property SweepInterval: Integer;`	Указывает, сколько транзакций должно успешно завершиться до момента автоматической чистки страниц БД от ненужных версий записей
`property UserNames: TStringList;`	Содержит список имен всех пользователей, присоединенных к БД

Компонент TIBSQLMonitor

Компонент TIBSQLMonitor предназначен для создания отладочного инструментария. С помощью этого инструментария программист может осуществлять мониторинг всех SQL-предложений, которыми обмениваются клиент и сервер. Для мониторинга следует установить нужные флаги в свойство TraceFlags связанного компонента TIBDatabase (см. раздел «Компонент TIBDatabase»).

Компонент TIBSQLMonitor не имеет собственных свойств и методов. Единственное событие возникает, если клиент передал серверу SQL-предложение, тип которого выбран флагами TraceFlags:

```
TSQLEvent = procedure(EventText: String) of object;
property OnSQL: TSQLEvent;
```

В этом случае обработчик события в параметре EventText может прочитать соответствующее сообщение.

Компонент TIBExtract

Компонент TIBExtract впервые появился в версии 7 Delphi. Он предназначен для получения так называемых *метаданных* — списков таблиц, индексов, хранимых процедур и тому подобных сущностей БД. Использовать его довольно просто. В свойства Database и Transaction компонента нужно поместить ссылки на соответствующие связанные компоненты, после чего с помощью метода ExtractObject компонент может прочитать в свое свойство Items нужную информацию. Например, следующий обработчик покажет в многострочном текстовом поле список всех таблиц БД с указанием их определений:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  IBDatabase1.Connected := True;
  IBExtract1.ExtractObject(eoTable);
  Memo1.Lines.Assign(IBExtract1.Items);
end;
```

Метод ExtractObject определен следующим образом:

```
procedure ExtractObject(ObjectType : TExtractObjectTypes;
  ObjectName : String = ''; ExtractTypes : TExtractTypes = []);
```

Единственный обязательный параметр ObjectType определяет вид нужных метаданных и может иметь одно из следующих значений:

- eoDatabase — получить список всех баз данных;
- eoDomain — получить список доменов;
- eoTable — получить список таблиц;
- eoView — получить список представлений;
- eoProcedure — получить список хранимых процедур;
- eoFunction — получить список внешних (пользовательских) функций;

- `eoGenerator` — получить список генераторов;
- `eoException` — получить список исключений;
- `eoBLOBFilter` — получить список BLOB-фильтров;
- `eoRole` — получить список ролей;
- `eoTrigger` — получить список триггеров;
- `eoForeign` — получить список внешних ключей;
- `eoIndexes` — получить список индексов;
- `eoChecks` — получить список проверяемых ограничений;
- `eoData` — получить все данные объекта `ObjectName`.

Параметр `ObjectName` определяет имя сущности. Если он опущен, будет получена информация обо всех сущностях данного вида. Параметр `ExtractTypes` содержит набор уточняющих флагов.

Хранимые процедуры, триггеры и представления

7

Описываемые в этой главе механизмы типичны только для клиент-серверных БД. Они позволяют перенести на сервер реализацию части бизнес-правил и, таким образом, значительно снизить нагрузку на сеть.

Создание хранимых процедур

Хранимая процедура создается следующим оператором:

```
CREATE PROCEDURE ИмяПроцедуры
   [(<входной_параметр> <тип_данных>
   [,<входной_параметр> <тип_данных> ...])]
   [RETURNS
    (<выходной_параметр> <тип_данных>
   [,<выходной_параметр> <тип_данных> ...])]
AS
   [<объявление локальных переменных>]
BEGIN
   <оператор>;
   [<оператор>; ...]
END
```

Входные параметры служат для передачи в процедуру значений из вызывающего приложения. Изменять значения входных параметров в теле процедуры бессмысленно — эти изменения будут потеряны после окончания работы процедуры.

Выходные параметры служат для возвращения результирующих значений. Значения выходных параметров устанавливаются в теле процедуры и после окончания ее работы передаются в вызывающее приложение.

Как входные, так и выходные параметры могут быть опущены, если в них нет необходимости.

В следующем примере описывается хранимая процедура, которая по номеру накладной возвращает список связанных с ней книг:

```
CREATE PROCEDURE Books_For_Nakl (Nakl INT)
RETURNS
  (Name VARCHAR(150), Quan SMALLINT,
   Price FLOAT, Summa FLOAT)
AS
DECLARE VARIABLE Aut VARCHAR(40);
DECLARE VARIABLE Pub VARCHAR(40);
BEGIN
  FOR SELECT BName, BAuthor, BPublish,
             MQuan, MPrice, MQuan*MPrice
      FROM Books, Moves
      WHERE NaklID=:Nakl AND BookID=MBook
      ORDER BY BName
      INTO :Name, :Aut, :Pub, :Quan, :Price, :Summa
  DO
    BEGIN
      Name = Name||'/ '||Aut||'/ '||Pub;
      SUSPEND;
    END
END
```

Для вызова хранимой процедуры, возвращающей набор данных (как в представленном примере), используется компонент-запрос (TQuery, TIBQuery и т. п.).

Покажем, как можно использовать хранимую процедуру BOOKS_FOR_NAKL для вывода списков книг при просмотре накладных NAKLS:

1. С помощью утилиты IBConsoly или SQL Explorer создайте показанную ранее процедуру BOOKS_FOR_NAKL для БД IB_BIBL.
2. Начните новый VCL-проект и поместите на пустую форму компоненты TDatabase, TTable и TQuery (категория BDE).
3. С помощью свойства AliasName свяжите компонент Database1 с БД IB_BIBL. Поместите в его свойство DatabaseName локальный псевдоним A.
4. Выберите в списке свойства DatabaseName компонента Table1 псевдоним A, в списке свойства TableName — таблицу NAKLS и откройте НД. Назовите его по имени таблицы — NAKLS и создайте для него объекты-поля.
5. Свяжите компонент Query1 с псевдонимом A и разместите в его свойство SQL такой параметрический запрос:

   ```
   SELECT * FROM BOOKS_FOR_NAKL (:NAKLID)
   ```

6. Раскройте окно свойства Params запроса и настройте параметр NAKLID: DataType = ftInteger; ParamType = ptInput. Назовите компонент по имени детальной таблицы MOVES и создайте для него объекты-поля.
7. С помощью трех панелей TPanel разбейте поверхность формы на три части: поместите в свойства Align двух из них значения alBottom, а в такое же свойство третьей — значение alClient.

8. Разместите на форме два источника TDataSource. Свяжите один с НД NAKLS, а второй — с НД MOVES. Для НД MOVES в свойстве DataSource сошлитесь на источник, связанный с главным НД NAKLS.
9. Для создания подстановочных полей поместите на форму еще два компонента TTable и свяжите один с таблицей FIRMS, а второй — с таблицей TYPES.

Вид окна работающей программы показан на рис. 7.1.

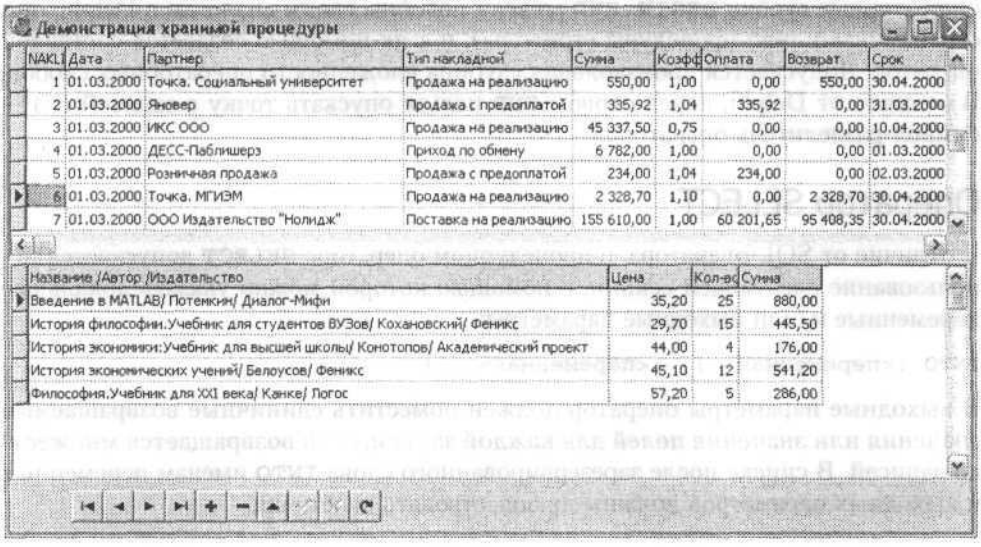


Рис. 7.1. Использование хранимой процедуры для создания детального НД

Алгоритмический язык процедур и триггеров

Хранимые процедуры и триггеры кодируются на специальном алгоритмическом языке, в основе которого лежит язык SQL, дополненный локальными переменными, условными, циклическими операторами, операторами присваивания и некоторыми другими средствами. Отдельные операторы могут располагаться на нескольких строках и отделяются друг от друга точкой с запятой. Регистр букв в операторах не имеет значения. В любом месте программы можно поставить комментарий, обрамленный парой символов /* и */. Символы каждой пары, а также символы операций сравнения >=, <=, <>, != (не равно), !< (не меньше), !> (не больше) не должны разрываться пробелами.

Локальные переменные

Локальные переменные объявляются для тех же целей, что и переменные в Delphi: они служат для временного запоминания промежуточных результатов вычислений. Объявление переменной имеет такой формат:

```
DECLARE VARIABLE <имя_переменной тип>;
```

Если локальные переменные, входные или выходные параметры указываются в теле SQL-оператора, их именам должны предшествовать двоеточия. В других операторах имя переменной указывается без двоеточия (см. представленный ранее пример).

Операторные скобки

Операторные скобки **BEGIN...END** во всем подобны своим аналогам в Delphi: они ограничивают тело процедуры (триггера), и с их помощью создается составной оператор. Допускается произвольная глубина вложенности операторных скобок. В отличие от Delphi, перед словом **END** нельзя опускать точку с запятой (;) — символ-разделитель операторов.

Оператор SELECT

В отличие от SQL-оператора, в процедурном операторе **SELECT** допускается использование следующей секции, с помощью которой можно указать локальные переменные и/или выходные параметры:

```
INTO :<переменная> [, :<переменная>...]
```

В выходные параметры оператор должен поместить единичные возвращаемые значения или значения полей для каждой записи, если возвращается множество записей. В списке после зарезервированного слова **INTO** именам переменных и выходных параметров должны предшествовать двоеточия.

Условный оператор

Условный оператор имеет формат

```
IF (<условное_выражение>) THEN <оператор1> [ELSE <оператор2>]
```

В отличие от Delphi, в процедурном условном операторе условное выражение должно заключаться в круглые скобки. Условное выражение строится из знаков операций сравнения и логических операций **AND**, **OR**, **NOT**. Приоритет операций сравнения меньше, чем логических операций, что избавляет от необходимости расстановки группирующих скобок:

```
IF (NSum>0 AND NType IN (0, 6)) THEN ...
```

Операторы FOR и SUSPEND

Оператор **FOR** предназначен для получения множества значений. Он имеет такой формат:

```
FOR SELECT <тело оператора SELECT> INTO <список переменных> DO
<оператор>
```

Для каждой записи, извлекаемой оператором **SELECT**, выполняется оператор <оператор>, которым часто бывает оператор **SUSPEND**. Этот оператор приоста-

навливает работу процедуры до тех пор, пока вызывающая программа не потребует очередной порции данных. На основе операторов **FOR** и **SUSPEND** строятся процедуры, возвращающие наборы данных.

Оператор присваивания

Оператор присваивания служит для занесения значения выражения в переменную или выходной параметр:

```
<переменная> = <выражение>
```

Выражение и переменная должны быть совместимого типа. Перед знаком равенства двоеточие не ставится. Имена переменных и выходных параметров в обеих частях оператора указываются без двоеточия. Поскольку в InterBase нет логического типа, выражение может быть только арифметическим или строковым. В арифметическом выражении используются четыре арифметические операции, поля, переменные, константы и обращения к функциям пользователя. В строковом выражении допустима единственная операция ||, определяющая конкатенацию (сцепление) строк.

Операторы WHILE и EXIT

Оператор **WHILE** соответствует аналогичному оператору Delphi:

```
WHILE (<условие>) DO <оператор>
```

Оператор **EXIT** служит для принудительного завершения работы процедуры (в триггере оператор **EXIT** использовать нельзя):

```
IF (NType IN (2,3)) THEN EXIT;
```

Оператор EXECUTE PROCEDURE

С помощью оператора **EXECUTE PROCEDURE** процедура (триггер) может обратиться к другой хранимой процедуре (к триггеру обратиться нельзя). Формат оператора:

```
EXECUTE PROCEDURE <имя> [<параметр> [, <параметр> ...]]
[RETURNING_VALUES :<переменная> [, :<переменная> ...]];
```

После имени вызываемой процедуры перечисляются входные параметры, если они есть, а после **RETURNING_VALUES** — локальные переменные и/или выходные параметры, в которые вызываемая процедура должна поместить результат обращения. Именам переменных должны предшествовать двоеточия.

Оператор **EXECUTE PROCEDURE** предназначен для обращения к так называемым процедурам действия, которые либо вообще не возвращают значений, либо возвращают лишь один экземпляр группы значений. Таким образом, процедуры действия не создают наборов данных. Для обращения к таким процедурам из программы предназначен специальный компонент `TStoredProc` (`TADOStoredProc`, `TSQLStoredPros` и т. п.).

Исключения

С помощью следующего оператора процедура (триггер) может возбудить исключение:

```
EXCEPTION <имя_исключения>;
```

Процедурное исключение имеет собственное имя <имя_исключения> и предварительно создается оператором

```
CREATE EXCEPTION <имя_исключения>"<сообщение>";
```

Здесь <сообщение> — произвольный текст. Обработка этого оператора осуществляется блоком **WHEN...DO**, который имеет такой формат:

```
WHEN {<ошибка> [, <ошибка> ...] | ANY}
DO <оператор>
<ошибка>=
  {EXCEPTION <имя_исключения> | SQLCODE <номер> | GDSCODE
<код_ошибки>
```

Здесь:

- **EXCEPTION**, **SQLCODE**, **GDSCODE** — зарезервированные слова;
- <номер> — код ошибки SQL;
- <код_ошибки> — код ошибки InterBase;
- **ANY** — зарезервированное слово, определяющее обработку любых ошибок;
- <оператор> — любой оператор или блок **BEGIN...END**.

Блок **WHEN...DO** должен стоять непосредственно перед ключевым словом **END**, даже если ему предшествует слово **SUSPEND**.

Если возбуждено исключение, выполнение текущего блока прекращается и управление передается блоку **WHEN...DO**. Если такового нет или в нем не предусмотрена реакция на данное исключение, ищется блок **WHEN...DO** в блоке более высокого уровня и т. д. Если исключение не обработано, выполнение процедуры прекращается, ликвидируются все сделанные ею изменения и выдается сообщение <сообщение> (если исключение обработано оператором <оператор> в блоке **WHEN...DO**, сообщение не выдается).

Пусть в БД имеется таблица `Users` со списком всех сотрудников, имеющих те или иные права доступа к данным. Среди них есть сотрудник Иванов с неограниченными правами доступа, данные о котором из таблицы удалять нельзя. Создадим исключение:

```
CREATE EXCEPTION No_Delete 'Нельзя удалить пользователя Иванова!'
```

Кроме того, создадим триггер:

```
CREATE TRIGGER Bef_Del_Users FOR Users
BEFORE DELETE AS
BEGIN
  IF (old.Name = 'Иванов') THEN
    EXCEPTION No_Delete;
END
```

Теперь попытаемся очистить всю таблицу следующим оператором:

```
DELETE FROM Users
```

В этом случае появится соответствующее сообщение и все записи таблицы будут сохранены.

Замечу, что объявленные в БД исключения можно увидеть, раскрыв узел Exceptions в окне утилиты SQL Explorer. Однако если среди них есть хотя бы одно, текст которого набран кириллицей, этот список станет недоступным.

Компоненты доступа к хранимым процедурам

Как уже говорилось, если хранимая процедура возвращает набор данных, ее вызов помещается в компонент-адаптер (для технологии ADO.NET) или в компонент-запрос (остальные технологии). В этом разделе рассматриваются компоненты для доступа к процедурам, возвращающим единичное значение или единственный экземпляр нескольких значений. Такие процедуры называются процедурами действия.

Компонент TStoredProc

Компонент `TStoredProc` используется в технологии BDE.NET и предназначен для обращения к процедурам действия. При работе с сервером InterBase он не может создать механизм курсора, поэтому его нельзя использовать в качестве НД. Для серверов другого типа (например, Oracle и Microsoft SQL Sever) компонент `TStoredProc` может служить в качестве НД.

Свойства компонента `TStoredProc` перечислены в табл. 7.1.

Таблица 7.1. Свойства компонента TStoredProc

Свойство	Описание
`type TParamBindMode = (pbByName, pbByNumber);` `property ParamBindMode: TParamBindMode;`	Устанавливает способ связи параметров компонента с параметрами процедуры: `pbByName` — по имени (в этом случае имена параметров компонента должны совпадать с именами параметров процедуры); `pbByNumber` — по номеру (1-й с 1-м, 2-й со 2-м и т. д.)
`property ParamCount: Word;`	Содержит количество параметров процедуры
`property Params: TParams;`	Открывает доступ к параметрам процедуры (см. далее)
`property Prepared: Boolean;`	Если содержит `True`, процедура подготовлена к работе (см. далее)
`property StoredProcName: String;`	Указывает имя процедуры

Свойство Params имеет тип TParams, который представляет собой класс со свойствами Count, Items и ParamValue. Свойство Count указывает количество параметров компонента. Другое свойство открывает доступ к значению параметра по его имени:

```
property ParamValues[const ParamName: String]: Variant;
```

Еще одно свойство позволяет получить описание параметра по его индексу:

```
property Items[Index: Word]: TParam;
```

Описание задается классом TParam, который содержит тип параметра (входной, выходной или входной и выходной), тип его значения, его имя и значение по умолчанию. Эти свойства можно проверить и установить в окне инспектора объектов. Для этого сначала задается псевдоним БД в свойстве DatabaseName, затем с помощью раскрывающегося списка свойства StoredProcName выбирается нужная хранимая процедура. В этот момент компонент считывает из БД описание параметров процедуры и соответствующим образом настраивает свое свойство Params. Теперь после двойного щелчка на компоненте или с помощью команды Fields Editor его контекстного меню можно вызвать окно редактора параметров и работать с параметрами так, как если бы они были объектами-полями компонентов TTable и TQuery.

Получить или установить значение параметра можно с помощью свойства Params.ParamValues:

```
spNakls.Params.ParamValues['Nakl'].AsInteger := 123;
```

Свойство Prepared позволяет подготовить процедуру к выполнению. Подготовка заключается в том, что сервер оптимизирует выполнение и компиляцию процедуры, после чего сохраняет результат компиляции. Это сокращает до минимума время обращения к процедуре. Если в свойство поместить значение False, сервер освободит ресурсы, выделенные для хранения результата компиляции, и при каждом обращении к процедуре всякий раз будет заново готовить ее к выполнению.

Важнейшие методы компонента TStoredProc перечислены в табл. 7.2.

Таблица 7.2. Методы компонента TStoredProc

Метод	Описание
`procedure ExecProc;`	Требует от сервера выполнить процедуру
`procedure GetResults;`	Возвращает результат обращения к процедуре для серверов Sybase и MS SQL Server
`function ParamByName(const Value: String): TParam;`	Позволяет обратиться к параметру по его имени

В отличие от свойства Params.ParamValues, функция ParamByName возвращает объект класса TParam, содержащий не только значение параметра, но и все

другие его свойства (имя, тип и т. д.). Класс `TParam` содержит свойства `As`*XXXX*, позволяющие обратиться к значению параметра:

```
spNakls.ParamByName('Nakl').AsInteger := 123;
```

Общая схема работы с компонентом такова. Сначала с помощью свойства `DatabaseName` компонент связывается с нужной БД. Затем из списка свойства `StoredProcName` выбирается нужная хранимая процедура. Эти два этапа обычно реализуются при конструировании формы. Три следующих — в работающей программе: обращением к методу `ParamByName` устанавливаются необходимые входные параметры, методом `ExecProc` процедура выполняется и методом `ParamByName` получаются нужные выходные параметры.

Проиллюстрируем сказанное следующим примером (рис. 7.2):

1. Создайте в БД IB_BIBL процедуру, которая по номеру книги возвращает максимальную партию ее продажи, то есть максимальное количество экземпляров, проданных одному покупателю:

```
CREATE PROCEDURE MAX_QUAN (BOOKID INT)
RETURNS (BOOK_QUAN INT) AS
BEGIN
  SELECT
    MAX(MQUAN)
  FROM
    MOVES, NAKLS
  WHERE
    MBOOK=:BOOKID AND NAKLS.NAKLID=MOVES.NAKLID AND NTYPE IN (1,7)
  INTO :BOOK_QUAN;
END
```

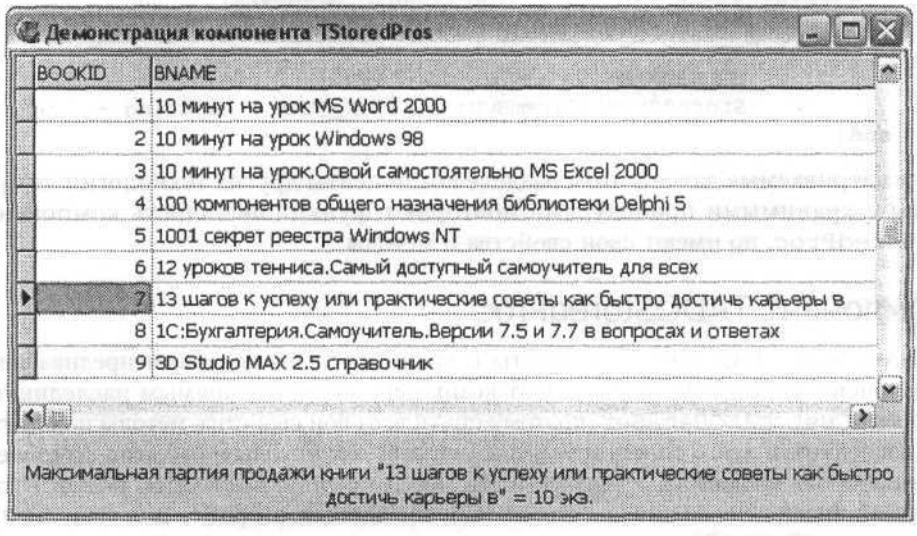


Рис. 7.2. Иллюстрация использования процедуры действия

2. Начните новый VCL-проект, поместите на пустую форму панель TPanel, расположите ее вдоль нижнего края формы (Align = alBottom) и на панель поместите надпись TLabel, установив такие ее параметры:
   - AutoSyze = False;
   - Anchors = [akLeft, akTop, akRight];
   - Alignment = taCenter;
   - WordWrap = True.
3. Поместите на форму компоненты TDatabase, TTable, TStoredProc (категория BDE), TDataSuorce (категория Data Access) и TDBGrid (категория Data Controls). Настройте компонент Database1 на связь с БД IB_BIBL, а компоненты Table1 и StoredProc1 — на связь с локальным псевдонимом БД.
4. В списке свойства TableName компонента Table1 выберите таблицу BOOKS, назовите компонент именем таблицы, создайте для него объекты-поля и откройте НД.
5. В списке свойства StoredProcName компонента StoredProc1 выберите имя процедуры MAX_QUAN.
6. Свяжите источник DataSource1 с НД BOOKS, а сетку DBGrid1 — с источником. Сетка должна наполниться данными.
7. Напишите такой обработчик события OnDataChange компонента DataSourse1:

```
procedure TForm1.DataSource1DataChange(Sender: TObject;
                                       Field: TField);
// Выполняет хранимую процедуру и опубликовывает ее результат
begin
  // Настраиваем входной параметр:
  StoredProc1.ParamByName('BOOKID').Value := BOOKSBOOKID.Value;
  StoredProc1.ExecProc;  // Выполняем процедуру
  // Получаем и используем ее результат:
  Label1.Caption := 'Максимальная партия продажи книги "' +
          BOOKSBNAME.Value + '" = ' +
          StoredProc1.ParamByName('BOOK_QUAN').AsString + ' экз.';
end;
```

Рассматриваемые далее в этом разделе компоненты других технологий для работы с хранимыми процедурами повторяют функциональность компонента TStoredProc, но имеют свои свойства и методы.

Компонент TADOStoredProc

В технологии dbGo.NET для работы с хранимыми процедурами предназначен компонент TADOStoredProc. Этот компонент является прямым наследником класса TCustomADODataSet, поэтому почти все свои свойства, методы и события он наследует от этого класса (см. раздел «Свойства, методы и события компонентов-наборов» в главе 4). У него есть два метода активизации: Open и ExecProc. Первый предназначен для получения наборов данных, второй — для выполнения процедур действия.

Компонент TSQLStoredProc

В технологии dbExpress.NET для работы с хранимыми процедурами предназначен компонент `TSQLStoredProc`. Этот компонент является прямым наследником класса `TCustomSQLDataSet`, поэтому почти все свои свойства, методы и события он наследует от этого класса (см. раздел «Компонент TSQLDataSet» в главе 5). У него есть два метода активизации: `Open` и `ExecProc`. Первый предназначен для получения наборов данных, второй — для выполнения процедур действия. Два свойства строкового типа определяют имя хранимой процедуры: `PackageName` и `StoredProcName`. Первое используется только для сервера Oracle, в котором хранимые процедуры помещаются в пакеты. Его специфичный метод `NextRecordSet` используется для получения от процедуры очередной порции информации, если процедура возвращает набор данных.

Компонент TIBStoredProc

В технологии IBX.NET для работы с хранимыми процедурами предназначен компонент `TIBStoredProc`. Этот компонент является прямым наследником класса `TCustomIBDataSet`, поэтому большинство своих свойств, методов и событий он наследует от этого класса (см. раздел «Компонент TIBDataSet» в главе 6).

Специфичные для компонента свойства и методы перечислены в табл. 7.3 и 7.4.

Таблица 7.3. Свойства компонента TIBStoredProc

Свойство	Назначение
`property Filtered: Boolean;`	Указывает, будет ли фильтроваться НД, получаемый от хранимой процедуры
`property ParamCount; Integer;`	Содержит общее количество параметров хранимой процедуры
`property Params: TParams;`	Содержит параметры хранимой процедуры
`property Prepared: Boolean;`	Указывает, будет ли предварительно готовиться выполнение хранимой процедуры
`property StoredPropertyNames: TString;`	Возвращает список всех хранимых процедур, определенных в серверной БД
`property StoredPropertyName; String;`	Содержит имя исполняемой хранимой процедуры

Таблица 7.4. Методы компонента TIBStoredProd

Метод	Назначение
`procedure CopyParams(Value: TParams);`	Копирует параметры хранимой процедуры в список `Value`
`procedure ExecProc;`	Выполняет хранимую процедуру действия

продолжение

Таблица 7.4 (продолжение)

Метод	Назначение
`function ParamByName(Name: String): TParam;`	Открывает доступ к параметру хранимой процедуры по его имени
`procedure Prepare;`	Готовит хранимую процедуру к исполнению
`procedure UnPrepare;`	Ликвидирует последствия вызова метода `Prepare`

Создание триггеров

Триггер определяет программный отклик на изменение данных. Он создается таким SQL-оператором:

```
CREATE TRIGGER <Имя_триггера> FOR <Имя_таблицы>
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION <номер>]
AS
  [<объявление локальных переменных>]
BEGIN
  <операторы>
END
```

Здесь:

- `ACTIVE | INACTIVE` — указывает, будет ли триггер активен (по умолчанию используется значение `ACTIVE`);
- `BEFORE | AFTER` — определяет, будет ли триггер срабатывать до (`BEFORE`) или после (`AFTER`) изменения данных;
- `DELETE | INSERT | UPDATE` — указывает тип изменения данных;
- `POSITION <номер>` — необязательный параметр, определяющий очередность срабатывания триггера, если для одной и той же таблицы создано несколько однотипных триггеров (по умолчанию используется значение `POSITION 0`); триггеры с меньшими номерами выполняются раньше.

При написании триггера используется процедурный язык (см. раздел «Алгоритмический язык процедур и триггеров»), в который добавлена возможность обращения к старым и новым значениям столбцов с помощью предикатов **OLD** и **NEW** соответственно.

Реализация бизнес-правил с помощью триггеров

В клиент-серверных БД с помощью триггеров следует реализовывать бóльшую часть бизнес-правил (если это возможно, *все* бизнес-правила). Преимущества такого решения очевидны: вся работа реализуется сервером, и клиентская программа избавляется от необходимости обмена данными с ним. В результате резко

снижается сетевой трафик, а система управления БД в целом становится надежнее и проще: если понадобится изменить бизнес-правила, это гораздо легче реализовать в одном месте, чем вносить изменения в многочисленные клиентские программы.

Как уже говорилось, бизнес-правила тесно связаны с сутью тех задач, которые решаются с помощью создаваемой СУБД, поэтому невозможно дать какие-то единые рецепты их реализации. В данном разделе на примере БД «Книголюб» мы рассмотрим лишь те действия, которые необходимо проделать при удалении данных о накладной и при удалении названия книги из списка связанных с накладной книг.

Перед удалением данных о накладной следует удалить данные обо всех связанных с ней книгах. Это реализуется таким простым триггером:

```
CREATE TRIGGER BEFORE_DEL_NAKLS FOR NAKLS
ACTIVE BEFORE DELETE AS
BEGIN
  DELETE FROM Moves M
  WHERE M.NaklID = OLD.NaklID;
END
```

Если в момент срабатывания триггера таблицы NAKLS и MOVES связаны реляционным отношением *один ко многим*, в таблице MOVES будут видны только те книги, которые относятся к удаляемой накладной, поэтому уточнение **WHERE** можно опустить:

```
CREATE TRIGGER BEFORE_DEL_NAKLS FOR NAKLS
ACTIVE BEFORE DELETE AS
BEGIN
  DELETE FROM Moves;
END
```

Удаление данных об отдельной книге влечет за собой более сложную логику действий: на основании типа накладной необходимо скорректировать количество экземпляров книги на складе и сальдо партнера, а также изменить сумму накладной:

```
CREATE TRIGGER BEFORE_DEL_MOVES FOR MOVES
ACTIVE BEFORE DELETE
AS
/* Объявление вспомогательных переменных: */
DECLARE VARIABLE TypeN INTEGER; /* Тип накладной */
DECLARE VARIABLE FirmN INTEGER; /* Шифр партнера */
DECLARE VARIBLE Coeff FLOAT;    /* Коэффициент накладной */
BEGIN
/* Помещаем тип накладной в переменную TypeN, шифр партнера
в переменную FirmN и коэффициент в Coeff */
  SELECT NType, NFirm
  FROM Nakls
  WHERE Nakls.NaklID=OLD.NaklID
  INTO :TypeN, :FirmN;
```

```
/* Коррекция данных в других таблицах зависит от типа накладной: */
  IF (:TypeN IN (0,3,4,6)) THEN
  BEGIN
/* Удаляется книга из накладной, связанной с приходом книг.
Уменьшаем количество книг на складе */
    UPDATE Books
    SET BQuan = BQuan - OLD.MQuan
    WHERE BookID = OLD.MBook;
    /* Корректируем сальдо партнера */
    IF (:TypeN=4) THEN
    /* Увеличиваем обменное сальдо */
      UPDATE Firms
      SET FChgDelta = FChgDelta + OLD.MQuan * OLD.MPrice * :Coeff
      WHERE FirmID = :FirmN;
    ELSE
    /* Увеличиваем финансовое сальдо */
      UPDATE Firms
      SET FFinDelta = FFinDelta + OLD.MQuan * OLD.MPrice * :Coeff
      WHERE FirmID = :FirmN;
  END ELSE BEGIN
/* Удаляется книга из накладной, связанной с уходом книг.
 Увеличиваем количество книг на складе */
    UPDATE Books
    SET BQuan=BQuan + OLD.MQuan
    WHERE BookID = OLD.MBook;
    IF (:TypeN=4) THEN
    /* Уменьшаем обменное сальдо */
      UPDATE Firms
      SET FChgDelta = FChgDelta - OLD.MQuan * OLD.MPrice * :Coeff
      WHERE FirmID = :FirmN;
    ELSE
    /* Уменьшаем финансовое сальдо */
      UPDATE Firms
      SET FFinDelta = FFinDelta - OLD.MQuan * OLD.MPrice * :Coeff
      WHERE FirmID = :FirmN;
  END
END
```

Многочисленные комментарии поясняют особенности тех или иных действий.

ПРИМЕЧАНИЕ

Если в текст объявления триггера, процедуры или любого другого компонента БД включить русскоязычный комментарий, этот текст станет недоступным (если включены англоязычные комментарии или комментариев вообще нет, текст объявления любого компонента БД можно увидеть на вкладке Text окна утилиты SQL Explorer).

Как видите, в триггерах допускаются локальные переменные и условные операторы. Я не привожу объявления других триггеров, связанных с редактированием данных о накладных и книгах, — моя цель состоит в том, чтобы продемонстрировать особенности разработки клиент-серверных систем, и я вовсе не хочу

утомлять вас деталями реализации конкретной учебной системы. Тем не менее не могу не обратить вашего внимания на следующую деталь. Триггер BEFORE_DEL_MOVES имеет серьезный недостаток: он не корректирует сумму накладной. Сделано это не случайно. Можно, например, в его начале вставить такие строки:

```
UPDATE Nakls
SET NSum = NSum - OLD.MQuan * OLD.MPrice * :Coeff
WHERE NaklID = OLD.MNakl;
```

Если это сделать, то триггер прекрасно справится с задачей удаления данных о книге из накладной, но попытка удаления всей информации по накладной окажется невозможной, так как триггер будет пытаться изменить поле в удаляемой накладной, что категорически запрещается сервером. Как решить эту задачу? Дело в том, что, в отличие от процедуры, триггеру нельзя передать параметр, который указывал бы на то, что удаляется информация по накладной в целом. Получив такой параметр, триггер просто не стал бы изменять сумму накладной, да и коррекцию сальдо партнера можно было бы в этом случае сделать в триггере BEF_DEL_NAKLS сразу для всей суммы, а не делать этого для каждой отдельной книги. Однако повторяю, триггеру нельзя передать параметр, а в БД не существует глобальных переменных. Когда я впервые столкнулся с подобной проблемой на практике, она показалась мне неразрешимой. Однако после некоторого раздумья я нашел, как мне кажется, вполне приемлемое решение. Я добавил в таблицу MOVES новое поле:

```
ALTER TABLE MOVEBOOK ADD IsDelNakl VARCHAR(1)
```

Это поле заполняется в процессе ввода значением F в триггере BEF_INS_MOVES:

```
CREATE TRIGGER BEF_INS_MOVES FOR MOVES BEFORE INSERT AS
BEGIN
  NEW.IsDelNakl = 'F';
  NEW.MoveID = GEN_ID(GEN_MOVEBOOK, 1);
END
```

Триггер BEF_DEL_NAKL первым делом выполняет такой оператор:

```
UPDATE MoveBook
SET IsDelNakl = 'T'
WHERE MNakl = OLD.NaklID;
```

Теперь триггер BEF_DEL_MOVES может проанализировать это поле и блокировать коррекцию суммы в накладной, если поле содержит T:

```
IF (OLD.IsDelNakl <> 'T') THEN
  UPDATE Nakls
  SET NSum = NSum - OLD.MQuan * OLD.MPrice * :Coeff
  WHERE NaklID = OLD.MNakl;
```

Правильные тексты триггеров можно увидеть с помощью утилиты SQL Explorer. Для этого раскройте узел рядом с нужной таблицей, затем раскройте узел Triggers, выделите название нужного триггера и перейдите на вкладку Text.

Изменение и удаление процедур и триггеров

Изменение процедур (триггеров) выполняется с помощью операторов **ALTER PROCEDURE** (**ALTER TRIGGER**), формат которых ничем не отличается от формата операторов создания процедуры (триггера), за исключением того, что при изменении триггера нельзя указывать часть **FOR** <имя_таблицы>. При выполнении оператора старое определение процедуры (триггера) заменяется новым.

Удалить процедуру (триггер) можно оператором **DROP PROCEDURE** (**DROP TRIGGER**) <имя_процедуры> (<имя_триггера>).

Представления

Представление (view) — это заранее составленный и хранящийся в БД SQL-запрос для выборки данных из одной или нескольких таблиц БД. Главное достоинство представления состоит в том, что, подобно хранимым процедурам, представление в момент создания оптимизируется и компилируется сервером, что сокращает время выполнения запроса.

Представление создается следующим оператором:

```
CREATE VIEW <Имя_представления> [(<Имя_столбца_VIEW>
                                 [, <Имя_столбца_VIEW> ...])]
AS <Оператор_SELECT> [WITH CHECK OPTION]
```

В этом операторе за именем представления следует необязательный список имен столбцов представления, а <Оператор_SELECT> — это полнофункциональный оператор **SELECT**. Необязательный параметр **WITH CHECK OPTION** запрещает вставлять в обновляемое представление записи, значения полей которых противоречат условиям выборки записей представления.

Например:

```
CREATE VIEW Nakl_View
AS
SELECT NaklID, NDate, FName, TName, NSum,
  NPayedSum, NRetSum, NCoeff, NRetDate
FROM Nakls, Firms, TypeNakl
WHERE
  FirmID = NFirm AND TypeID = NType
ORDER BY NDate
```

Представление является «виртуальной» таблицей БД, поэтому с ним могут работать компоненты-наборы, как с реальными таблицами БД: имя представления можно указывать в компоненте `TTable` как имя реальной таблицы, к нему может обращаться компонент `TQuery` с запросом типа

```
SELECT * FROM Nakl_View
```

Представление может быть обновляемым или необновляемым. Чтобы представление было обновляемым, оно должно отвечать следующим требованиям:

- состоять из записей одной таблицы;
- в столбцы представления должны входить все столбцы таблицы, имеющие атрибут **NOT NULL**;
- в представлении не должны использоваться агрегатные функции, параметры **DISTING** и **HAVING**, хранимые процедуры и пользовательские функции.

Если представление удовлетворяет этим условиям, к нему можно применять операторы **INSERT**, **UPDATE** и **DELETE**. Если в столбцах представления указаны не все столбцы **NOT NULL**, к нему можно применять только операторы **UPDATE** и **DELETE**.

Список имен столбцов указывается в том случае, когда в представление включаются вычисляемые столбцы:

```
CREATE VIEW Move_View(ID, Nakl, Book, Quan, Price, Summa)
AS SELECT MoveID, NaklID, MBook, MQuan, MPrice, MQuan*MPrice
   FROM Moves
   WHERE MBook = 1 WITH CHECK OPTION
```

Количество имен полей представления должно строго соответствовать количеству полей, возвращаемых оператором **SELECT**. Наличие в представлении вычисляемых столбцов не запрещает ему быть обновляемым (показанное выше представление — обновляемое, однако в него нельзя вставить записи, столбец `MBook` которых содержит значение, отличное от 1, — это запрещает параметр **WITH CHECK OPTION**).

Представление нельзя изменить, но его можно удалить оператором

```
DROP VIEW <Имя_представления>
```

8 Отчеты

В большинстве случаев данные, которые хранятся в таблицах БД, необходимо тем или иным способом публиковать, создавая так называемые *отчеты*. Для облегчения этой работы вместе с Delphi 2005 поставляются специальные программы (генераторы отчетов) Crystal Reports и Rave Reports. Обе программы могут работать на платформе .NET, но первая используется только в технологии ADO.NET (то есть в приложениях WinForms); для остальных технологий (для VCL-приложений) предусмотрена программа Rave Reports. В этой главе рассматриваются обе технологии.

Основы технологии Crystal Reports

Программа Crystal Reports позволяет выполнить выборку и форматирование результирующего НД из БД или другого источника данных. Эта программа способна создавать любые отчеты — от простого на основе одной таблицы до сложного отчета, группирующего данные из нескольких таблиц, с многочисленными диаграммами и производящего сложные вычисления над публикуемыми данными.

Создание отчетов в Crystal Reports осуществляется под управлением встроенного дизайнера отчетов, который запускается при добавлении к проекту WinForms объекта Crystal Reports. Созданный отчет запоминается в файле с расширением RPT. Для связи с отчетом в проект добавляется объект `ReportDocument`, а для визуализации отчета — объект `CrystalReportViewer`.

Пример создания простого отчета

В этом разделе описывается процесс создания несложного отчета, показанного на рис. 8.1. Отчет представляет собой прайс-лист оптового поставщика, созданный на основе информации, содержащейся в таблице `BOOKS`.

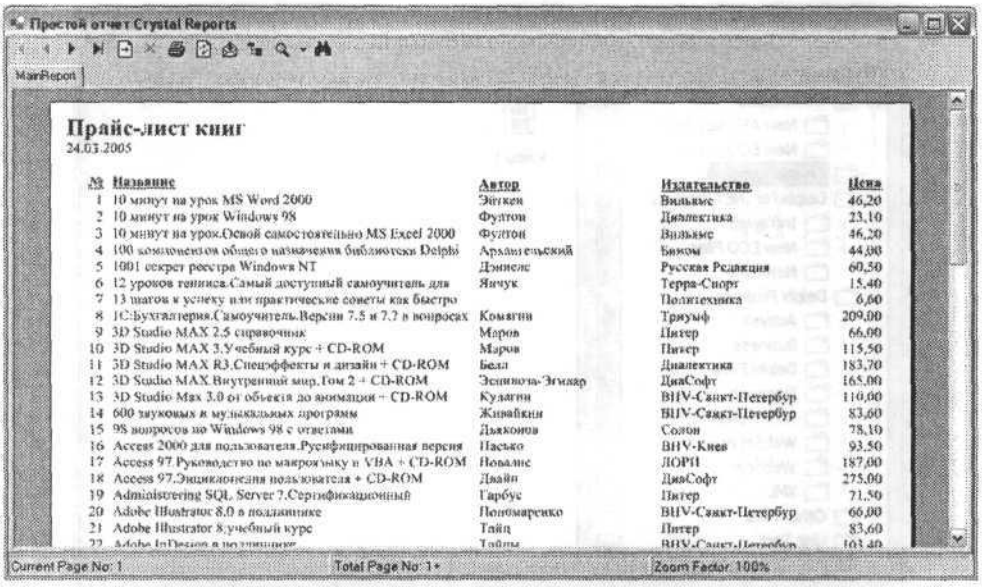


Рис. 8.1. Пример отчета в окне предварительного просмотра

К сожалению, Crystal Reports не может использовать технологию BDP-провайдера, поэтому для доступа к таблицам БД «Книголюб» мы будем использовать провайдер Microsioft OLE DB Provider for ODBC driver:

1. Настройте драйвер Microsoft Paradox Driver на работу с таблицами файл-серверной БД «Книголюб» так, как описано в разделе «Пример простой программы» главы 4.
2. Начните новое приложение WinForms и выберите команду File ▸ New ▸ Other. В появившемся окне New Item (рис. 8.2) раскройте узел Crystal Reports и на правой панели щелкните на значке Report, после чего закройте окно.

   Процесс создания отчета начинается с нового окна Crystal Report Gallery (рис. 8.3).

   В этом окне переключатель Using the Report Expert определяет пошаговый процесс создания под управлением эксперта. Переключатель As a Blank Report также инициирует пошаговый процесс, но он менее автоматизирован и требует бóльшего умения со стороны программиста. Переключатель From an Existing Report служит для создания отчета на основе уже существующего. Список Choose an Expert позволяет выбрать нужного эксперта, в том числе:

   - Standard — эксперт создания стандартного отчета, содержащего данные из одной или нескольких таблиц, причем разрешены фильтрация, сортировка, группировка и численная обработка данных, допустимо включение в отчет диаграмм, использование различных стилей отчета;
   - From Letter — создание писем с использованием информации из БД;
   - Form — создание отчета с логотипом на базе отсканированного изображения;

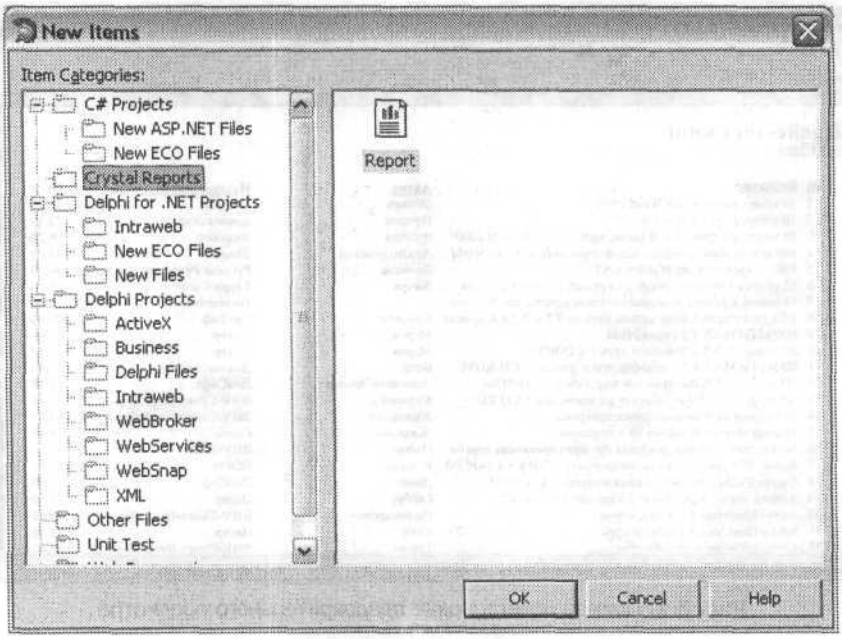


Рис. 8.2. Выбор нового отчета

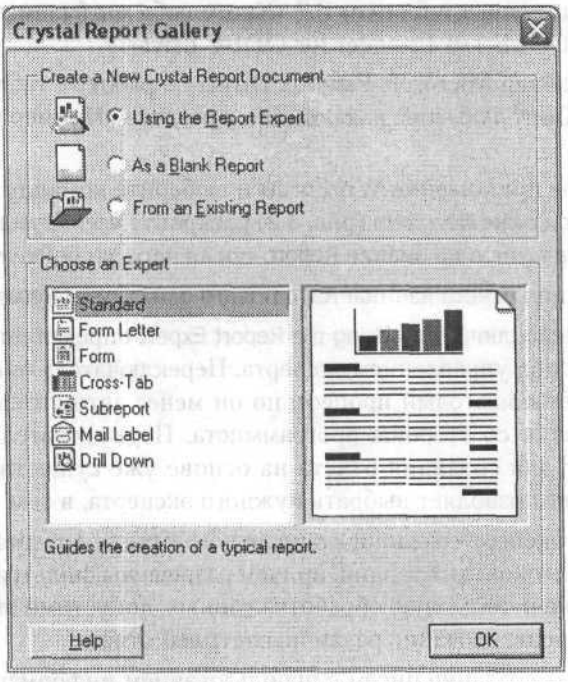


Рис. 8.3. Окно Crystal Report Gallery

- Cross-Tab — создание отчета с перекрестными ссылками;
- Subreport — создание детального отчета, данные из которого включаются в главный отчет;
- Mail Label — создание серии почтовых этикеток (конвертов);
- Drill Down — создание сводного отчета с возможностью его детализации.

3. Выберите в окне пошаговый процесс создания стандартного отчета под управлением эксперта, установив переключатель Using the Report Expert, и закройте окно щелчком на кнопке OK.
4. В окне Standard Report Expert (рис. 8.4) щелкните на узле OLE DB (ADO), чтобы активизировать процесс создания источника данных (рис. 8.5).

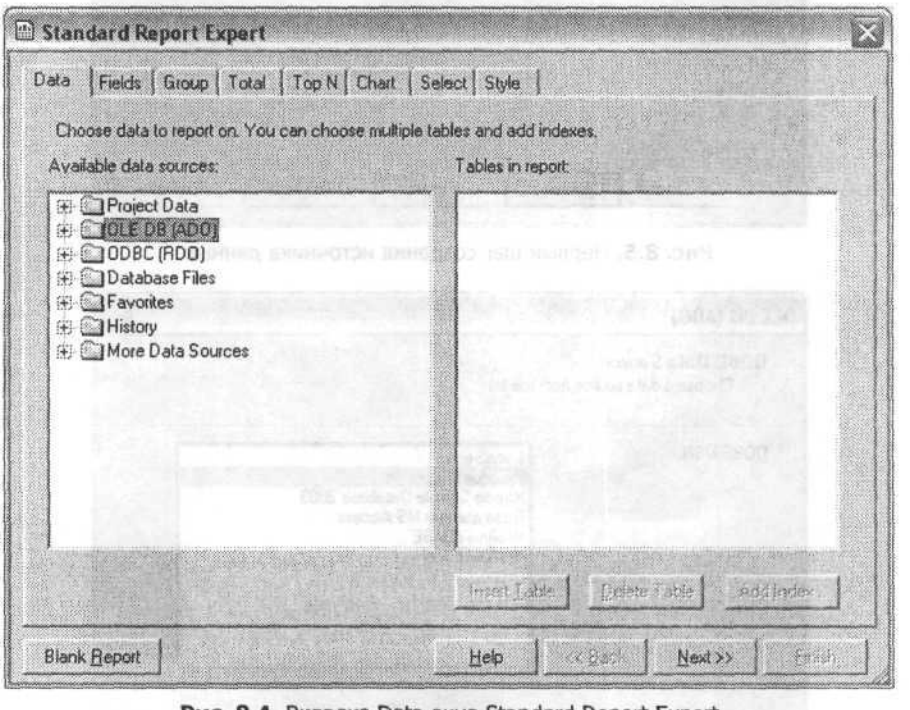


Рис. 8.4. Вкладка Data окна Standard Report Expert

5. Выберите провайдер Microsoft OLE DB Provider for ODBC Driver и щелкните на кнопке Далее.
6. В новом окне выберите драйвер Paradox files, как показано на рис. 8.6, и щелкните на кнопке Готово. На экране вновь появится вкладка Data окна Standard Report Expert (рис. 8.7).
7. Откройте узел C:\BIBLDATA, выберите таблицу BOOKS, щелкните на кнопке Insert Table, затем — на кнопке Next.

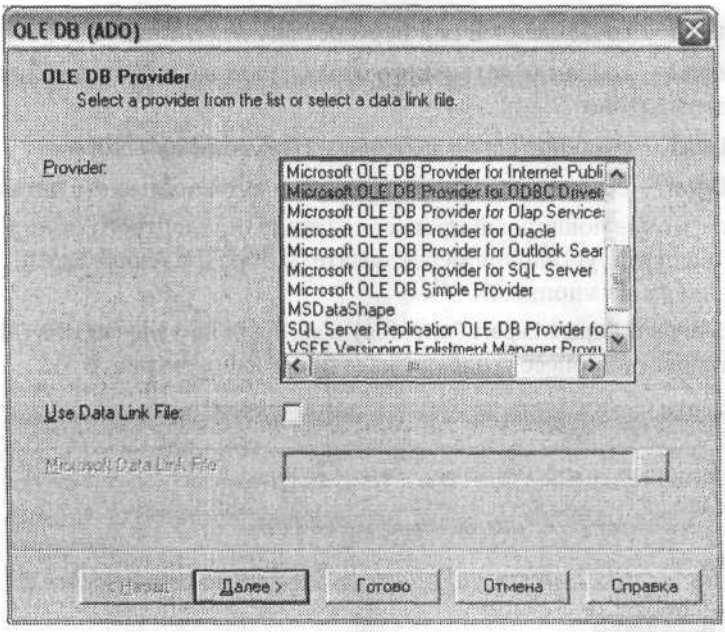


Рис. 8.5. Первый шаг создания источника данных

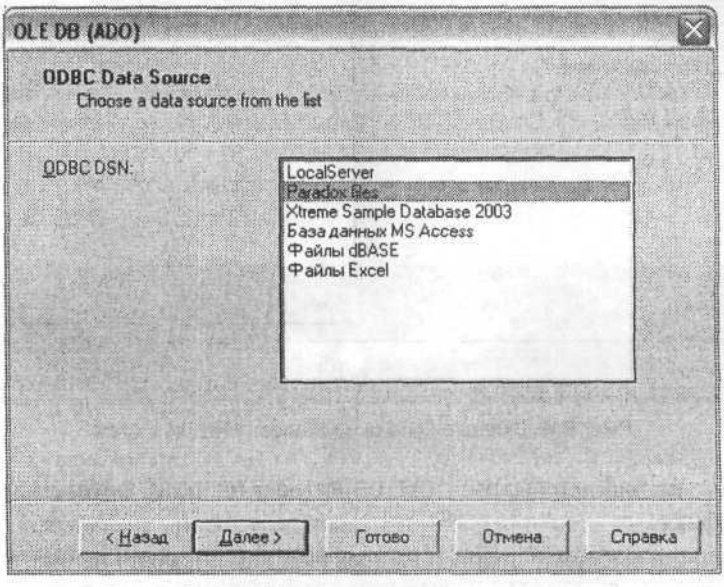


Рис. 8.6. Второй шаг создания источника данных

8. Щелчок на кнопке Next форсирует переход к работе с новой вкладкой окна Standard Report Expert. В нашем случае откроется вкладка Fields, позволяющая отобрать поля, данные из которых будут опубликованы в отчете (рис. 8.8).

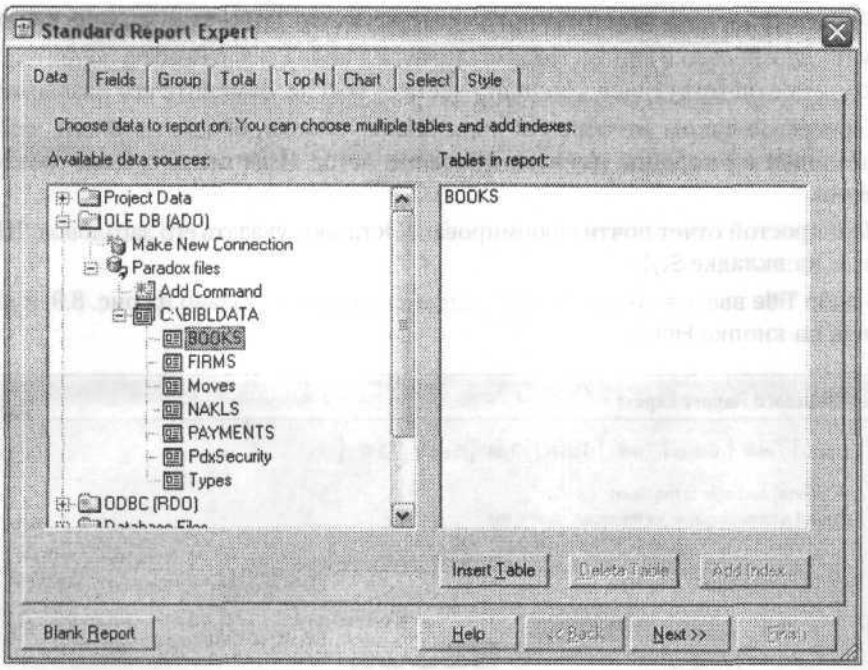


Рис. 8.7. Выбор таблицы БД

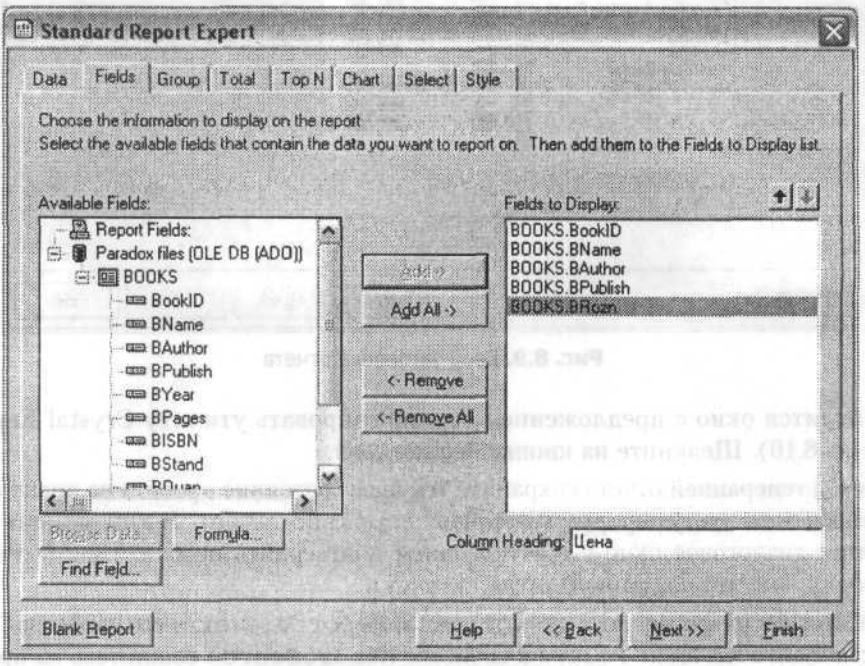


Рис. 8.8. Выбор отображаемых полей и их названий

9. Щелкните на поле BookID в списке Available Fields, затем — на кнопке Add. Поле будет перенесено в список Fields to Display, а в поле Column Heading появится заголовок соответствующей колонки отчета — BookID. Замените его названием №.
10. Перенесите таким же образом поля BName, BAuthor, BPublish и BRozn, заменив заголовки их колонок именами Название, Автор, Издательство, Цена соответственно.
11. Наш простой отчет почти сформирован. Осталось указать его заголовок. Щелкните на вкладке Style.
12. В поле Title введите текст Прайс-лист книг, как показано на рис. 8.9, и щелкните на кнопке Finish.

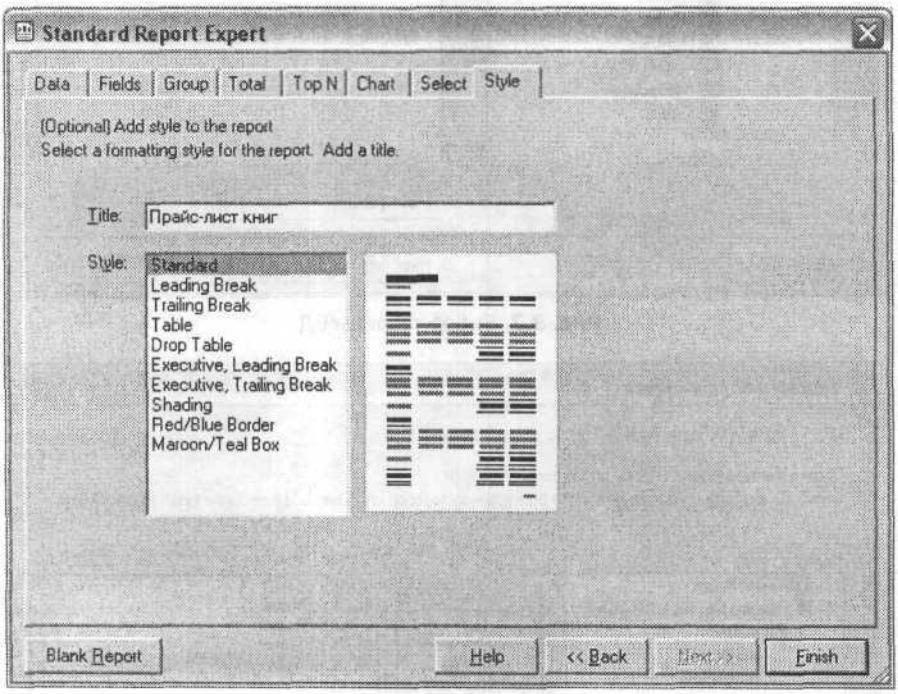


Рис. 8.9. Ввод заголовка отчета

13. Появится окно с предложением зарегистрировать утилиту Crystal Reports (рис. 8.10). Щелкните на кнопке Register Later.
14. Перед генерацией отчета сохраните текущее состояние проекта на диске и перенесите на пустую форму компонент `ReportDocument`. В этот момент появится диалоговое окно с предложением подтвердить необходимость генерации только что созданного проекта отчета.
15. Поместите на форму компонент `CrystalReportViewer`, в его свойство `Dock` установите значение `Fill`, а в свойстве `ReportSource` сошлитесь на проект отчета. Отчет и средство его просмотра готовы.

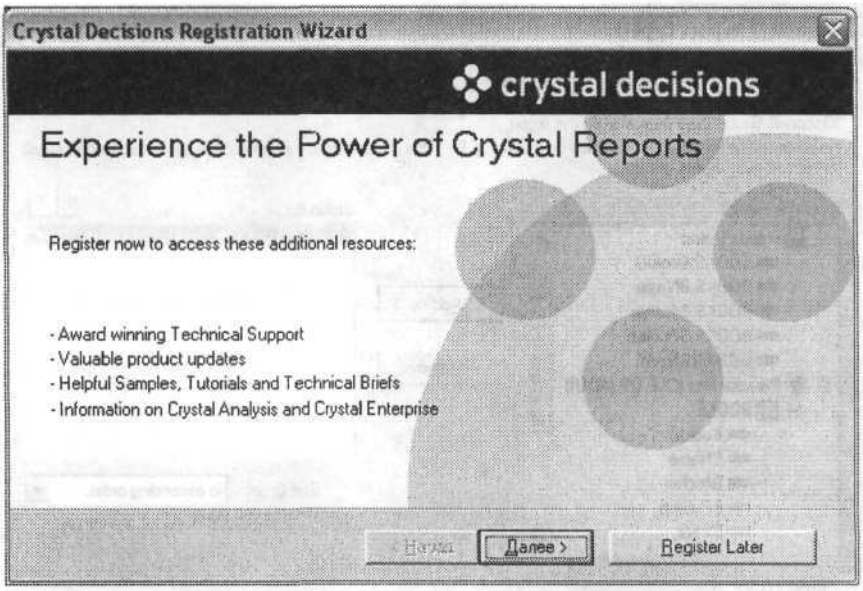


Рис. 8.10. Окно регистрации Crystal Reports

Дополнительные средства эксперта создания стандартного отчета

В созданном простейшем отчете не использовались многие возможности эксперта создания стандартного отчета. Эти возможности связаны с не рассмотренными нами вкладками Group, Total, TopN, Chart и Select.

Вкладка Group предназначена для группировки данных (рис. 8.11).

В левом списке указаны отобранные для публикации поля. Выберите нужное поле и перенесите его в правый список — группировка данных в предложении `GROUP BY` будет вестись по этому полю. Пусть, например, мы решили создать отчет с группировкой книг по издательствам. Выделите щелчком пункт BPublish и щелкните на кнопке Add. Вид отчета с группировкой показан на рис. 8.12.

Вкладка Total становится доступной только после группировки данных. Она позволяет добавлять в отчет сводную информацию по группам. Например, в нашем случае можно в конце каждой группы указать количество книг, выпущенных издательством и предлагаемых для продажи (рис. 8.13).

С этой целью удалите из правого списка добавленное экспертом единственное числовое поле BRozn и добавьте в него поле BName. Раскрывающийся список Summary Type определяет тип сводных данных. Содержимое этого списка зависит от типа поля, используемого для получения сводной информации. Для строковых полей он включает такие значения:

- maximum — максимальное значение;
- minimum — минимальное значение;

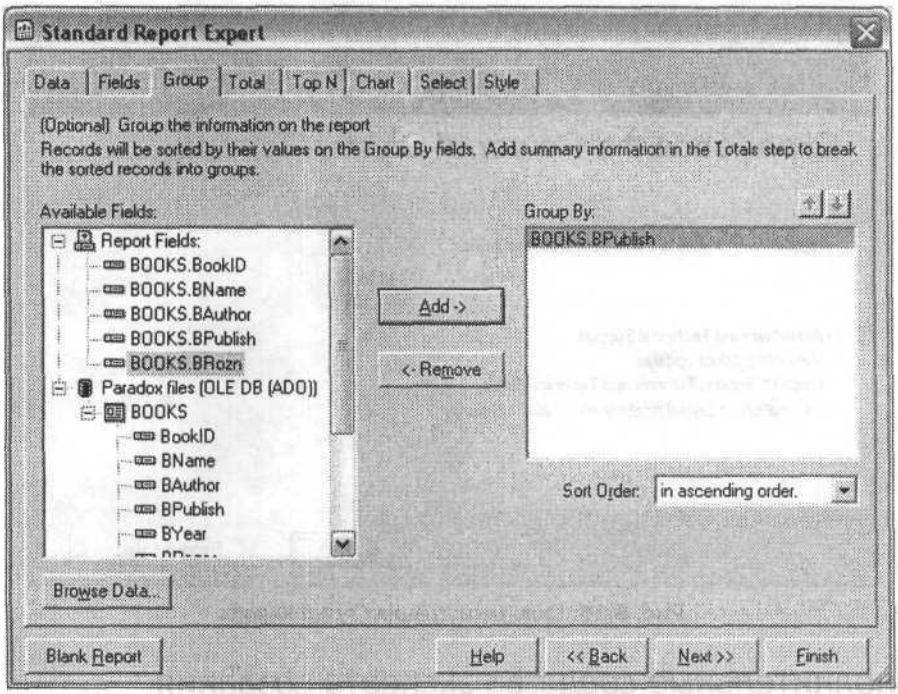


Рис. 8.11. Группировка данных

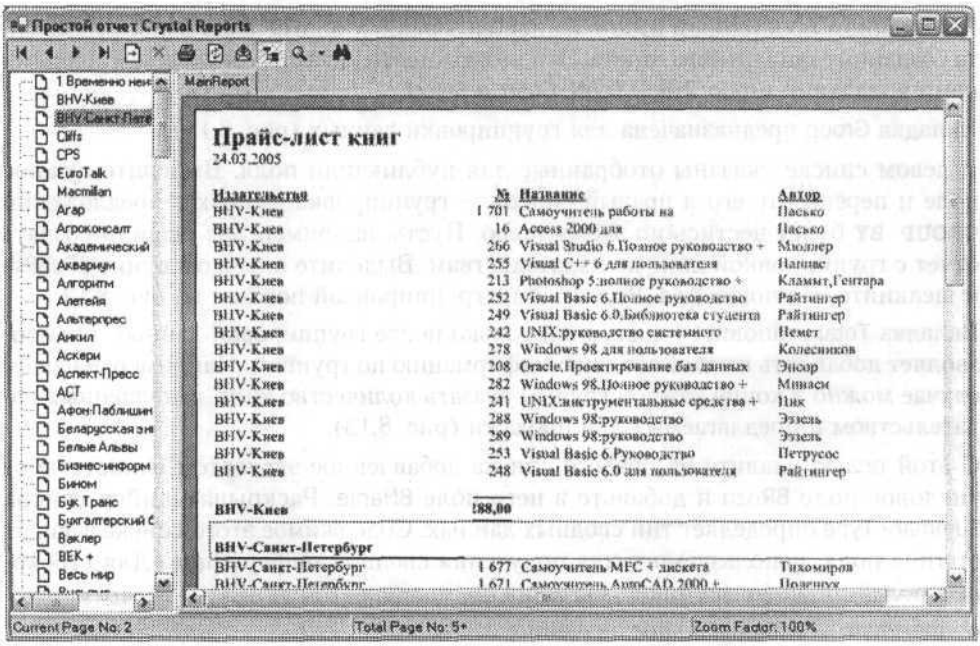


Рис. 8.12. Отчет с группировкой книг по издательствам

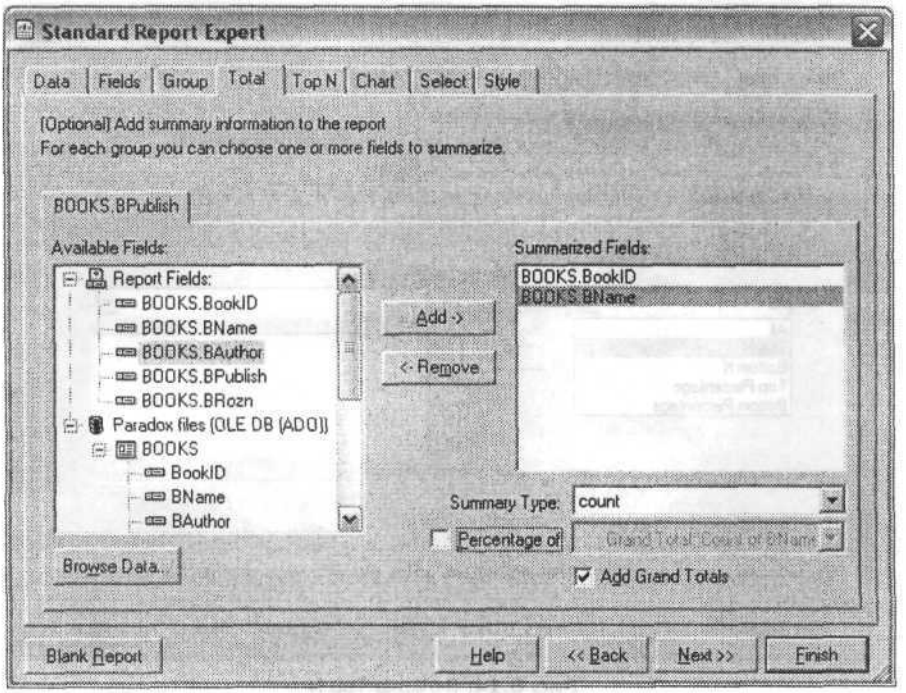


Рис. 8.13. Вкладка Total

- count — количество значений;
- distinct count — количество разных значений;
- Nth largest — *N*-е наибольшее значение;
- Nth smallest — *N*-е наименьшее значение;
- mode — статистическая мода значений, то есть наиболее вероятное значение;
- Nth most frequent — *N*-е наиболее часто встречающееся значение.

В случае числовых полей в списке имеются еще и такие значения:

- sum — сумма значений;
- average — среднее значение;
- weighted average — взвешенное среднее значение;
- medium — медиана значений поля;
- sample variance — дисперсия значений;
- sample standard deviation — среднее отклонение значений;
- population variance — дисперсия генеральной совокупности;
- correlation — коэффициент корреляции;
- covariance — смешанный момент второго порядка.

Вкладка Top N (рис. 8.14) позволяет добавить в отчет дополнительные средства анализа на основе полученных на вкладке Total сводных результатов.

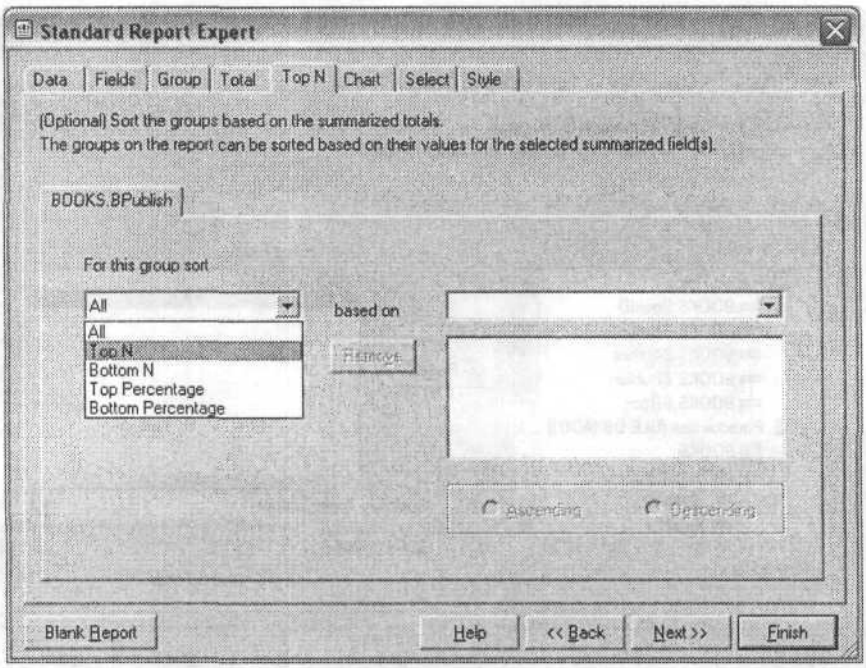


Рис. 8.14. Вкладка Top N

В списке For this group sort имеются следующие варианты средств анализа, определяющие, какие именно данные будут участвовать в получении сводных данных:

- All — все данные;
- Top N — первые *N* значений;
- Bottom N — последние *N* значений;
- Top Percentage — *N*-й процент первых;
- Bottom Percentage — *N*-й процент последних.

С помощью вкладки Chart (рис. 8.15) в отчет можно добавить диаграмму.

ПРИМЕЧАНИЕ

В БД «Книголюб» содержатся книги более 200 издательств. Чтобы не загромождать диаграмму таким количеством данных, ограничимся несколькими издательствами. Для этого на вкладке Top N в списке For this group sort выберите вариант Top N, в списке Based On — вариант Count of BOOKSBName, в поле Where N Is введите значение 7 и сбросьте флажок Include other groups with the label.

Эта вкладка имеет три внутренние вкладки — Type, Data и Text.

Вкладка Type представлена на рис. 8.15. С ее помощью выбираются тип диаграммы и ее расположение (по вертикали или по горизонтали листа). В списке Chart type перечислены все возможные типы диаграмм, справа от списка располагается от одной до шести (количество зависит от выбранного типа) кнопок, уточняющих вид диаграммы.

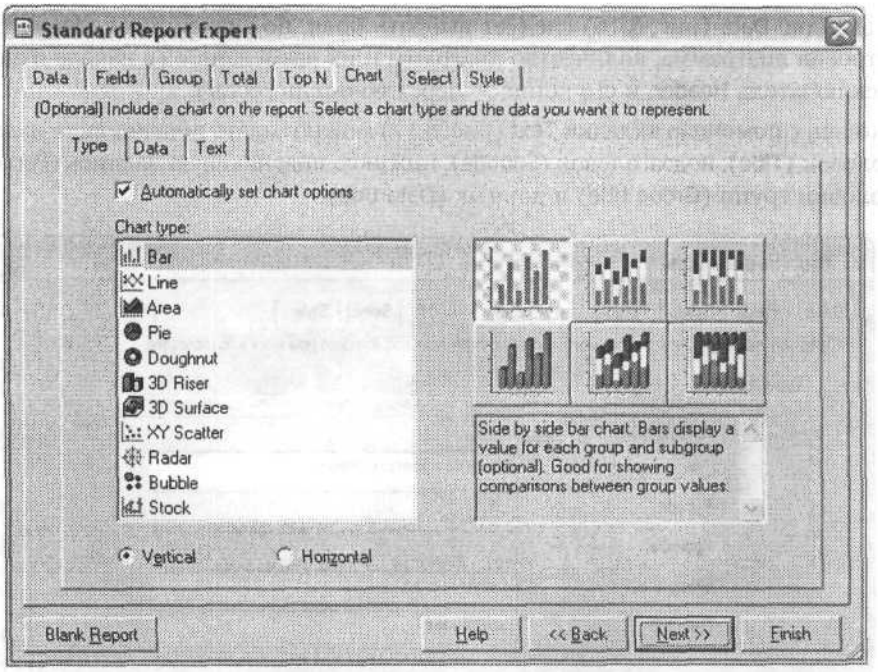


Рис. 8.15. Вкладка Type на вкладке Chart

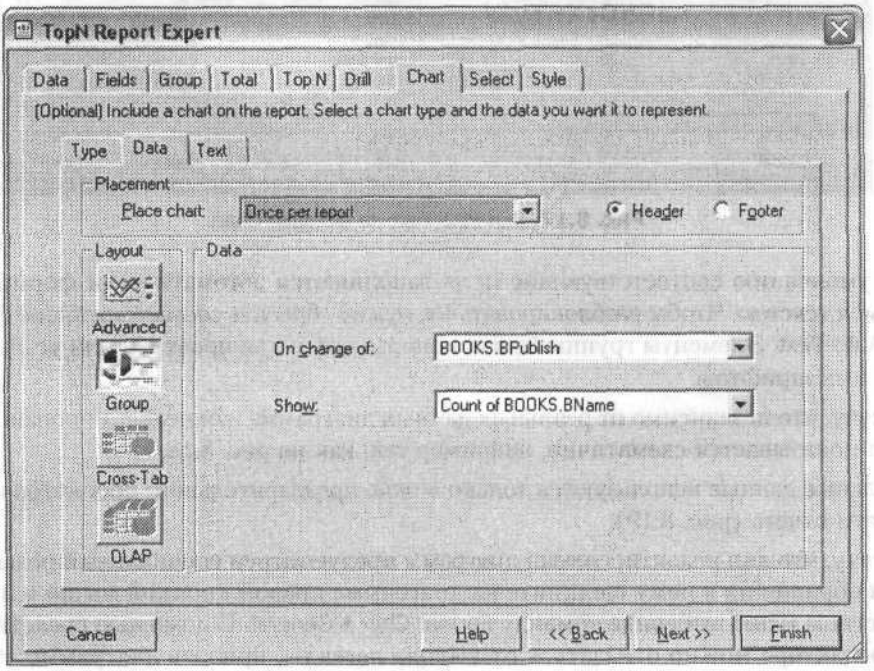


Рис. 8.16. Вкладка Data на вкладке Chart

На вкладке Data (рис. 8.16) следует выбрать поля, по значениям которых будет построена диаграмма, количество диаграмм и их размещение (в начале отчета — переключатель Header, в его конце — переключатель Footer).

Наконец, с помощью вкладки Text (рис. 8.17) можно задать поясняющие надписи: заголовок (Title), подзаголовок (Subtitle), нижнюю поясняющую надпись (Footnote), заголовки групп (Group title) и данных (Data title).

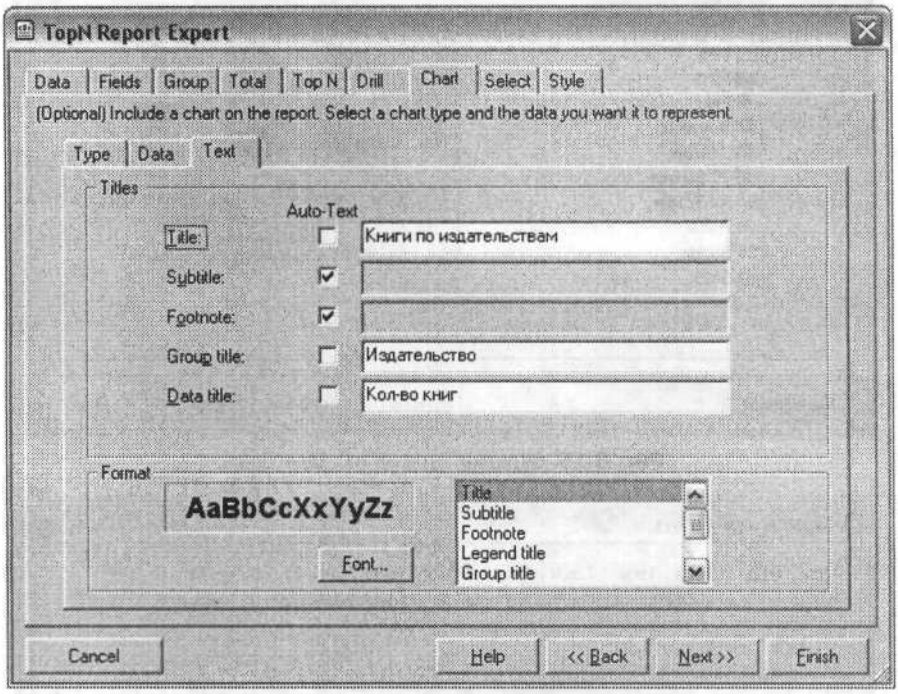


Рис. 8.17. Вкладка Text на вкладке Chart

По умолчанию соответствующие поля заполняются автоматически формирующимся текстом. Чтобы разблокировать их, нужно сбросить соответствующие флажки Auto-Text. Элементы группы Format используются для проверки или установки нужных шрифтов.

Замечу, что независимо от реальных данных диаграмма в окне формирования отчета показывается схематично, например так, как на рис. 8.18.

Реальные данные используются только в окне предварительного просмотра и при печати отчета (рис. 8.19).

Замечу, что для редактирования диаграмм предусмотрен специальный редактор. Для обращения к нему щелкните на диаграмме правой кнопкой мыши и в контекстном меню выберите команду Format Char ▸ General. С помощью средств этого редактора можно отказаться от вывода легенды, придать диаграмме «трехмерный» вид и т. д.

Рис. 8.18. Диаграмма в окне формирования отчета

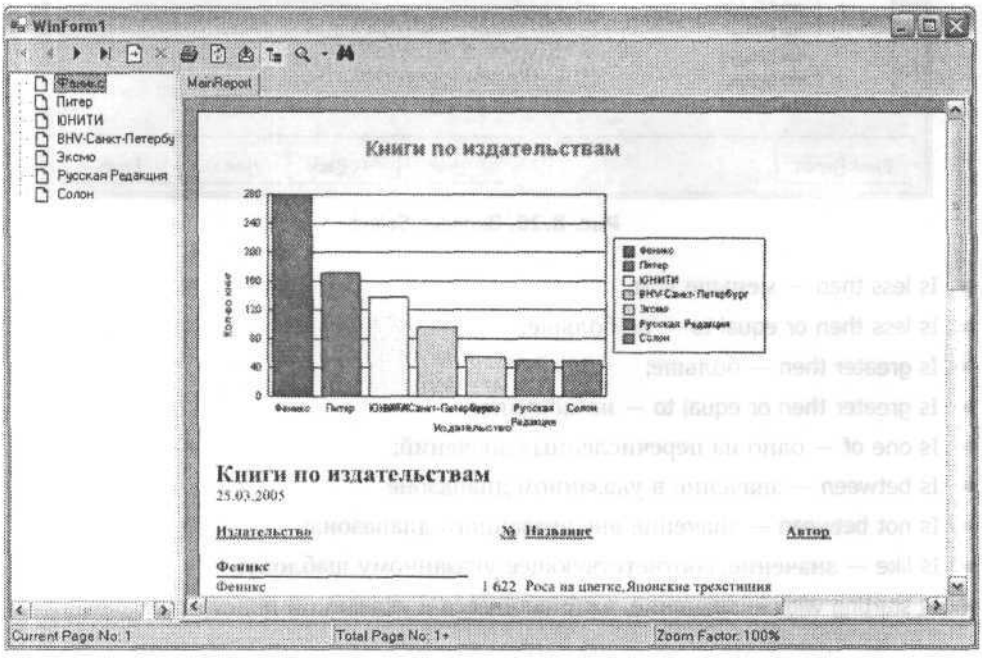


Рис. 8.19. Диаграмма в окне предварительного просмотра

Вкладка Select окна эксперта создания стандартного отчета показана на рис. 8.20. С ее помощью можно задать фильтр отбора данных. Фильтр записывается на специальном языке, допускающем названия полей, константы и символы следующих операций:

- Is equal to — равно;
- Is not equal to — не равно;

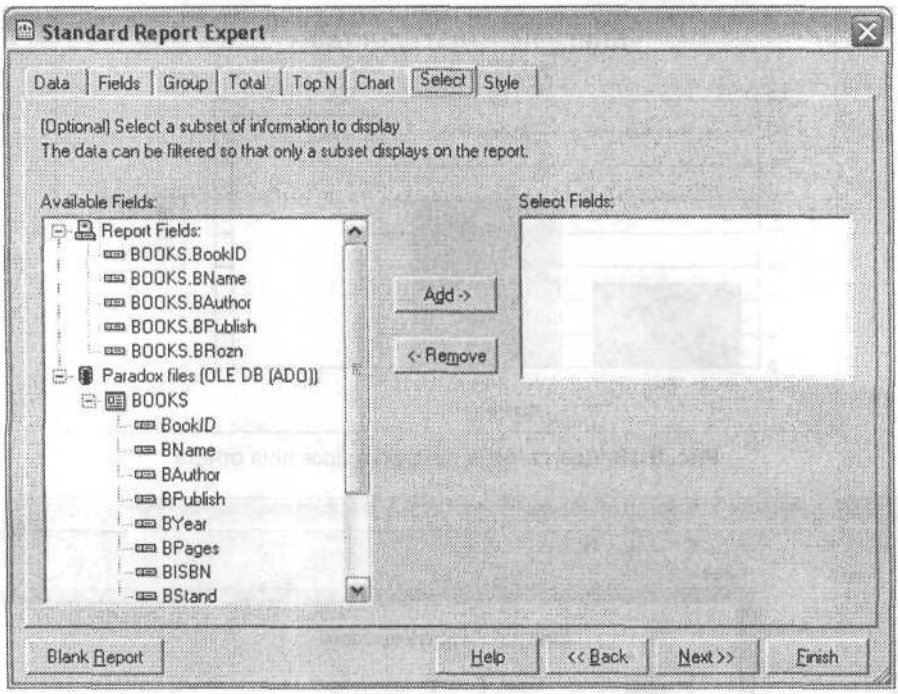


Рис. 8.20. Вкладка Select

- Is less then — меньше чем;
- Is less then or equal to — не больше;
- Is greater then — больше;
- Is greater then or equal to — не меньше;
- Is one of — одно из перечисленных значений;
- Is between — значение в указанном диапазоне
- Is not between — значение вне указанного диапазона;
- Is like — значение, соответствующее указанному шаблону;
- Is starting with — значение, начинающееся с указанной подстроки.

Экспорт отчета

С помощью кнопки Export Report в окне предварительного просмотра можно экспортировать отчет в файл формата PDF, который читается утилитой Acrobat Reader корпорации Adobe. На рис. 8.21 показан вид отчета в окне этой утилиты.

Как видите, качество преобразования очень высокое. А поскольку утилита Acrobat Reader распространяется бесплатно (http://www.adobe.com), созданный вами отчет сможет прочитать любой желающий.

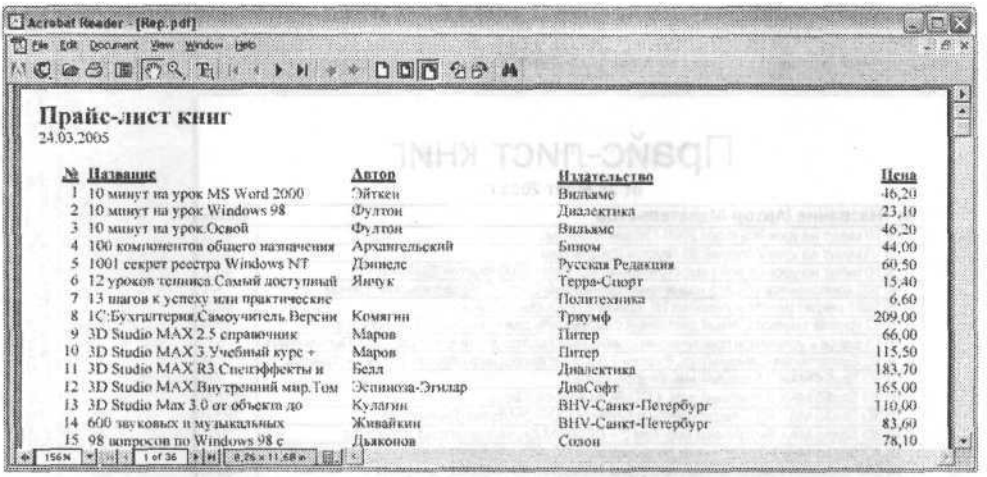


Рис. 8.21. Отчет в окне утилиты Acrobat Reader

Основы технологии Rave Reports

Технология Rave Reports разработана компанией Nevrona Designs, которая является авторизованным членом Комитета открытых систем и разрабатывает программное обеспечение для Delphi и C++ Builder. В ее основе лежит идея отделения процесса разработки отчета от процесса его создания за счет использования промежуточного документа, который называется *проектом отчета*. Проект отчета готовится с помощью среды Rave[1] (Report Authoring Visual Environment — визуальная среда автора отчета), а в создаваемую программу внедряется так называемая машина генерации отчета, которая по данным, полученным из файла проекта, строит документ отчета.

В категории Rave палитры компонентов (эта категория доступна только в режиме создания VCL-приложений; она не видна в окне Categories, так как располагается за категорией Internet — для доступа к ней используйте полосу прокрутки) имеется два ключевых компонента — `RvProject` и `RvSystem`, перенос которых в программу внедряет в нее машину генерации отчетов. Проект будущего отчета создается заранее с помощью утилиты Report Manager Designer, которая вызывается из Delphi и взаимодействует с загруженной в среду Delphi программой.

Пример создания отчета

В следующем примере будет создан простейший отчет, показанный на рис. 8.22:

1. Для создания главного окна начните новый проект, поместите на пустую форму компонент `Database` (категория BDE), раскройте список его свойства `AliasName` и выберите псевдоним `BIBLDATA`, в свойстве `DatabaseName` введите `AAA`.

[1] В буквальном переводе с английского «rave» означает «бредить», «бушевать», «выть».

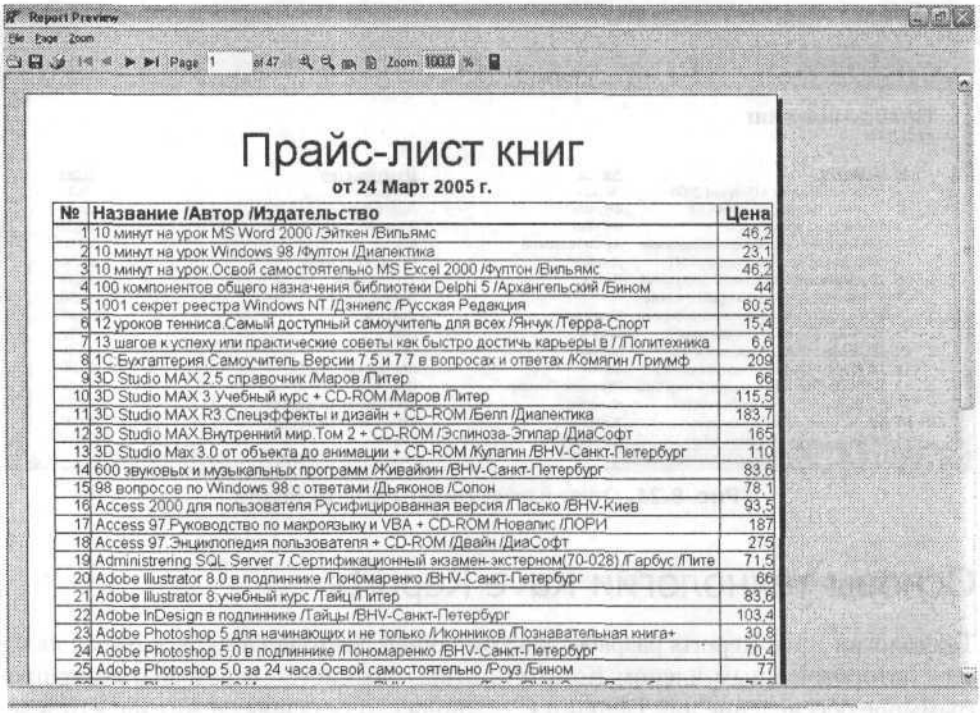


Рис. 8.22. Пример отчета в окне предварительного просмотра

2. Поместите на форму компонент `TTable` (категория **BDE**), в списке его свойства `DatabaseName` выберите локальный псевдоним `AAA`, а в списке свойства `TableName` — таблицу `BOOKS`. Назовите компонент именем `Books` (свойство `Name`). Создайте для набора данных 5 объектов-полей, связав их с полями `BookID`, `BName`, `BAuthor`, `BPublish` и `BPrice`. Для объекта `BName` напишите такой обработчик события `OnGetText`:

```
procedure TForm1.BooksBNameGetText(Sender: TField; var Text: String;
  DisplayText: Boolean);
begin
  Text := BooksBName.Value+' /'+BooksBAuthor.Value+
    ' /'+BooksBPublish.Value;
end;
```

3. Разместите на форме компонент `DataSource` (категория **Data Access**) и свяжите его свойство `DataSet` с набором данных `Books`. Откройте набор данных (`Active = True`).
4. Поместите на форму две панели `TPanel` (категория **Standard**). Установите для одной из них свойство `Align = alBottom`, для другой — `Align = alClient`. Очистите их свойства `Caption`.
5. На верхнюю панель поместите компонент `DBGrid` (категория **Data Controls**), установите в его свойство `Align` значение `alClient`, а в свойство `DataSource` —

значение DataSource1. Создайте для сетки два столбца, связав первый с полем BName, а второй — с полем BPrice. В свойство Title.Name первого поместите значение Название /Автор /Издательство, а в такое же свойство второго — значение Цена. Установите ширину первого столбца (свойство Width) равной 500.

6. На нижнюю панель поместите навигатор DBNavigator (категория Data Controls) и кнопку TButton (категория Standard). Свяжите навигатор с источником DataSource1 и оставьте в нем только первые 4 кнопки (в примере не предполагается изменение набора данных). В свойство Caption кнопки поместите значение Показать отчет.

   Вид формы на этом этапе показан на рис. 8.23.

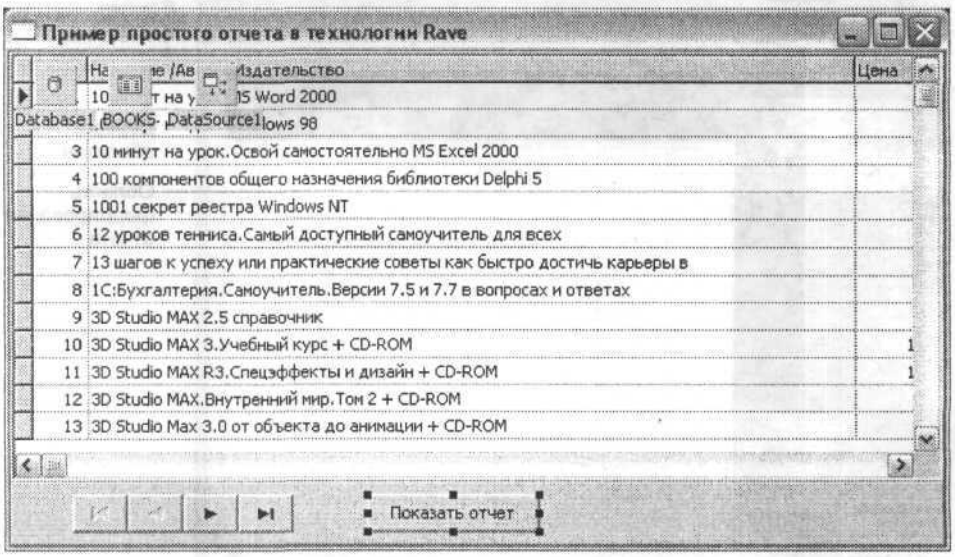


Рис. 8.23. Форма на этапе конструирования

7. Сохраните проект на жестком диске под именем RaveRepDemo.

   До этого момента мы создали лишь программу с набором данных. Теперь начинается собственно создание отчета.

8. Поместите на форму компонент TRVTableConnect (категория Rave) и свяжите его с набором данных Books (свойство Table).

9. Вся дальнейшая работа осуществляется под управлением утилиты Rave Reports Designer. Для ее вызова выберите в главном меню команду Tools ▸ Rave Reports Designer. На экране появится окно, показанное на рис. 8.24.

   Как видите, утилита Rave Reports Designer имеет собственную палитру компонентов, дерево объектов, инспектор объектов и набор инструментальных кнопок. Центральную часть окна занимает рабочая область с двумя вкладками: Page Designer и Event Editor. Первая используется подобно окну формы среды Delphi —

на ней автор отчета размещает нужные компоненты из палитры Rave Reports Designer (но никак не из палитры Delphi). На вкладке Event Editor он может написать код для обработчиков событий `OnBeforePrint`, `OnBeforeReport`, `OnAfterPrint`, `OnAfterReport`, `OnGetText` для любого размещенного на вкладке Page Designer компонента отчета. Обработчики пишутся на языке, который представляет собой подмножество языка Delphi.

Рис. 8.24. Окно Rave Reports Designer

10. С помощью команды File ▸ New Data Object вызовите окно Data Connections, в котором выберите продолжение Direct Data View и — после щелчка на кнопке Next — RvTableConnection1. Таким образом отчет связывается с источником данных, которые он будет отображать.
11. Командой Tools ▸ Report Wizards ▸ Simple Table вызовите эксперт создания отчета по данным, получаемым из единственной таблицы. Этот эксперт с помощью последовательно раскрывающихся окон позволяет задать основные параметры отчета. В первом окне (рис. 8.25) выбирается источник данных (`DataView1`), во втором — отображаемые поля (отметьте флажками все поля или установите флажок All). В следующем окне вы можете изменить порядок следования в отчете выбранных полей (в данном случае оставьте все без изменения). Окно Report Layout Options позволяет задать заголовок отчета (введите

в строку Report Title текст Прайс-лист книг) и ширину полей (группа строк Report Margin). Для экономии бумаги введите во все строки значение 0,3.

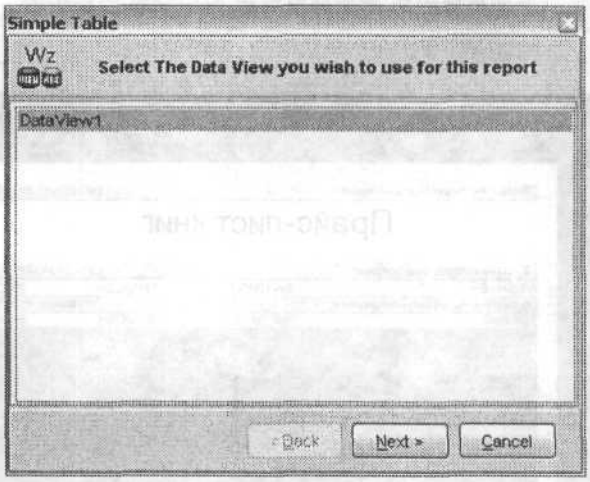


Рис. 8.25. Начальное окно эксперта создания отчета

Заключительное окно (рис. 8.26) предназначено для выбора шрифтов, которыми будут отображаться три основные части отчета: его заголовок, заголовки полей и данные.

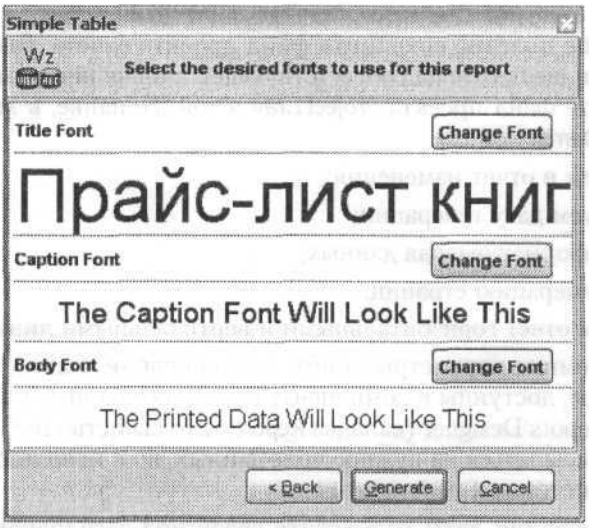


Рис. 8.26. Завершающее окно эксперта создания отчета

12. После щелчка на кнопке Generate в завершающем окне эксперта происходит начальная генерация файла проекта отчета, который появляется в окне утилиты Rave Designer (рис. 8.27).

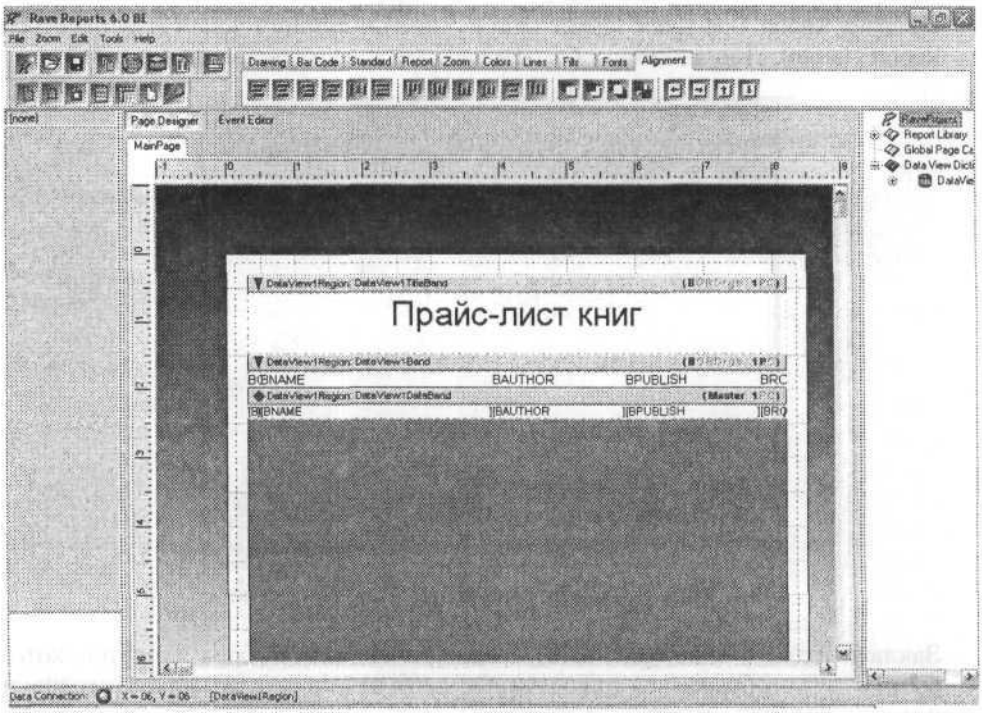


Рис. 8.27. Вид отчета после начальной генерации

13. На этом этапе полезно сохранить файл проекта отчета. Выберите команду File ▸ Save или щелкните на соответствующей кнопке инструментальной панели. Сохраните файл проекта Project1.rav в той же папке, в которой хранится проект RaveDemo.

    Теперь внесем в отчет изменения:

    - сформируем дату генерации;
    - изменим формат вывода данных;
    - введем нумерацию страниц;
    - расчертим отчет горизонтальными и вертикальными линиями.

    Многие системные параметры отчета, в том числе текущая дата и номер текущей страницы, доступны в компоненте `DataText` палитры компонентов утилиты Rave Repots Designer (вкладка Report). Его свойство `DataField` позволяет не только ссылаться на нужное поле данных, но и использовать выражения с участием системных переменных.

14. Для формирования даты разместите сразу под заголовком отчета компонент `DataText` (вкладка Report палитры компонентов утилиты Rave Reports Designer) и установите для него следующие свойства:

    - `Left` = 0;
    - `Width` = 7,9;

- `Top = 0,6;`
- `FontJustify = pjCenter;`
- `Font = Arial, 14, Bold.`

ПРИМЕЧАНИЕ

В Rave Reports Designer координаты задаются в дюймах. При вводе вещественных чисел знак разделения дробной и целой частей зависит от локализации. Для русифицированной версии Windows таким разделителем является запятая, для англоязычной (панъевропейской) — точка.

15. В свойство `DataField` поместите следующий текст:

```
"от "&Report.DateLong
```

Как видите, вместо имени поля в свойство можно помещать произвольное выражение с участием системных переменных (в нашем случае — с переменной `Report.DateLong`, содержащей текущую дату в полном формате `ДД Месяц ГГГГ`). Другой вариант формирования значения этого поля: раскройте редактор поля (кнопка с многоточием в правом углу значения свойства), в поле Data Text окна Data Text Editor введите `'от ' +`, раскройте список Report Variables и выберите в нем пункт DateLong, после чего щелкните на кнопке Insert Report Var (рис. 8.28).

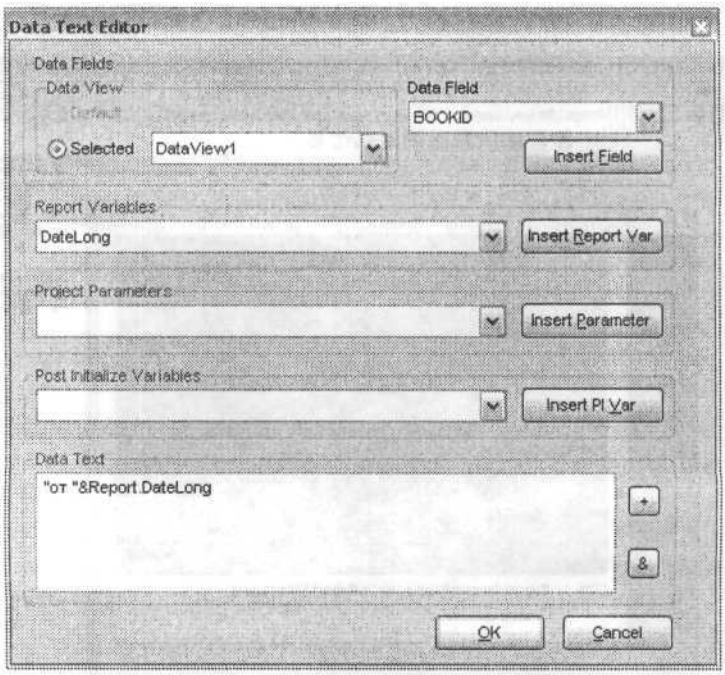


Рис. 8.28. Формирование значения поля DataField с помощью редактора

Заметьте: редактор выражений поддерживает как синтаксис C# (`"от "&`), так и синтаксис Delphi (`'от '+`).

16. Для изменения формата отображаемых данных удалите с нижней полосы компоненты DataText3 и DataText4 для отображения полей BAuthor и BPublish, а в свойство DataField компонента DataText2 (для поля BName) поместите такую строку: BName+' /'+BAuthor+' /'+BPublish. Установите для этого компонента свойства Left = 0,39 и Width = 7. Для компонента DataText1 (поле BookId) — свойства Left = 0, Width = 0,3. Для компонента DataText5 (поле BPrice) — свойства Left = 7,4, Width = 0,5. В свойства FontJustify обоих компонентов поместите значение pjRight.
17. Соответственно измените названия колонок в средней полосе: № вместо BookId, Название /Автор /Издательство вместо BName и Цена вместо BPrice.
18. Чтобы вставить в отчет номера страниц, нужно предусмотреть в проекте страницы пространство для размещения компонента DataText. Эксперт создания отчета отвел для заголовков и данных всю страницу. Щелкните на компоненте DataView1Region в дереве объектов (предварительно раскройте узлы ReportLibrary, Report1, MainPage) и установите в его свойство Height значение 9,7. С помощью вертикальной полосы прокрутки сместите изображение страницы в окне Page Design к самому ее концу и вставьте под серым пространством компонент DataText (рис. 8.29). В его свойство DataField поместите такую строку:

```
'Стр. '+Report.CurrentPage+' из '+Report.TotalPages'
```

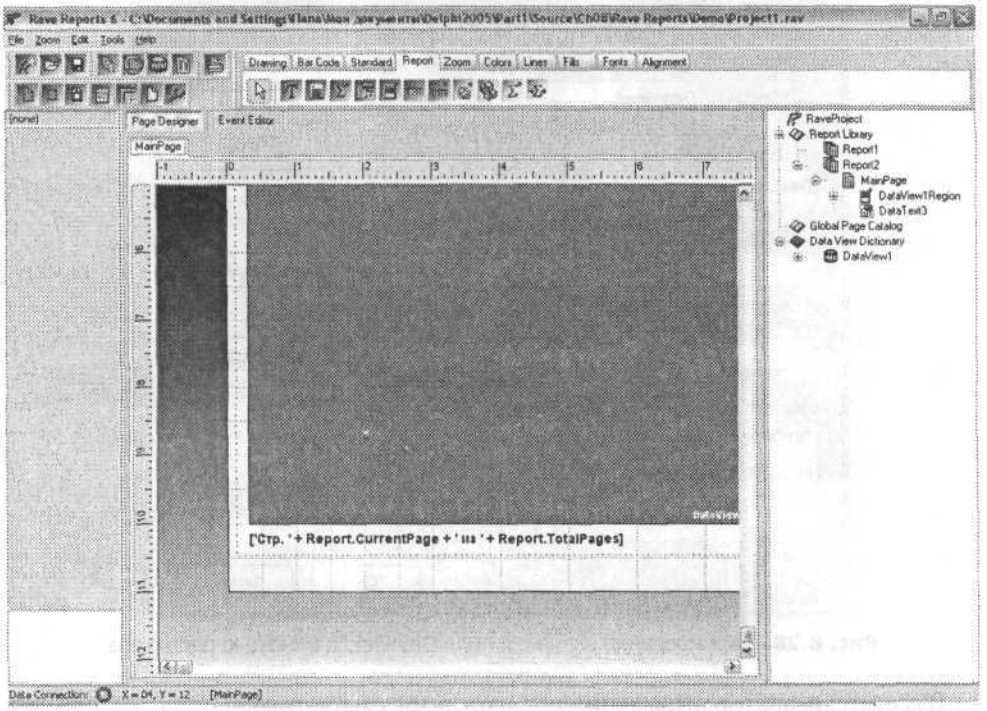


Рис. 8.29. Нумерация страниц отчета

19. Установите для компонента шрифт Arial высотой 14 и начертанием Bold.
20. Чтобы расчертить отчет линиями, поместите на среднюю полосу (объект DataView1Band в дереве объектов) компонент HLine (вкладка Drawing) со свойствами Top = 0, Left = 0, Width = 7,9, а также два компонента VLine с параметрами Top = 0, Height = 0,24, Left = 0,35 (для первого) и Left = 7,4 (для второго).
21. На нижнюю полосу с данными поместите по два компонента HLine и VLine. Длина горизонтальных линий равна ширине проекта отчета (7,9 единицы), а свойство Top равно 0 для первой и 0,2 для второй. Высота вертикальных линий равна высоте полосы (0,2), а свойство Left равно 0,35 для первой и 7,4 для второй.
22. Для предварительного просмотра формируемого отчета нажмите клавишу F9 или щелкните на соответствующей инструментальной кнопке.
23. После подготовки проекта отчета сохраните его на диске, закройте окно утилиты Rave Designer и вернитесь в Delphi. Поместите на форму компонент RvProject, в его свойстве ProjectFile укажите ссылку на файл проекта Project1.rav. Напишите такой обработчик щелчка на кнопке Показать отчет:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RvProject1.Execute
end;
```

Привязка проекта отчета к приложению

Имеет смысл обсудить вот какую проблему. Для организации связи приложения с проектом отчета имя файла отчета помещается в свойство ProjectFile компонента TRvProject. Это делает программу чувствительной к положению ее файла на диске или к изменению в ходе ее работы текущей папки. В самом деле, в свойство ProjectFile можно поместить полное имя файла с маршрутом доступа или только имя файла. В первом случае программа будет зависеть от положения файла проекта, что затруднит ее тиражирование или продажу. Во втором случае программа и файл проекта просто должны быть в одной папке, и эта папка должна быть текущей. Это снимает проблему тиражирования, но если в ходе работы программы произойдет смена текущей папки, связь программы с проектом окажется потерянной. Выходом из положения может быть указание в свойстве ProjectFile только имени файла и дополнение этого имени полным маршрутом доступа в момент старта программы. Например:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  S: String;
begin
  S := Application.ExeName;
  while S[Length(S)]<>'\' do
    Delete(S, Length(S), 1);
  rvProject.ProjectFile := S+rvProject.ProjectFile
end;
```

Визуальная среда Rave Reports Designer

Как уже говорилось, для технологии Rave Reports характерно то, что отчет создается специальной машиной генерации отчета по указаниям, получаемым из файла проекта отчета. Файл проекта разрабатывается с помощью утилиты Rave Reports Designer, которая создает особую визуальную среду (см. рис. 8.24).

С помощью главного меню решаются задачи общего управления проектом (команда File — создание нового проекта, включение в проект нового отчета, связь с наборами данных и т. д.), настройки параметров среды и проекта (команды Zoom и Edit ▸ Preferences) и вызова экспертов создания отчетов (команда Tools).

Некоторые наиболее важные команды представлены восемью верхними инструментальными кнопками. Семь нижних инструментальных кнопок упрощают настройку среды.

Окно инспектора объектов во многом похоже на окно инспектора объектов среды Delphi. В нем отображаются свойства объекта, выбранного в области формирования отчета или в дереве объектов. Замечу, что свойства, определяющие положение и размеры объекта, по умолчанию задаются в дюймах. При этом разделитель целой и дробной частей в вещественных значениях определяется настройкой Windows (в русскоязычной ОС этим разделителем является запятая). С помощью команды Edit ▸ Preferences ▸ Default ▸ Units можно установить другие единицы измерения (в том числе миллиметры или пикселы). Однако эта установка будет действительной только для нового проекта. Ниже окна свойств находится вспомогательное окно, содержащее справку о выбранном свойстве.

Справа от инструментальных кнопок располагается область компонентов и инструментальных панелей. Представленные здесь вкладки перечислены в табл. 8.1.

Таблица 8.1. Назначение вкладок области компонентов и инструментальных панелей

Вкладка	Назначение
Drawing	Компоненты для вставки в отчет простейших геометрических фигур
Bar Code	Компоненты для отображения различных штриховых кодов
Standard	Компоненты для вставки в отчет данных, не связанных с БД (поясняющие надписи, многострочный текст, изображения и т. п.)
Report	Компоненты для вставки в отчет данных из БД
Zoom	Инструментальная панель с кнопками, управляющими масштабом отображаемого отчета
Colors	Инструментальная панель для изменения цвета выделенного элемента отчета
Lines	Инструментальная панель для управления толщиной и стилем текущей линии
Fills	Инструментальная панель для выбора стиля заполнения текущей фигуры
Fonts	Инструментальная панель для выбора шрифта текущего элемента с текстом
Alignment	Инструментальная панель для управления положением выбранного элемента

Область формирования отчета занимает всю центральную часть окна. В ее верхней части находятся вкладки Page Designer и Event Editor, переключающие режимы отображения страницы отчета и ввода/редактирования обработчика того или иного события.

Режим Page Designer — основной режим окна. Отчет (точнее, его прообраз, представленный в режиме Page Designer) может содержать произвольное количество страниц, каждая из которых выбирается щелчком на соответствующей вкладке в верхней части окна. Прообраз страницы отображается в макете печатной страницы в центре окна. Программист формирует этот прообраз, размещая на макете компоненты из вкладок области компонентов. Он может в любой момент просмотреть или напечатать отчет, нажав клавишу F9 или щелкнув на соответствующей инструментальной кнопке.

С помощью команды Tools основную содержательную часть отчета можно сформировать, используя диалоговые окна соответствующих экспертов (для отчета с одной таблицей и отчета главный—детальный).

Расположенное в правой части окна утилиты Rave Reports Designer окно дерева объектов отображает основные объекты проекта отчета — отчеты (проект может содержать произвольное количество отчетов), глобальные страницы и объекты отображения данных из БД.

Режим Event Editor предназначен для создания/редактирования обработчиков событий `OnAfterPrint`, `OnAfterReport`, `OnBeforePrint`, `OnBeforeReport`, `OnGetText`. Язык программирования обработчиков является подмножеством языка Delphi. Если, например, на странице отчета расположить компонент `Text3`, то представленный ниже обработчик события `OnBeforePrint` этого компонента обеспечит нумерацию страниц отчета (начальное значение его свойства `Text` должно содержать символ 0):

```
Text3.Text := IntToStr(StrToInt(Text3.Text)+1);
```

Мои эксперименты с языком среды Rave Reports Designer (он не описан в документации) показали, что он имеет очень скромные возможности и рассчитан в основном на однострочные операторы присваивания, изменяющие значения того или иного свойства объекта. В нем нельзя использовать локальные переменные, блоки **`begin`**...**`end`**, **`try`**...**`end`**, условные операторы.

Составляющие проекта отчета

Любой проект отчета имеет три составные части:

- библиотеку отчетов (узел Report Library в дереве объектов);
- каталог глобальных страниц (узел Global Page Catalog);
- каталог объектов данных (узел Data View Dictionary).

Далее эти составляющие рассматриваются более подробно.

Библиотека отчетов

Библиотека отчетов хранит все созданные в проекте отчеты. Каждый отчет не зависит от других (эта связь подразумевается создателем проекта). В проекте

может быть сколько угодно отчетов. Чтобы начать очередной отчет в рамках текущего проекта, нужно выбрать команду File ▸ New Report или щелкнуть на соответствующей инструментальной кнопке.

Каждый отчет характеризуется своим уникальным именем (свойство Name) и/или полным именем (свойство FullName). В свойство Description следует поместить комментарий с краткой характеристикой отчета.

Первый созданный в проекте отчет становится умалчиваемым, то есть он печатается командой

```
RvProject1.Execute;
```

Получить список всех отчетов программа может с помощью такого метода компонента TRvProject:

```
procedure GetProjectList(ReportList: TStrings; FullName: Boolean);
```

Параметр FullName при обращении к этому методу должен содержать значение True, если программист хочет получить список всех значений свойства FullName, и False, если нужен список значений свойства Name. Напечатать любой отчет из этого списка можно с помощью такого метода:

```
procedure ExecuteReport(ReportName: String);
```

Для удаления ненужного отчета из библиотеки отчетов следует выбрать его в дереве объектов и нажать клавишу Delete.

Каждый отчет может содержать одну или несколько страниц. Для добавления к отчету новой страницы используйте команду File ▸ New Report Page или щелкните на соответствующей инструментальной кнопке. Чтобы удалить страницу из отчета, выберите ее в дереве объектов и нажмите клавишу Delete. Если на странице проекта расположена полоса с данными, при печати эта страница воспроизводится столько раз, сколько необходимо для полного отображения всех данных. Каждая страница характеризуется уникальным (в пределах отчета) именем (свойство Name), а также свойствами Orientation, PageHeight, PageWidth и PageSize (последнее определяет размер бумаги, например Letter 8 1/2 by 11-inch или A4 Sheet, 210 by 297-mm). Страницы по умолчанию печатаются в том порядке, в котором они представлены в дереве объектов. Свойство GotoPage определяет страницу, которая будет печататься после текущей. Изменение этого свойства позволяет печатать страницы в произвольном порядке.

Каталог глобальных страниц

Разработчик может создать одну или несколько глобальных страниц, то есть страниц, которые будут доступны из любого отчета. Таким способом можно оформить все отчеты единым образом, например, каждый отчет может предваряться печатью глобальной страницы с названием и эмблемой организации. Чтобы добавить к отчету глобальную страницу, нужно вызвать редактор его (отчета) свойства PageList (рис. 8.30), предварительно выделив отчет в библиотеке отчетов. В этом же редакторе можно изменить очередность печати страниц.

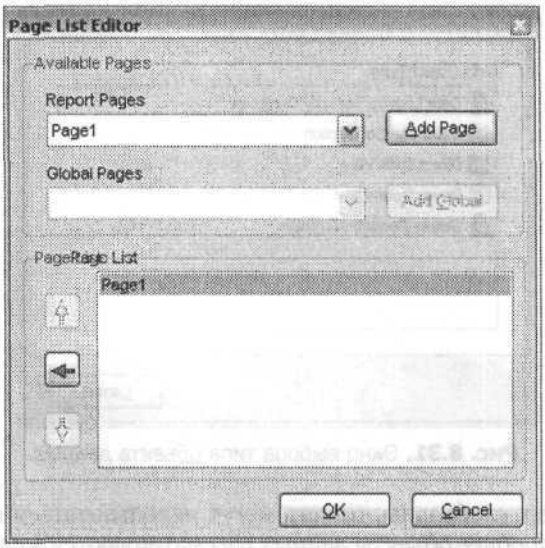


Рис. 8.30. Редактор свойства PageList отчета

Для создания глобальной страницы используется команда File ▸ New Global Page или соответствующая инструментальная кнопка. Чтобы отобразить глобальную страницу на вкладке Page Designer, нужно дважды щелкнуть на ее названии в дереве объектов. Таким способом можно выбирать не только глобальную, но и любую страницу любого отчета.

Каталог объектов данных

Каталог объектов данных содержит все определенные в проекте источники данных и, возможно, средства аутентификации пользователей для парольной защиты тех или иных отчетов. Для включения в проект нового объекта данных необходимо выбрать команду File ▸ New Data Object или щелкнуть на соответствующей инструментальной кнопке. В ответ появляется диалоговое окно, позволяющее уточнить тип объекта данных (рис. 8.31).

Вы можете выбрать один из следующих типов объектов данных:

- Data Lookup Security Controller — организует аутентификацию пользователя при попытке просмотра или печати защищенного источника данных;
- Database Connection — создает непосредственное соединение с источником данных на основе технологии dbGo.NET, BDE.NET, IBX.NET или dbExpress.NET с использованием специальных драйверов технологии Rave Reports;
- Direct Data View — создает соединение с набором данных, созданным в приложении Delphi;
- Driver Data View — создает соединение с ранее созданным в проекте объектом данных Database Connection;
- Simple Security Controller — организует список имен и паролей пользователей для доступа к отчету.

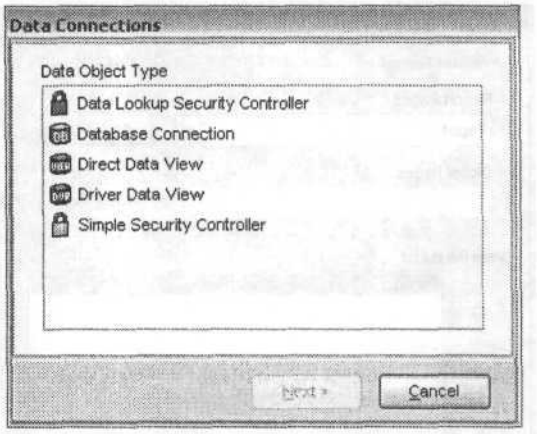


Рис. 8.31. Окно выбора типа объекта данных

Все объекты данных глобальны, то есть могут использоваться в любой странице любого отчета. Подробнее объекты данных рассматриваются в следующем разделе.

Объекты данных

Объектами данных называются различные объекты, связанные с импортом в проект внешних данных. Характерной особенностью технологии Rave Reports является возможность непосредственного (без использования Delphi) импорта данных из таблиц или запросов, а также импорта данных из произвольных файлов. Другая интересная особенность технологии — возможность парольной защиты публикуемых данных.

Доступ к наборам данных в приложении Delphi

Самый часто используемый объект данных — Direct Data View — обеспечивает импорт в отчет данных, полученных наборами данных (НД) приложения Delphi. Для взаимодействия отчета с НД в Delphi должны размещаться посредники — компоненты `TRvDataSetConnection` (универсальный связной компонент; в документации рекомендуется для клиентских НД в трехзвенной архитектуре), `TRvTableConnection` (для таблиц) и `TRvQueryConnection` (для запросов). Эти компоненты находятся в категории Rave среды Delphi. Каждый публикуемый в отчете НД должен снабжаться соответствующим посредником.

Для добавления в проект нового объекта нужно выбрать команду File ▸ New Data Object или щелкнуть на соответствующей инструментальной кнопке и затем в диалоговом окне выбрать вариант Direct Data View. После щелчка на кнопке Next этого окна появится новое окно, в котором предлагается выбрать один из определенных в приложении Delphi объектов-посредников.

Непосредственный доступ к данным

Технология Rave Reports позволяет создать отчет, не связанный с какими-либо наборами данных в приложении Delphi, а получающий эти данные непосредственно

из БД. Для этого предусмотрен специальный объект данных Database Connection. Этот объект может связываться с БД по технологии dbGo.NET (ADO), BDE.NET, dbExpress.NET (DBX) или IBX.NET.

Для реализации этой возможности нужно в окне Data Connections выбрать вариант Database Connection, а после щелчка на кнопке Next — одну из перечисленных технологий. Дальнейший диалог зависит от выбранной технологии доступа.

Если выбрана технология BDE, в новом диалоговом окне (рис. 8.32) нужно выбрать псевдоним БД (список Alias) и драйвер BDE для доступа к данным (список Driver).

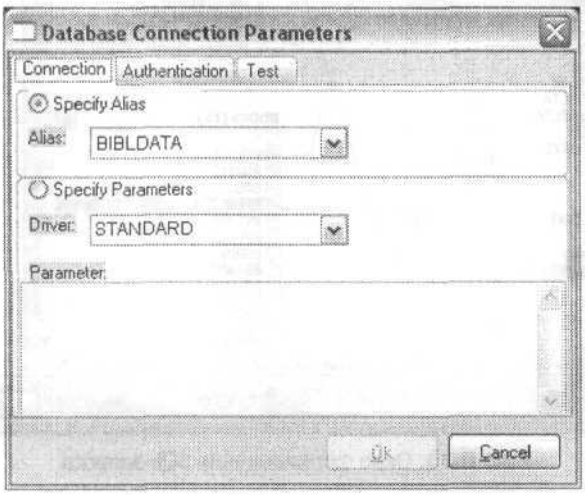


Рис. 8.32. Настройка соединения BDE

Диалоговые окна настройки соединения для технологий ADO, DBX и IBX описаны в главах 4, 5 и 6 соответственно.

Хотя объект Database Connection обеспечивает техническую связь отчета с данными, он не может поставлять эти данные отчету. Поставщиком данных является объект Driver Data View. После выбора этого объекта в окне Data Connections (см. рис. 8.31) в следующем диалоговом окне предлагается выбрать один из определенных в проекте объектов Database Connection, после чего появляется окно, показанное на рис. 8.33.

Это окно автоматизирует процесс формирования SQL-запроса к БД. В правой его части содержится список всех таблиц БД. Запрос формируется путем перетаскивания таблиц на вкладку Layout, где они отображаются в виде перечней всех своих полей. По умолчанию выбраны все поля (установлен флажок *). Установка флажка рядом с именем любого поля ведет к сбросу флажка * и позволяет выбрать только нужные поля. Для организации реляционных связей главный—детальный поле связи из детального набора перетаскивается на ключевое поле главной таблицы. Также устанавливается связь с подстановочной таблицей.

После создания всех необходимых связей щелкните на кнопке Editor, чтобы увидеть автоматически сгенерированный SQL-запрос (рис. 8.34).

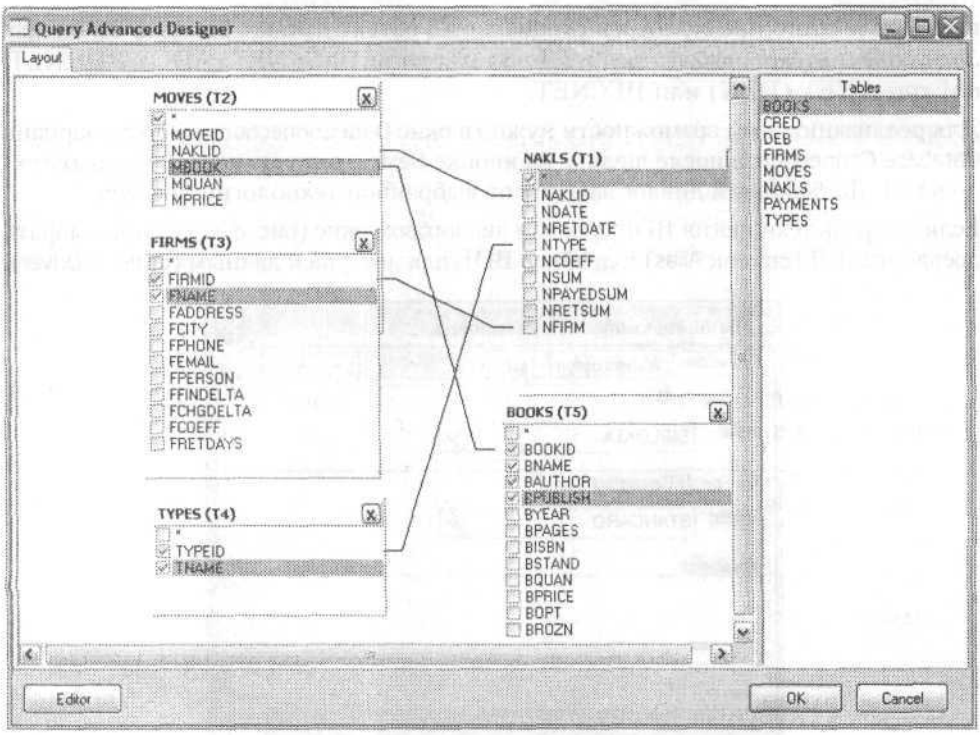


Рис. 8.33. Окно формирования SQL-запроса

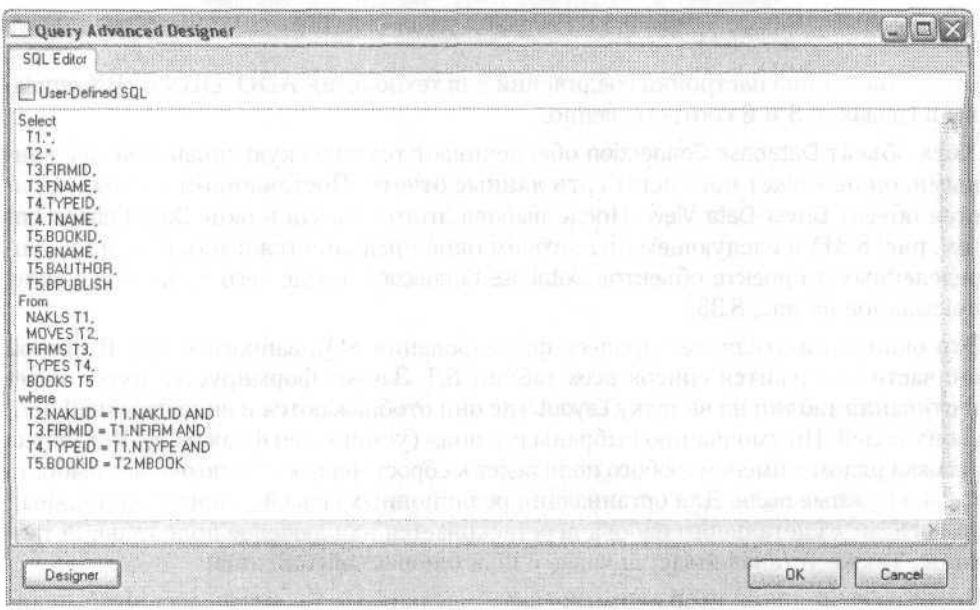


Рис. 8.34. Окно просмотра/редактирования запроса

При необходимости этот запрос можно редактировать. Например, добавить к нему секцию **ORDER BY** или **GROUP BY**.

Импорт в отчет произвольных внешних файлов

С помощью связного компонента TRvCustomConnection можно импортировать в отчет данные из внешнего файла. Характер данных, хранящихся в файле, может быть любым — лишь бы программист умел с ними работать.

Поскольку утилита Rave Designer не распознает формат файловых данных, основная работа возлагается на программиста. Он должен создать как минимум три обработчика событий компонента TRvCustomConnection:

- в обработчике события OnOpen необходимо определить общее количество строк отчета и поместить это число в свойство DataRows компонента;
- в обработчике события OnGetCols с помощью метода WriteField компонента определяются количество, тип и ширина всех полей отчета;
- в обработчике события OnGetRow программист должен передать в отчет значения всех полей для очередной строки отчета.

ВНИМАНИЕ

Поскольку метаданные отчета (количество, тип и ширина полей) определяются динамически, в момент создания проекта отчета эти обработчики должны быть определены, а приложение Delphi — запущено.

В следующем примере программа создает отчет на основе данных из произвольного текстового файла:

1. Начните новый отчет, поместите на пустую форму панель со свойством Align = alBottom, а на нее — две кнопки (Файл и Отчет). Для кнопки Отчет установите свойство Enabled = False. На форму выше панели поместите компонент TMemo с такими свойствами: Align = alClient, ScrollBars = ssBoth (рис. 8.35).
2. Поместите на форму компонент TOpenDialog, его свойство Filter сформируйте следующим образом:

```
PAS-файлы|*.pas|Текстовые файлы|*.txt|Файлы HTML|*.htm
```

3. Напишите такой обработчик щелчка на кнопке Файл:

```
procedure TForm1.Button1Click(Sender: TObject);
// Загрузка файла
begin
  if OpenDialog1.Execute then
  begin
    Memo.Lines.LoadFromFile(OpenDialog1.FileName);
    Button2.Enabled := True
  end
end;
```

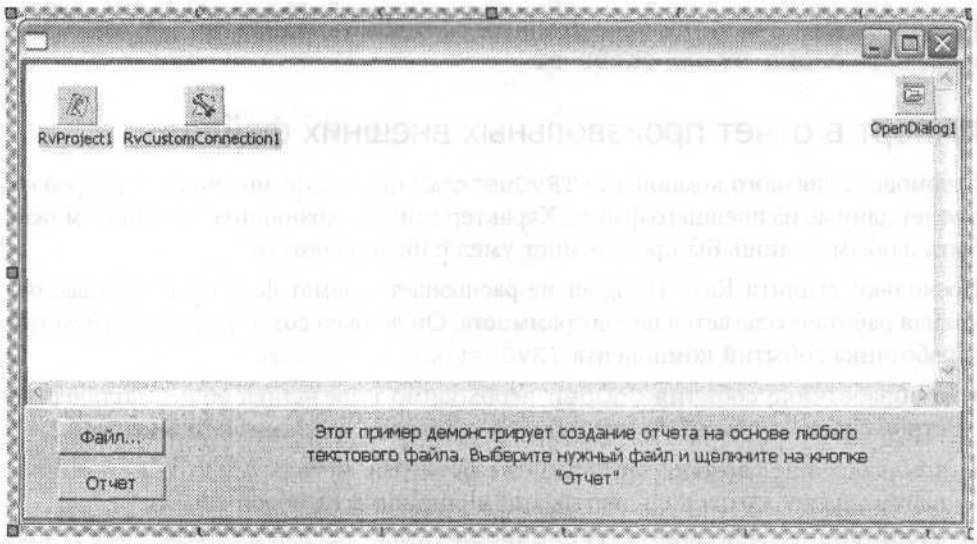


Рис. 8.35. Вид формы на этапе конструирования

4. Разместите на форме компоненты RvProject и RvCustomConnection. Для компонента RvCustomConnection1 поместите в свойство Name значение Connection.
5. Для компонента Connection напишите такие обработчики событий OnOpen, OnGetCols и OnGetRow:

```
procedure TForm1.ConnectionOpen(Connection: TRvCustomConnection);
// Определяет количество строк отчета и сбрасывает счетчик строк
begin
  Connection.DataRows := Memo.Lines.Count;
  k := 0
end;

procedure TForm1.ConnectionGetCols(Connection: TRvCustomConnection);
// Определяет единственное текстовое поле шириной 80 символов
begin
  Connection.WriteField('StrField', dtString, 80, 'StrField', '');
end;

procedure TForm1.ConnectionGetRow(Connection: TRvCustomConnection);
// Передает в отчет очередную строку
begin
  if Memo.Lines.Count >= k then
    Connection.WriteStrData('', Memo.Lines[k]);
    inc(k)
end;
```

6. Определите глобальную для обработчиков переменную k типа Integer. В обработчике ConnectionOpen указывается количество строк отчета, которое

равно количеству строк файла, загруженного в многострочное поле Memo1. С помощью метода WriteField в обработчике ConnectionGetCols определяется единственное поле отчета. При обращении к методу первым параметром передается имя поля (Name), вторым — тип поля, третьим — его длина, четвертым — полное имя (FullName) и последним — описание (Description). Обработчик ConnectionGetRow передает в отчет очередную строку файла. При обращении к методу WriteStrData первым параметром указывается форматирующая строка, а вторым — собственно текстовые данные.

7. Сохраните проект на диске, запустите его и вызовите утилиту Rave Designer.
8. Начните новый проект отчета и создайте в нем объект Direct Data View, связав его с компонентом Connection. Раскройте узлы Direct View Dictionary и DataView1 и убедитесь, что создано поле DataView1StrField. Если к моменту создания объекта Direct Data View приложение Delphi не будет запущено, Rave Designer не сможет определить количество и тип полей и создаст единственное абстрактное поле DataView1DataField. Если все сделано верно и определено строковое поле, закройте приложение Delphi и продолжите формирование отчета.
9. С помощью команды Tools ▸ Simple Table вызовите эксперт создания отчета по данным из единственной таблицы и с его помощью оформите заголовок отчета, заголовки полей и отображаемые данные. Сохраните отчет на диске и укажите ссылку на него в свойстве ProjectFile компонента RvProject1.

ПРИМЕЧАНИЕ

Вот типичный пример проблемы связи приложения с файлом проекта: при обращении программы к OpenDialog1 может произойти смена текущего каталога! Решение проблемы см. в подразделе «Привязка проекта отчета к приложению». См. также обработчик OnCreate главной формы примера (файл Chap_08\CustomConnect\Main.pas).

10. Для кнопки Отчет напишите такой обработчик:

```
procedure TForm1.btRunClick(Sender: TObject);
// Создает отчет
begin
  RvProject1.Execute
end;
```

Защита данных

Два объекта данных — Data Lookup Security Control и Simple Security Control — позволяют защитить проект в целом или какой-то его отчет от несанкционированного доступа. Различие в компонентах состоит в том, что первый считывает список имен пользователей и их паролей из таблицы БД, а второй имеет собственный список. Наличие этих объектов в проекте отчета лишь дает возможность проверить право пользователя на получение доступа к данным, однако собственно механизм защиты реализуется программно в приложении Delphi.

Компонент TRvProject имеет сложное свойство ProjMan (от *Project Manager* — менеджер проекта), в котором определено свойство SecurityControl как ссылка на один из определенных в проекте компонентов защиты — для защиты проекта

в целом, а также свойство ActiveProject.SecurityControl — для защиты на уровне отчета. Собственно контроль пользователя реализуется следующим методом любого из этих свойств:

```
function IsValidUser(UserName, Password): Boolean
```

Перед проверкой компонент TRvProject должен быть открыт. Вот пример реализации контроля на уровне проекта в целом:

```
procedure TForm1.btRunClick(Sender: TObject);
begin
  RvProject1.Open;
  if RvProject1.ProjMan.SecurityControl.IsValidUser
     (edUserName.Text, edPassword.Text) then
    RvProject1.Execute
  else
    ShowNessage('В доступе отказано');
  RvProject1.Close
end;
```

Для использования объекта Data Lookup Security Control его нужно связать с объектом типа Direct Data View и указать в свойстве UserField поле НД с именами пользователей, а в свойстве PasswordField — поле с паролями. Для объекта Simple Security Control следует заполнить его свойство UserList строками вида

```
имя_пользователя = пароль
```

Например:

```
User1 = Password1
Guest = Delphi
```

Типы отчетов

С помощью технологии Rave Reports можно создавать отчеты разных типов — с получением данных из единственного набора данных, из наборов данных, связанных отношением главный—детальный, с группировкой данных и т. д. Эти способы создания отчетов рассматриваются в этом разделе.

Отчет с единственной таблицей или запросом

Отчет с единственной таблицей (запросом) подробно описан в разделе «Пример создания отчета», где рассматривается общая технология создания подобных отчетов и свойства основных компонентов визуализации — областей и полос.

В общем виде процедура создания отчетов с одним источником данных выглядит следующим образом:

1. Создайте приложение Delphi, содержащее нужный НД (таблицу или запрос), компонент TRvProject и компонент TRvDataSetConnection. Вместо последнего можно использовать компонент TRvTableConnection или TRvQueryConnection в зависимости от типа НД. Во всех случаях компонент TRv*XXXX*Connection нужно связать с НД с помощью свойства DataSet, Table или Query.

ПРИМЕЧАНИЕ

Выбор нужного связного компонента зависит от используемой технологии доступа к данным. В технологиях dbExpress.NET, dbGO.NET, IBX.NET для связи с таблицами и запросами может использоваться только компонент `TRvDataSetConnection`, а компоненты `TRvTableConnection` и `TRvQueryConnection` работают лишь в технологии BDE.NET.

2. Вызовите утилиту Rave Reports Designer, начните новый отчет (или проект отчета) и создайте в нем объект данных Direct Data View, связав его с компонентом `TRv`*XXXX*`Connection`.
3. Вызовите эксперт создания отчета (командой Tools ▸ Simple Table) и с его помощью завершите начальное формирование отчета.
4. Если вас не устраивает отчет, созданный экспертом, модифицируйте его — введите нумерацию страниц, включите графические компоненты, например для вставки в страницы отчета эмблемы вашего предприятия, или создайте связанную с отчетом глобальную страницу и т. п.
5. Сохраните отчет в файле и вернитесь в среду Delphi.
6. Установите в свойство `ProjectFile` компонента `TRvProject` ссылку на файл проекта. Для вызова окна просмотра объекта и/или для печати отчета предусмотрите в программе соответствующий интерфейсный элемент (кнопку или команду меню). В обработчике связанного с ним действия укажите оператор `ProjectName.Execute`, если отчет единственный (умалчиваемый), или оператор `ProjectName.ExecutePerort('ReportName')`, если отчет не единственный в проекте и не умалчиваемый (`ProjectName` — имя компонента `TRvProject`, `ReportName` — имя проекта).

В результате работы эксперта будет создан отчет, содержащий единственную страницу `MainPage`, в которой имеется область просмотра данных `DataView1Region` с тремя полосами: `DataView1TitleBand`, `DataView1Band` и `DataView1DataBand`.

Две первые полосы являются полосами заголовка отчета и заголовков полей. Они различаются тем, что первая печатается только в начале отчета, а вторая — на каждой странице. Поэтому в редакторе важнейшего свойства полей `BandStyle`, определяющего положение полосы и ее наличие на странице при печати отчета, для первой установлены флажки Body Header и First, а для второй — еще и флажок New Page. Если этот флажок сбросить, полоса напечатается только на первой странице. На любом из этих полей можно размещать компоненты вкладки Standard палитры компонентов Rave Designer.

Полоса данных `DataView1DataBand` повторяется столько раз, сколько записей содержит соответствующий НД. Если после полосы данных расположить еще одну текстовую полосу (например, для нумерации страниц), она будет напечатана только в конце отчета. Чтобы вставить внизу или вверху каждой печатаемой страницы элементы оформления, нужно располагать эти элементы вне области просмотра данных. При этом, возможно, понадобится соответствующим образом уменьшить вертикальный размер области.

Если вы не хотите пользоваться услугами эксперта создания простого отчета, последовательность формирования отчета должна быть такой:

1. Создайте в отчете связанный с данными объект отображения Direct Data View (если отчет связан с НД из приложения Delphi) или Driver Data View (отчет получает данные без помощи приложения Delphi).
2. На пустую страницу поместите надписи, графические изображения и другие элементы оформления с вкладок Drawing, Bar Code и Standard окна Rave Reports Designer, которые должны печататься на каждой странице.
3. На страницу поместите компонент `Region` (вкладка Report). Его размеры и положение на странице должны быть такими, чтобы не закрывать оформительские элементы вверху, внизу или по бокам страницы.
4. Поместите на страницу полосы `Band` (вкладка Report) для заголовка отчета и названий полей. Каждая помещаемая в область заголовка полоса занимает весь ее горизонтальный размер. Для полосы заголовка отчета в редакторе свойства `BandStyle` должны быть установлены флажки Body Header и First, для полосы названий полей — флажки Body Header, First и New Page.
5. За полосами заголовка на страницу поместите полосу `DataBand` (вкладка Report). Своим свойством `DataView` она связывается с объектом отображения данных.
6. На полосе `DataBand` расположите компоненты для отображения данных из отдельных полей НД. Каждый компонент через свойство `DataView` свяжите с объектом отображения данных, а в его свойстве `DataField` укажите отображаемое компонентом поле данных или выражение с участием полей и системных переменных.

Отчет главный—детальный

Отчет главный—детальный воспроизводит данные из одной записи главной таблицы и все записи связанной с ней дочерней таблицы. Например, таким отчетом будет накладная на отпуск книг с указанием реквизитов покупателя и перечня проданных ему книг.

Для подготовки отчетов главный—детальный в Rave Reports Designer предусмотрен эксперт, который вызывается командой Tools ▸ Master/Detail Report. Перед обращением к эксперту выполните следующие действия:

1. В приложении Delphi создайте два НД — главный и детальный. Между ними можно не устанавливать соответствующую реляционную связь, так как эта связь будет создана в проекте отчета.
2. Поместите в приложение два связных компонента `TRvDataSetConnection`, связав их с наборами данных. Как уже говорилось, если в приложении используется технология BDE.NET, допустимо применение компонентов `TRvTableConnection` и `TRvQueryConnection`. Но если детальный НД создан на базе параметрического запроса, то и в этом случае для связи с ним требуется компонент `TRvDataSetConnection`, который, таким образом, является универсальным связным компонентом.

3. Создайте в проекте отчета два объекта данных типа Direct Data View, которые используют соответствующие связные компоненты, и обратитесь к услугам эксперта создания отчета главный—детальный.

   Пример отчета главный—детальный показан на рис. 8.36.

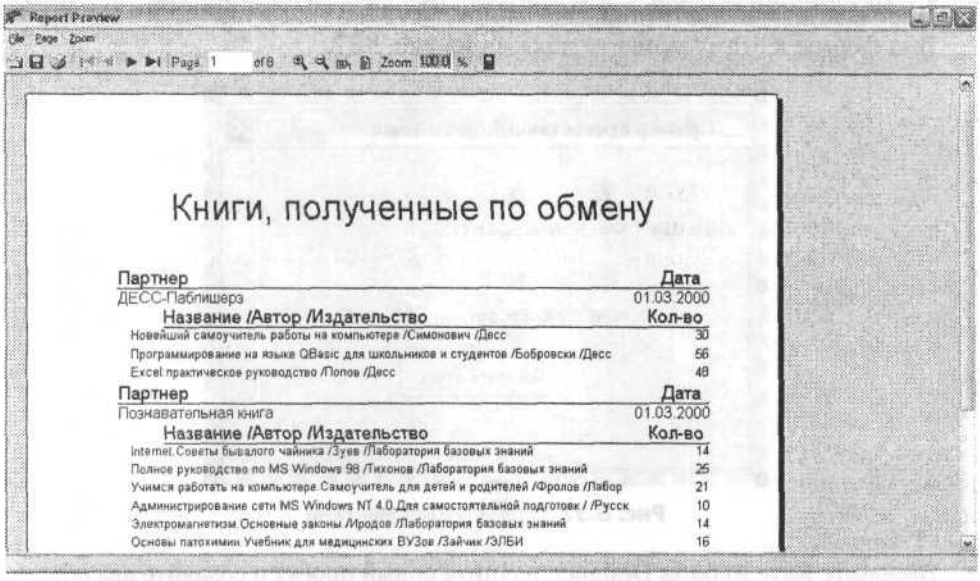


Рис. 8.36. Вид отчета главный—детальный в окне предварительного просмотра

4. Для создания этого отчета начните новое VCL-приложение и на пустую форму поместите компонент `TDatabase`, два компонента `TQuery` (категория BDE), два компонента `TRvDataSetConnection` (категория Rave) и кнопку (категория Standard). Настройте компонент `Database1` на связь с БД IB_BIBL.GDB.
5. Свяжите компонент `Query1` с локальным псевдонимом, созданным компонентом `Database1`, назовите его `NAKLS` и разместите в его свойстве `SQL` такой запрос:

```
SELECT
  NAKLID, FNAME, NDATE
FROM
  NAKLS, FIRMS
WHERE
  FIRMID = NFIRM AND NTYPE = 4
```

6. Компонент `Query2` также свяжите с локальным псевдонимом, назовите его `MOVES` и разместите в нем такой запрос:

```
SELECT
  NAKLID, BNAME, BAUTHOR, BPUBLISH, MQUAN
FROM
  MOVES, BOOKS
WHERE
  BOOKID = MBOOK
```

Как видите, детальный НД создан обычным (не параметрическим) запросом. Таким образом, реляционная связь в приложении не создается — она будет создана в отчете.

7. Один компонент `TRvDataSetConnection` свяжите с НД `NAKLS` и назовите его `NAKLSConn`, второй — с НД `MOVES`, назовите его `MOVESConn`.

   Вид формы к этому моменту показан на рис. 8.37.

Рис. 8.37. Форма с компонентами

8. Вызовите Rave Reports Designer, начните новый проект и создайте два объекта Direct Data View: первый для компонента `NAKLSConn`, второй для `MOVESConn`.
9. С помощью команды Tools ▸ Report Wizards ▸ Master/Detail Report вызовите эксперт создания отчета главный—детальный. Работа с экспертом происходит в последовательно раскрывающихся окнах. В первом окне эксперт предлагает выбрать главный НД (рис. 8.38).

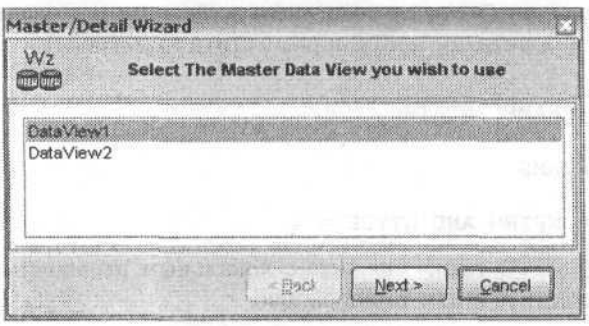


Рис. 8.38. Выбор главного НД

10. Щелчком выберите набор `DataView1` и щелкните на кнопке Next. Во втором окне предлагается выбрать детальный НД — выберите набор `DataView2`. В третьем окне (рис. 8.39) можно отобрать поля главного НД для показа в отчете.

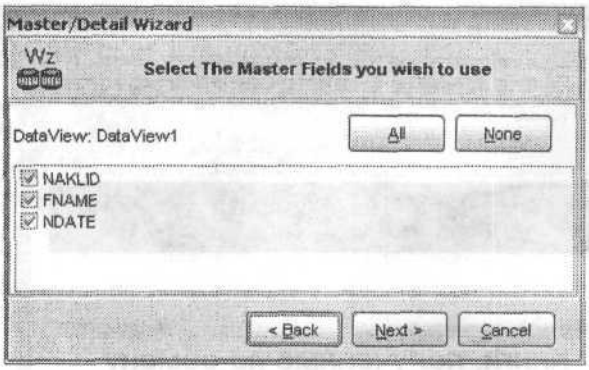


Рис. 8.39. Выбор полей главного НД

11. Щелкните на кнопке All, затем — на кнопке Next. В очередном окне эксперт попросит указать порядок отображения полей. Согласитесь с умалчиваемым. Два следующих окна позволяют указать поля и порядок их следования в детальном НД. Выберите все поля и умалчиваемый порядок. Наконец, в седьмом окне предлагается выбрать ключевые поля (рис. 8.40).

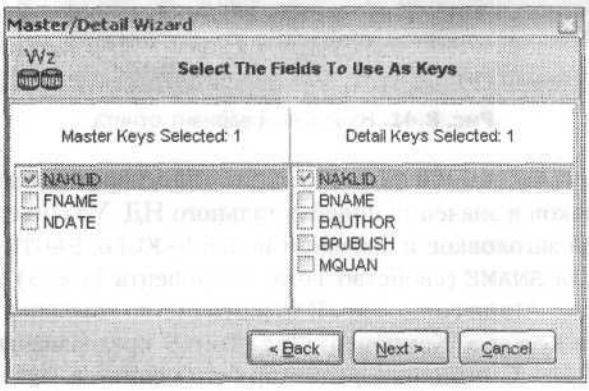


Рис. 8.40. Установление реляционной связи

12. Установите флажки рядом с полями NAKLID обоих НД и щелкните на кнопке Next. В следующем окне в строке Report Title введите текст Книги, полученные по обмену и установите ширину полей отчета в строках Left и Right равными 0,3 дюйма. В заключительном окне щелкните на кнопке Generate. Начальный вариант проекта отчета готов (рис. 8.41).

Теперь доработаем отчет — изменим названия полей, удалим ненужные.

13. В полосах DataView1Band и DataView1DataBand удалите компоненты Text1 и DataText1, предназначенные для отображения заголовка и значения поля NAKLID главного НД. В свойстве Text компонента Text2 вместо значения FNAME введите значение Партнер. Измените заголовок поля NDATE на Дата.

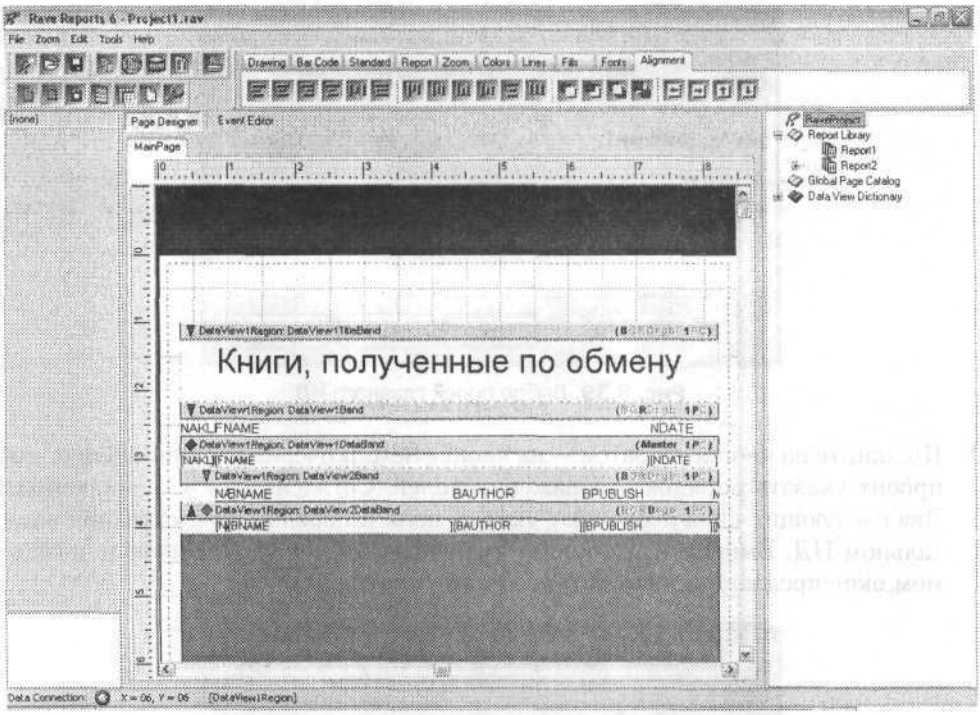


Рис. 8.41. Начальный вариант отчета

14. Полосы `DataView2Band` и `DataView2DataBand` предназначены для отображения заголовков и значений полей детального НД. Удалите в них компоненты для вывода заголовков и значений полей `NAKLID`, `BAUTHOR` и `BPUBLISH`. Заголовок поля `BNAME` (свойство `Text` компонента `Text5`) замените на `Название /Автор /Издательство`. В то же свойство компонента `Text8` поместите значение `Кол-во`. Компонент `DataText5` предназначен для отображения поля `BNAME`. С помощью редактора его свойства `DataField` введите такое выражение:

```
BNAME + ' /' +BAUTHOR + ' /' +BPUBLISH
```

15. Нажав клавишу F9, выполните пробную генерацию отчета. При необходимости измените высоту шрифта в компонентах, визуализирующих данные.
16. Сохраните проект отчета на диске и закройте окно утилиты Rave Reports Editor.
17. Поместите на форму компонент `TRvProject` (вкладка Rave). В его свойстве `ProjectFile` сошлитесь на файл проекта отчета. Напишите такой обработчик щелчка на кнопке:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RvProject1.Execute;
end;
```

Группирующий отчет

Группирующими называют отчеты, в которых вся информация разделяется на группы данных, объединенных каким-то общим признаком (например, остаток непроданных книг, поставленных конкретным поставщиком, или список книг, упорядоченный по издательствам).

В Rave Reports Designer нет эксперта создания группирующего отчета, так что мы познакомимся с технологией создания отчета «вручную».

Рассмотрим такой пример. Пусть необходимо получить список всех книг с группировкой по издательствам (рис. 8.42).

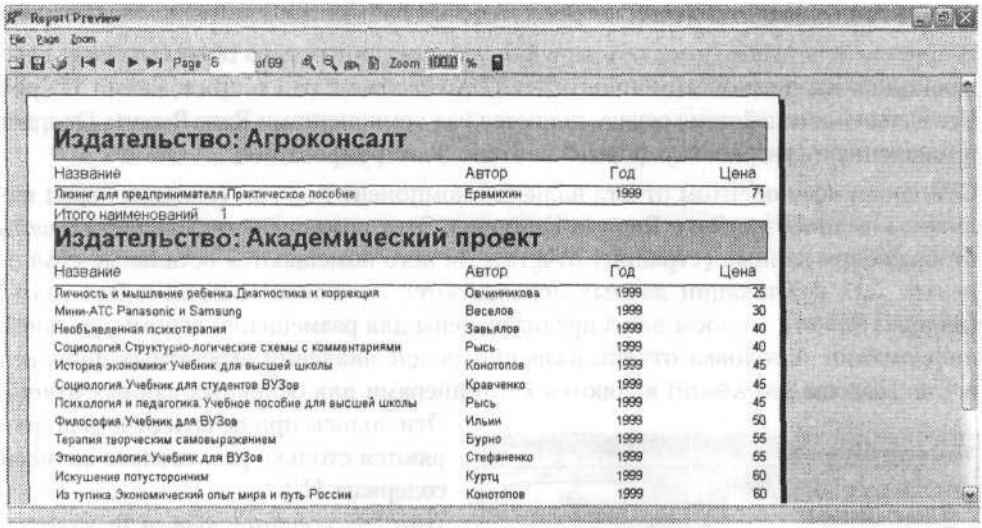


Рис. 8.42. Пример группирующего отчета

В этом случае в приложении Delphi должен быть компонент `TQuery` со следующим текстом запроса:

```
SELECT
  BName, BAuthor, BPublish, BYear, BOpt
FROM
  Books
GROUP BY
  BPublish, BName, BAuthor, BYear, BOpt
```

Другой вариант запроса может быть таким:

```
SELECT
  *
FROM
  Books
ORDER BY
  BPublish
```

Разница между вариантами незначительна. В первом случае информация упорядочивается (сортируется) по составному ключу, во втором — по какому-то одному полю. Напомню, что в запросах нельзя указывать индекс, по которому осуществляется доступ к данным. Если какая-то таблица имеет нужный индекс, он будет использоваться автоматически. Поэтому с точки зрения скорости доступа оба запроса приблизительно равноценны.

В обоих случаях группирующим является поле `BPublish`: изменение значения в нем воспринимается как свидетельство окончания очередной группы данных и *начало следующей*.

Для связи с НД типа `TQuery` можно использовать компонент `TRvQueryConnection` или `TRvDataSetConnection`.

В проекте отчета создается объект отображения данных типа Direct Data View, ссылающийся на связной компонент `TRvXXXXConnection` в приложении Delphi. Все дальнейшие действия осуществляются над компонентами Rave Reports Designer и напоминают разработку формы на этапе конструирования.

Основным компонентом отчета является компонент `Region` (вкладка Report палитры компонентов Rave Reports Designer). Этот компонент определяет область отображения данных (страницу отчета). На него помещаются остальные компоненты. Для публикации данных используются компоненты `Band` и `DataBand` (вкладка Report). Полосы `Band` предназначены для размещения вспомогательной информации: заголовка отчета, названий полей, значений агрегатных функций и т. д. Полосы `DataBand` являются контейнерами для основных данных отчета. Эти полосы при печати отчета повторяются столько раз, сколько записей содержит НД.

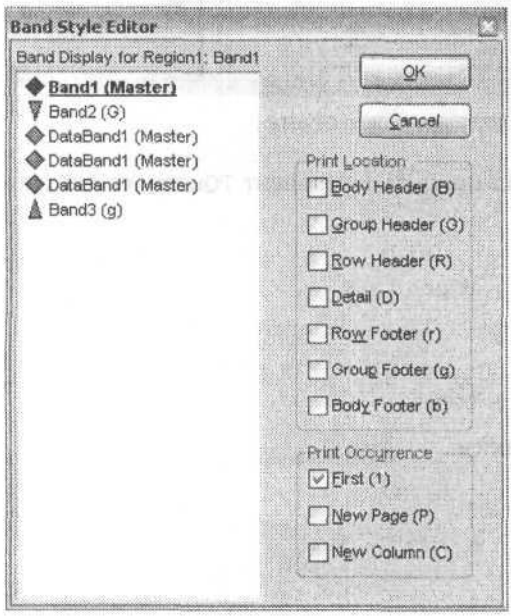


Рис. 8.43. Редактор свойства BandStyle

При размещении любой полосы на компоненте `Region` она автоматически растягивается по всей ширине компонента. Высота полосы может регулироваться программистом.

Основным свойством полос `Band` является свойство `BandStyle`. В его редакторе (рис. 8.43) имеется две группы флажков: Print Location и Print Occurrence.

Первая определяет положение полосы при печати, вторая — появление полосы в ходе печати. Назначение флажков группы Print Location (содержание полосы, если флажок установлен):

- Body Header — заголовок страницы;
- Group Header — заголовок группы;

- Row Header — заголовок данных;
- Detail —данные;
- Row Footer — подвал данных;
- Group Footer — подвал группы;
- Body Footer — подвал страницы.

Назначение флажков группы Print Occurrence (момент печати полосы, если флажок установлен):

- First — печатается первой;
- New Page — печатается в каждой новой странице;
- New Column — печатается в каждой новой колонке.

Замечу, что полоса не будет печататься, если не установлен хотя бы один флажок этой группы.

Спецификой группирующего отчета является то, что полосы с данными, относящимися к какой-то группе, должны указывать на объект отображения данных в своем свойстве `GroupView` и на поле группировки — в свойстве `GroupKey`. Поле группировки — это поле, изменение значения которого означает конец предыдущей группы и начало следующей. Обычно такое поле указывается первым в секции **`ORDER BY`** или **`GROUP BY`** SQL-запроса.

1. На пустой лист отчета поместите область отображения данных `Region` и установите нужные размеры этой области.
2. На область отображения данных поместите полосу `Band1` для заголовка отчета. Установите флажок First в ее свойстве `BandStyle`.
3. Ниже расположите три полосы для отображения заголовка данных (`Band2`), собственно сгруппированных данных (`DataBand1`) и итога отображения (`Band3`). Каждая из полос должна иметь ссылку на объект отображения данных в свойстве `GroupView` и группирующее поле (`BPublish`) в свойстве `GroupKey`. В редакторе свойства `BandStyle` полосы `Band2` установите флажки Group Header, First и New Page, последней полосы — флажок Group Footer. В редакторе свойства `BandStyle` средней полосы флажки в группе Print Location можно не устанавливать. Заголовочная и итоговая полосы должны ссылаться на полосу с группой данных в своем свойстве `ScrollerBand`. Чтобы это стало возможным, средняя полоса с данными должна быть компонентом `DataBand`. На рис. 8.44 показаны полосы группирующего отчета.

Использование агрегатных функций

Если вы внимательно посмотрите на рис. 8.44, то увидите в конце каждой группы итоговую строку с количеством наименований книг в группе. Такого рода данные получаются как результат обработки группы записей и формируются с помощью так называемых агрегатных функций. В Rave Reports предусмотрены стандартные для языка SQL92 агрегатные функции, перечисленные в табл. 8.2.

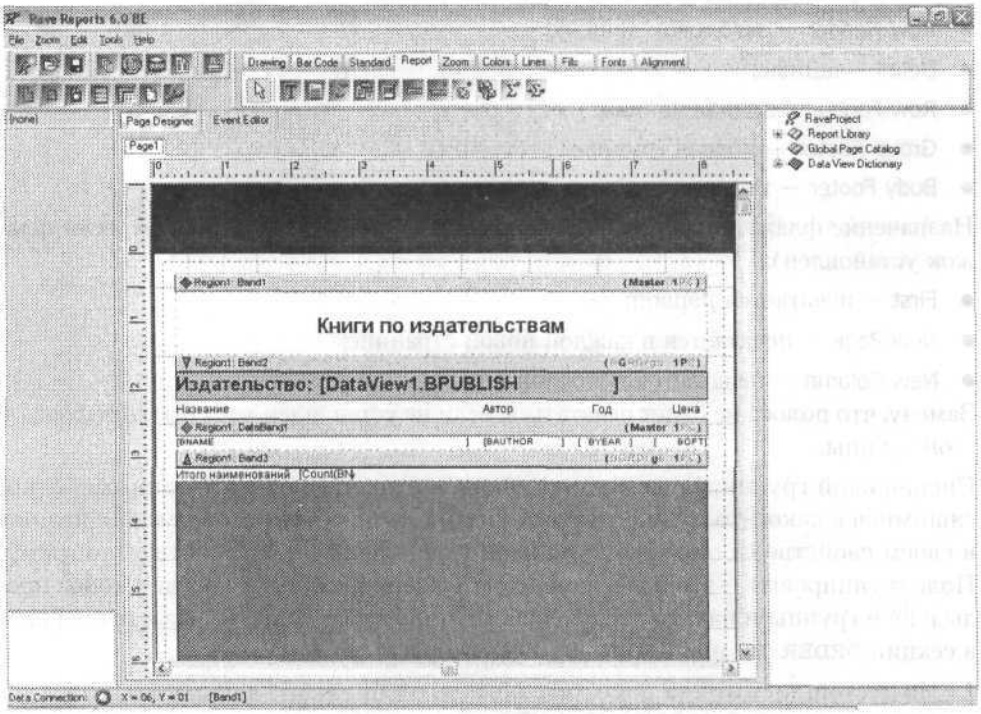


Рис. 8.44. *Полосы группирующего отчета*

Таблица 8.2. Агрегатные функции

Название функции	Назначение
AVERAGE (AVG)	Вычисление среднего значения из одного поля группы записей
COUNT	Подсчет количества значений поля
SUM	Суммирование значений поля для группы записей
MIN	Определение минимального значения поля
MAX	Определение максимального значения поля

Для получения в отчете результатов работы одной из этих функций предусмотрены два вычислительных компонента вкладки Report: `CalcText` и `CalcTotal`. Первый вычисляет и показывает значение агрегатной функции, второй лишь вычисляет значение и передает его другому вычислительному компоненту для его дальнейшего преобразования.

Оба компонента имеют свойства `DataView`, `DataField` и `CalcType`. Два первых определяют поле НД, значения которого будут использоваться при вычислении, третье определяет агрегатную функцию. Функция `COUNT` может быть применена к полю любого типа, остальные функции определены только для вещественных (целочисленных) полей.

Чтобы определить момент срабатывания вычислительного компонента, нужно каким-то образом пометить полосу отчета, в которой отображаются значения группы. Для этого используется специальный невидимый компонент `CalcController` (контроллер), который помещается на печатаемую группу. Ссылку на него следует указать в свойстве `Controller` вычислительного компонента. Другой вариант: сослаться в этом свойстве на саму полосу с данными. Печать полосы с контроллером (или полосы, указанной в свойстве `Controller`) вызывает срабатывание вычислительного компонента, то есть реализацию очередного цикла вычислений. В результате после завершения печати групповых данных вычислительный компонент накопит значение нужной агрегатной функции.

Поясним сказанное примером. Для оформления итоговой полосы отчета, показанного на рис. 8.44, на нее был помещен компонент `CalcText`. В его свойстве `DataView` указана ссылка на нужный объект отображения данных (`DataView1`). Он подсчитывает общее количество значений поля `BName`, поэтому его свойство `CalcType` имеет значение `ctCount`. В свойстве `Controller` компонента указана ссылка на саму полосу с данными `DataBand1`.

Экспорт отчета в файл

Экспорт отчета в файл того или иного формата необходим, когда созданный в программе отчет требуется переслать партнеру. Например, в работе оптового поставщика книг периодически создается и рассылается всем покупателям прайс-лист имеющихся на складе книг. В окне предварительного просмотра отчета по умолчанию можно сохранить отчет только в специфичном для технологии формате NDR или в виде последовательности поступающих на принтер команд в формате PRN. Ясно, что ни тот, ни другой формат нельзя считать приемлемым. (В самом деле, откуда у покупателя из Владивостока или Израиля возьмутся соответствующие средства просмотра?)

В технологии Rave Reports имеется возможность экспортировать отчет в файлы наиболее распространенных форматов — HTML, PDF (для просмотра с помощью бесплатно распространяемой утилиты Acrobat Reader корпорации Adobe Systems), RTF и TXT. Для этого в категории Rave предусмотрены компоненты `TRvRenderPDF`, `TRvRenderHTML` и т. д., которые нужно просто поместить на форму приложения. После этого в окне предварительного просмотра достаточно щелкнуть на кнопке сохранения файла и выбрать нужный формат в раскрывающемся списке Тип файла.

Практика показывает, что приемлемым форматом для русскоязычных отчетов является формат RTF, который позволяет получать прекрасные по качеству документы (рис. 8.45). Отчет, сохраненный в этом формате, можно просмотреть и напечатать с помощью текстового процессора MS Word или стандартной утилиты Windows WordPad (в этом случае качество гораздо хуже, а некоторые длинные файлы эта утилита не сможет отобразить).

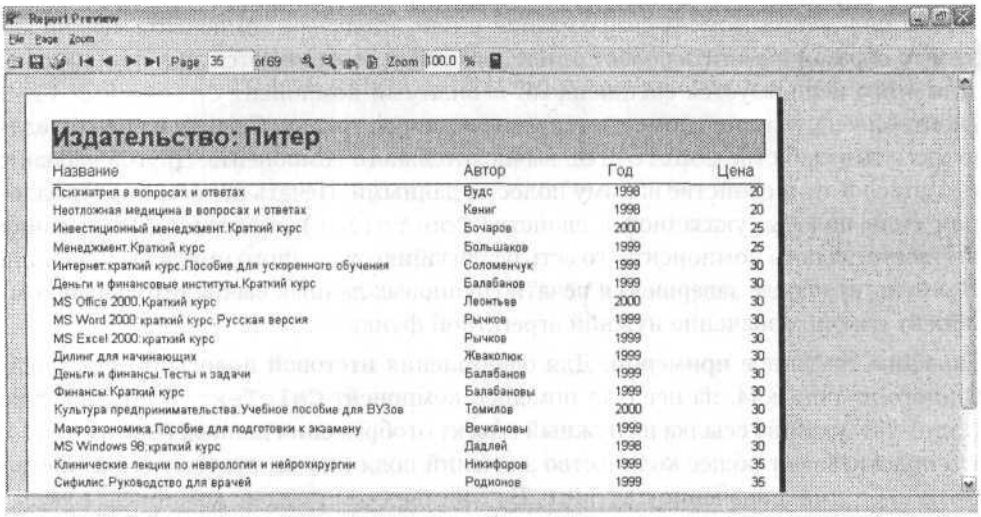


Рис. 8.45. Отчет в окне MS Word

Текстовый формат TXT (рис. 8.46) менее красив, но зато не искажает текст: если вы внимательно посмотрите на рис. 8.45, то заметите, что в названиях книг опущены некоторые слова.

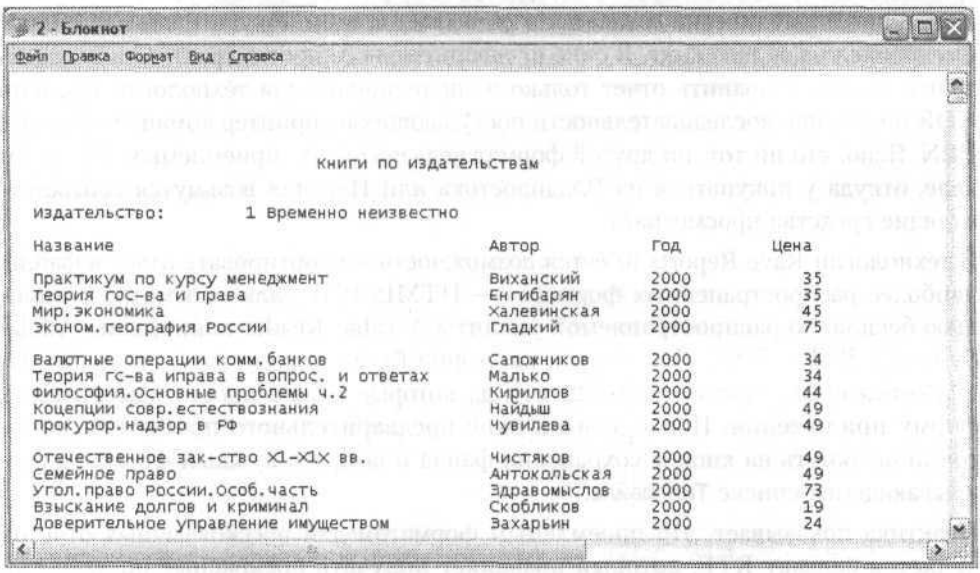
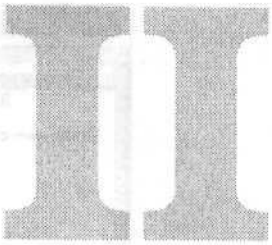


Рис. 8.46. Отчет в формате TXT

ВНИМАНИЕ

Форматы PDF и HTML искажают кириллицу.

Создание приложений для работы с Интернетом

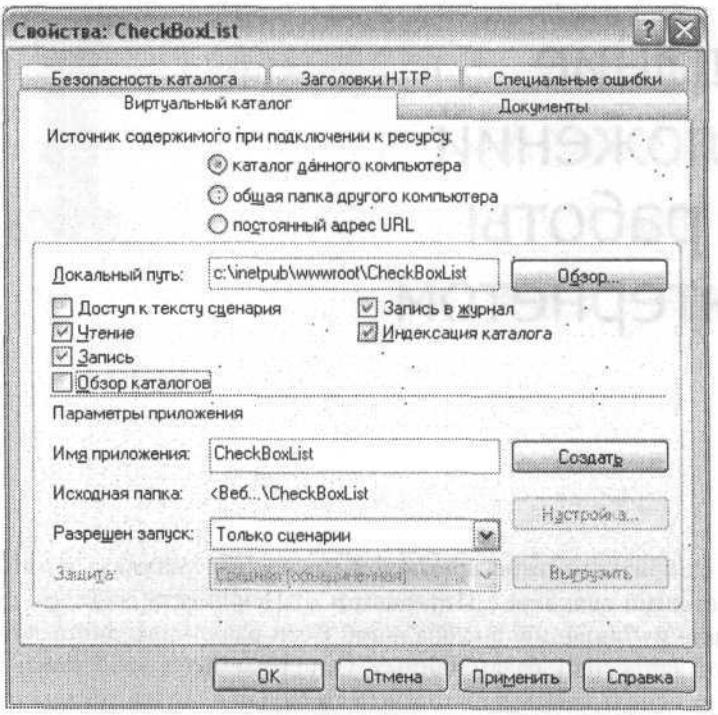


Без риска ошибиться можно предположить, что вы, уважаемый читатель, в той или иной степени знакомы с Интернетом. В Delphi есть средства, существенно упрощающие публикацию в глобальной Сети различных данных, в том числе результатов запросов к базам данных. Описываемые в этой части технологии Delphi могут использоваться не только в глобальной, но и в локальной сети предприятия (локальная сеть, в которой передача информации организована в соответствии с технологиями Интернета, называется *интранетом*).

Актуальность публикации именно баз данных связана с бурным ростом количества так называемых сетевых магазинов — сайтов, предлагающих те или иные товары и услуги. Содержимое таких сайтов должно создаваться на основе информации из реальной БД предприятия (магазина).

ПРИМЕЧАНИЕ

Исходные тексты описываемых в книге примеров размещены на сайте www.piter.com. Для использования текстов примеров, рассматриваемых в этой части книги, каждый из них следует сохранить в отдельном виртуальном каталоге, входящем в папку c:\inetpub\wwwroot (виртуальным называется каталог, доступный серверу MS IIS). После размещения файла примера в каталоге следует с помощью команды Пуск ▸ Панель управления ▸ Администрирование ▸ Internet Information Services операционной системы Windows запустить утилиту управления MS IIS, раскрыть узел Локальные веб-узлы, разыскать и раскрыть узел с примером, щелкнуть на раскрытом узле правой кнопкой мыши, выбрать в контекстном меню команду Свойства и на вкладке Виртуальный каталог открывшегося окна разрешить чтение, запись и запуск сценариев, как показано на следующем рисунке.



Установка прав доступа для виртуального каталога

Программирование для Интернета и технология ASP.NET

Технология ASP.NET (от англ. Active Server Pages — активные серверные страницы) является базовой технологией для работы с Интернетом на платформе .NET. В этой и последующих главах освещаются различные аспекты этой технологии.

Основы сетевого программирования

В этом разделе описываются базовые технологии, лежащие в основе Интернета. Если эти технологии вам известны, можете спокойно пропустить этот раздел.

Средства

Для работы с Интернетом ваш компьютер должен, прежде всего, иметь доступ к глобальной Сети. Кроме того, он должен содержать программные средства просмотра узлов Интернета, а если вы планируете не только потреблять информацию из Сети, но и поставлять в нее свою, на вашем компьютере должен быть установлен веб-сервер.

Важные средства — языки HTML и XML — описаны в приложении В.

Веб-сервер

Чтобы компьютер, владеющий данными, смог опубликовать их в сети (Интернет или интранет), он, во-первых, должен иметь постоянный доступ к этой сети. Поскольку обычная для пользователей связь с Интернетом через модем не обеспечивает нужной устойчивости и скорости обработки запросов, для этих целей создается выделенная линия (радио, оптоволоконная, спутниковая, телефонная) или используются специальные средства поставщика услуг Интернета. Во-вторых, на компьютере должен постоянно функционировать веб-сервер — программа, обслуживающая запросы клиентов.

Для серьезной корпоративной работы в сети обычно устанавливаются мощные серверы Microsoft Internet Information Services (IIS), Netscape FastTrack Server или Apache.

ПРИМЕЧАНИЕ

Так как описываемая далее базовая технология ASP.NET рассчитана на работу с IIS или с бесплатно распространяемым сервером Cassini Personal Web Server, один из этих серверов должен быть установлен на вашем компьютере или в доступном сетевом окружении. Мощный IIS-сервер поставляется со всеми версиями Windows, начиная с Windows 2000. Значительно более простой сервер Cassini доступен для бесплатной загрузки с узла Microsoft http://www.asp.net/Projects/Cassini/Download (объем — 217 Кбайт). В папке Samples/Cassini каталога установки Delphi есть пример использования этого сервера.

Назначение веб-сервера заключается в том, чтобы переадресовать запрос клиентского браузера нужному *веб-приложению*, то есть специальной программе, которая в ответ на запрос формирует текстовую страницу в формате HTML (или XML) и передает ее серверу, а тот посылает эту страницу клиенту. Веб-приложения часто называются *модулями расширения* сервера. Таким образом, для публикации данных нужно с помощью Delphi разработать необходимые веб-приложения и разместить их на машине веб-сервера в специально для этих целей предусмотренной папке wwwroot.

Браузер

HTML-браузер — это программа, преобразующая описание полученной от сервера страницы в ее видимое изображение. Фактически в мире доминирует браузер Internet Explorer (IE) корпорации Microsoft, входящий в поставку 32-разрядных версий Windows. Однако многие пользователи других платформ предпочитают Netscape Communicator (NC), первые версии которого назывались Netscape Navigator. Наконец, почти 1,5 миллиона компьютеров оснащены «третьим браузером для стран третьего мира» — речь идет о браузере Opera производства норвежской компании Opera Software.

Каждый браузер в общем случае интерпретирует описание страниц по-своему. Программируя для Интернета, вы постоянно должны помнить об этом.

Некоторые детали протокола HTTP

Протокол HTTP (HyperText Transfer Protocol — протокол передачи гипертекста) используется в Интернете для передачи HTML-страниц. Поскольку далее при рассмотрении компонентов Delphi используются термины этого протокола, в данном разделе кратко поясняются некоторые его технические подробности.

По своей сущности HTTP относится к протоколам пользовательского уровня, так как определяет набор правил для взаимодействия непосредственно с программой пользователя, игнорируя такие важные уровни, как транспортный (на этом уровне реализуются контроль и исправление всех ошибок передачи), сетевой (на нем данные разделяются на блоки и снабжаются маршрутами доставки) и самый нижний — физический, на котором данные собственно и передаются от клиента к серверу и обратно. Эти уровни реализуются протоколом TCP/IP.

Для взаимодействия с пользователем протокол предлагает 4 метода:

- HEAD — с помощью этого метода клиент получает от сервера информацию о передаваемых данных: их заголовок (часть между тегами HEAD) и дату последнего обновления. Метод HEAD обычно используется клиентом для проверки связей гипертекста на действительность и доступность.
- GET — этим методом клиент получает от сервера всю гипертекстовую страницу, включая ее заголовок, или передает серверу параметры заполненной диалоговой формы через переменные окружения операционной системы.
- POST — с помощью этого метода клиент передает серверу параметры заполненной диалоговой формы через стандартный поток ввода-вывода (поток данных).
- PUT — предписывает серверу заменить старое значение страницы новым. Таким методом обновляются удаленные сайты (и «взламываются» чужие!).

В подавляющем большинстве случаев клиент использует методы HEAD и GET, поэтому в запросе не присутствует дополнительная информация. Метод PUT обычно требует идентификации клиента, чтобы выяснить доступность метода. В этом случае в запросе указывается дополнительная информация, позволяющая отсеять нелегальных пользователей.

Протокол HTTP подразумевает обмен данными между браузером и веб-сервером с помощью *пакетов сообщений*. Различают два типа пакетов: пакет запроса и пакет ответа. Любой пакет состоит из трех частей:

- строки запроса (пакет запроса) или строки состояния (пакет ответа);
- заголовка HTTP-пакета;
- тела пакета.

Строка запроса состоит из указания метода передачи данных, маршрута доступа к запрашиваемому ресурсу и номера протокола HTPP. Например:

```
GET /web/index.htm HTTP/1.1
```

В этом примере используется метод GET, означающий, что клиент хочет что-то получить от сервера. Что именно — указано во второй части строки (`/web/index.htm`). Наконец, в третьей части строки запроса указывается номер версии протокола (`HTTP/1.1`).

Строка состояния также состоит из трех частей: версии протокола HTTP, кода состояния и текстовой строки, описывающей код состояния. Например:

```
HTTP/1.1 200 OK
```

Код состояния 200, описанный как OK, означает успех выполнения. Другие коды состояния классифицированы по первой цифре кода следующим образом:

- 1xx — информация;
- 2xx — успех;
- 3xx — переадресация;
- 4xx — клиентская ошибка;
- 5xx — ошибка сервера.

Заголовок HTTP-пакета содержит информацию о запросе или ответе. Строка заголовка состоит из пар «имя—значение». Существует четыре формы заголовков: *общий* (general), *запрос* (request), *ответ* (response) и *элемент* (entity). Общие заголовки содержат информацию о сервере или клиенте. Заголовки ответа содержат информацию о сервере, которая может быть полезна клиенту. Элемент содержит информацию о передаваемых данных. Более подробная информация о разных типах заголовков содержится в документе RFC2068, доступном по адресу www.w3c.org.

Наконец, тело пакета в случае запроса обычно отсутствует, а в случае ответа содержит HTML-текст.

Общая схема обработки запроса клиента

Структура URL

Пользователь оформляет свой запрос в виде URL-адреса. URL (Uniform Resource Locator — унифицированный указатель ресурса) — это цепочка символов, однозначно определяющая положение некоторого ресурса в сети и запрос клиента к этому ресурсу. В качестве примера рассмотрим следующий URL:

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

В этом примере URL состоит из таких частей:

- `http` — протокол передачи данных; здесь допускается указание протоколов `https` (secure HTTP — протокол HTTP повышенной секретности), `ftp` (File Transfer Protocol — протокол передачи файлов) и др.;
- `www.TSite.com` — сетевое имя машины, на которой расположен веб-сервер;
- `art/gallery.dll` — имя (возможно, с маршрутом доступа) веб-приложения, которому адресуется запрос клиента;
- `mammals` — условное имя запроса (одно и то же веб-приложение может обрабатывать несколько запросов клиента);
- `animal=dog&color=black` — параметры запроса (соседние параметры разделяются символом `&`).

Функционирование браузера при передаче запроса

Получив URL-адрес, браузер производит его предварительную обработку, дополняя информацией о методе протокола, параметрах самого браузера, операционном окружении и т. п. Например, рассмотренный выше URL-адрес может быть в конечном счете преобразован браузером в следующие строки:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/9.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

В первой строке помимо информации о запросе указываются также метод (GET) и протокол (HTTP/1.0), который собирается использовать браузер для связи с ресурсом.

Вторая строка указывает, что соединение с ресурсом будет активным до тех пор, пока ресурс не обработает клиентский запрос.

В третьей строке приводится информация о браузере клиента и операционном окружении.

В четвертой строке указываются сетевое имя машины, на которой располагается нужный ресурс, и ее порт связи.

В пятой строке перечисляются компоненты мультимедиа, которые веб-приложение может поместить в свой ответ.

Работа веб-сервера при обработке запроса

Получив запрос клиентского браузера, веб-сервер отыскивает нужное веб-приложение, активизирует его и передает ему параметры запроса. Сервер ожидает окончания работы веб-приложения и передает сформированный им ответ браузеру.

Введение в технологию ASP.NET

В технологии ASP.NET сервер работает особым образом. По расширению aspx он выясняет, что клиент запросил страницу ASP.NET. Затем он передает запрошенный файл выделенному серверу ISAPI — aspnet_isapi.dll. Последний посылает страницу рабочему процессу ASP.NET — aspnet_wp.exe. Рабочий процесс вызывает общеязыковую среду исполнения CLR, которая компилирует и выполняет код страницы. Результатом рабочего процесса является HTML-код, который и возвращается клиенту в теле ответа.

Назначение и архитектура технологии ASP.NET

Как следует из названия, технология ASP.NET предназначена для создания так называемых *активных серверных страниц*. Такие страницы содержат элементы управления и текст в формате HTML. Они размещаются на сервере и, таким образом, доступны в Интернете или в интранете. При помещении страницы в браузер пользователя последний может воздействовать на элементы управления (кнопки, списки и т. д.), чтобы добиться нужного результата.

Состав приложения ASP.NET

Каждое приложение ASP.NET размещается в отдельном виртуальном каталоге (см. примечание перед началом этой главы) и имеет несколько файлов и подкаталогов (вложенных папок).

Файлы с расширением aspx содержат веб-формы, которые могут загружаться в браузеры. В одном приложении может содержаться сколько угодно веб-форм. Каждая

форма обычно сопровождается pas-файлом, который содержит связанный с формой код Delphi. В этом файле чаще всего располагаются обработчики событий `Page_Load`, `OnInit`, а также обработчики щелчков на элементах управления формы. После компиляции pas-файла появляется файл с расширением dcuil.

Каждое приложение имеет два индивидуальных файла: Global.asax и Web.config. Первый содержит обработчики событий приложения в целом, а также объекты, доступные всем формам. С ним ассоциирован файл Global.pas, в котором объявляется класс `TGlobal` как наследник класса `System.Web.HttpApplication`, и имеются заготовки для обработчиков всех событий, касающихся приложения в целом. Файл Web.config содержит конфигурационную информацию. Он широко используется, например, для защиты приложений ASP.NET (см. главу 15).

Наконец, полноценное приложение ASP.NET в Delphi имеет файл проекта с расширением dpr и связанные с ним файлы bdsproj, cfg, dsk и пр. Замечу, что имя каждого из этих файлов совпадает с именем виртуального каталога. Файл проекта управляет компиляцией приложения. Результат компиляции всегда помещается во вложенную папку BIN в виде файла с именем проекта и расширением dll.

Веб-формы, компоненты управления сервером и HTML-компоненты

Основными компонентами архитектуры ASP.NET являются веб-формы, компоненты управления сервером и HTML-компоненты.

Веб-формы предназначены для размещения на них текста и компонентов управления сервером, которые предъявляются пользователю в окне его браузера. С каждой веб-формой связывается файл с расширением aspx, содержащий описание страницы на языке HTML, и файл с расширением pas, в котором описывается логика работы приложения на языке Delphi. Aspx-файл автоматически создается и редактируется на этапе разработки веб-формы. Файл с логикой работы компилируется в динамически загружаемую библиотеку DLL, которая вместе с aspx-файлом размещается на сервере IIS или Cassini.

Компоненты управления сервером представляют собой стандартные поля ввода, надписи, списки, кнопки и т. п., модифицированные для технологии ASP.NET. Их свойства, методы и события используются для организации логики работы приложения. Помимо стандартных интерфейсных компонентов, имеются компоненты для работы с базами данных. При взаимодействии пользователя с интерфейсными компонентами срабатывает связанная с ними программная логика, которая заставляет сервер переслать в браузер нужным образом измененную страницу.

HTML-компоненты предназначены для статического размещения текста и изображений. Обычно они не взаимодействуют с программной логикой.

Пример приложения

Чтобы пояснить сказанное, создадим простое приложение ASP.NET, которое в окне браузера выглядит так, как показано на рис. 9.1:

1. Выберите команду File ▸ New ▸ ASP.NET Web Application — Delphi for .NET. На экране появится окно (рис. 9.2), в котором необходимо определить имя приложения, сервер Интернета и расположение файлов. Оставьте предлагаемые по умолчанию параметры без изменения и щелкните на кнопке OK.

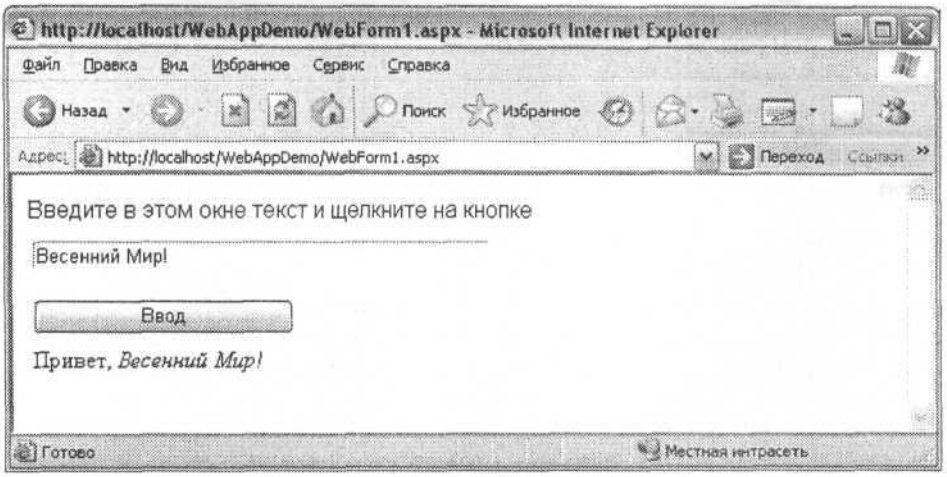


Рис. 9.1. Вид веб-приложения в окне браузера

2. Поместите на пустую веб-форму поле `TextBox`, кнопку `Button` и надпись `Label` (все компоненты из категории Web controls). Выше поля ввода напишите строку `Введите в этом окне текст и щелкните на кнопке`, как показано на рис. 9.3.
3. Для кнопки напишите такой обработчик события `Click`:

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  Label1.Text := 'Привет, <i>' + TextBox1.Text + '</i>'
end;
```

Рис. 9.2. Окно для определения названия и расположения веб-приложения

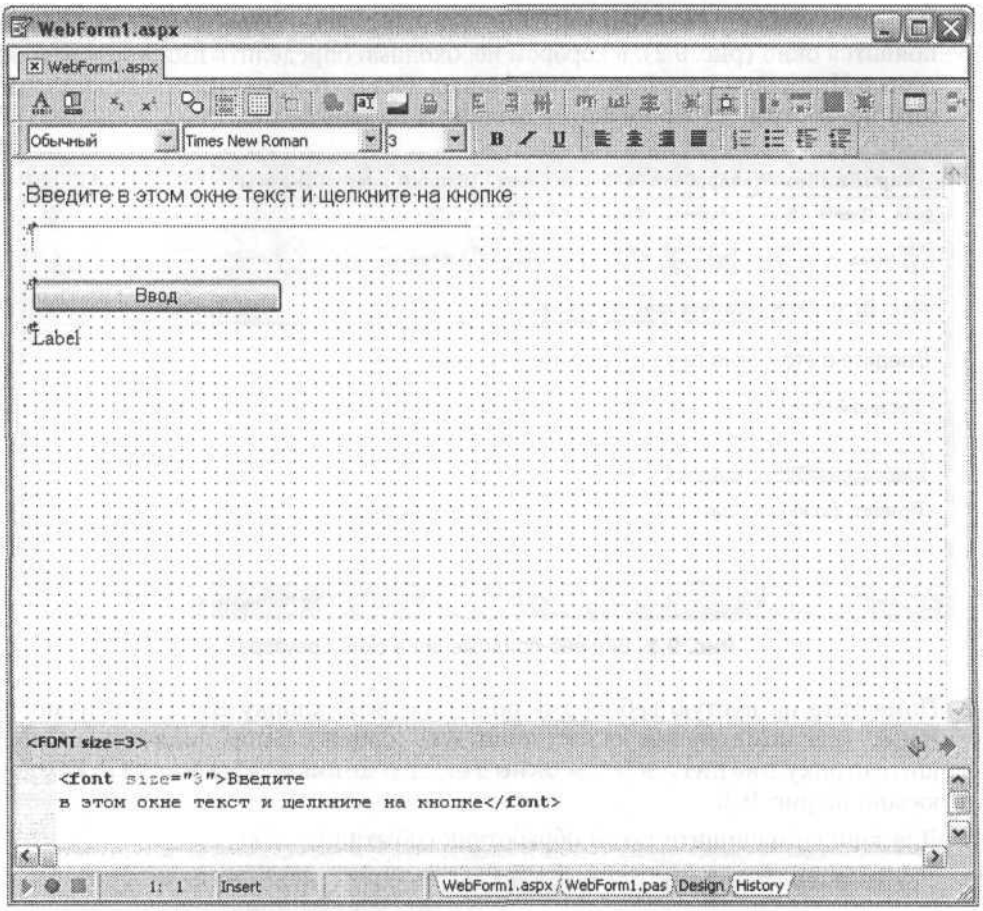


Рис. 9.3. Вид веб-формы на этапе конструирования

Для запуска приложения в среде Delphi просто нажмите клавишу F9 — среда сама вызовет браузер и заполнит адресную строку, указав имя и положение созданного aspx-файла.

Анализ приведенного примера показывает главное достоинство технологии ASP.NET: программист использует те же приемы визуального программирования, что и при создании обычного приложения или приложения для работы с базами данных. Ему не обязательно детальное знание языка HTML (XML). И хотя он может вручную изменять aspx-файл, обычно этого не требуется (щелкните на вкладке WebForm1.aspx, чтобы увидеть автоматически созданный текст HTML-документа).

Обратите внимание на использование атрибута `Runat="Server"` в стандартных HTML-дескрипторах. Например:

```
<form Runat="Server">
```

Этот атрибут указывает на серверную версию дескриптора. Кроме того, вместо атрибута NAME в серверных дескрипторах используется атрибут ID:

```
<asp:TextBox id="TextBox1"
   style="Z-INDEX: 3; LEFT: 38px; POSITION: absolute; TOP: 54px"
   runat="server" borderstyle="Inset" borderwidth="1px">
</asp:TextBox>
```

Возможности технологии ASP.NET

Помимо рассмотренных возможностей, технология ASP.NET имеет ряд других, таких как возможность работы с базами данных, возможность создания веб-служб, расширенные возможности среды и т. д. Эти возможности кратко рассматриваются в этом разделе.

Работа с базами данных

Технология ASP.NET тесно интегрирована с технологией доступа к базам данных ADO.NET. Это предельно упрощает довольно сложную задачу публикации баз данных в Интернете (интранете). Программист может использовать как стандартные для .NET Framework связные компоненты, провайдеры и адаптеры, так и средства технологии BDP.NET, которая, как это было показано в главе 3, поддерживает больше промышленных серверов БД. Для этого в состав Delphi 2005 включены компоненты категории DB Web, в том числе источники данных, сетки, навигаторы и т. д.

Создание веб-служб

Веб-службой называется установленное на сервере программное обеспечение, к которому клиент может получить доступ как к обычному сетевому ресурсу. Спецификой такого рода ресурсов является не создание текстовых HTML-страниц, а предоставление программных услуг. Например, можно создать службу, которая получает от клиента числовое значение денежной суммы и возвращает ему строку расшифровки (сумму прописью). Если, например, денежная сумма составляет 123 456,78, служба вернет такую строку:

```
сто двадцать три тысячи четыреста пятьдесят шесть рублей 78 копеек
```

С появлением технологии .NET корпорация Microsoft объявила, что делает акцент на предоставлении программных услуг, а не на разработке программного обеспечения. Клиент может на определенных условиях использовать веб-службы, то есть получать предлагаемые программные услуги. Технология ASP.NET имеет средства создания веб-служб.

Расширенные возможности среды

Кодовый редактор поддерживает автоматический контроль вводимого HTML-текста и выделение цветом синтаксических ошибок. Если текст содержит ошибку, он подчеркивается красной волнистой линией. При наведении указателя мыши на такой текст появляется окно с сообщением о возможном источнике ошибки.

Если ошибка обнаружена на этапе прогона, вызывается окно трассировки, показанное на рис. 9.4.

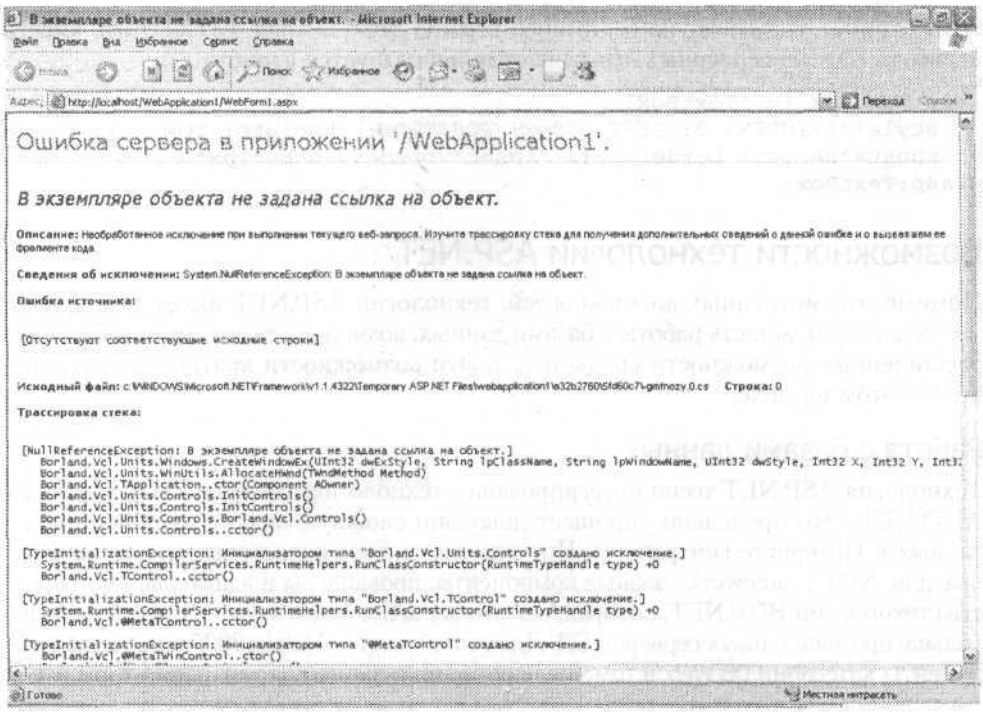


Рис. 9.4. Окно трассировки

Директива Page

Для доступа к странице ASP.NET щелкните на вкладке WebForm1.aspx в окне редактора кода. Вы увидите такой код[1]:

```
<%@ Page language="c#" Debug="true" Codebehind="WebForm1.pas"
 AutoEventWireup="false" Inherits="WebForm1.TWebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>

  <body ms_positioning="GridLayout">
    <form runat="server">
    </form>
  </body>
</html>
```

[1] Первые две строки кода в окне редактора представляют собой одну длинную строку. Вы можете разделить их так, как показано здесь, но перед компиляцией редактор вновь объединит их.

Этот код соответствует пустой веб-форме. Первый элемент кода — директива `@ Page`. Эта директива содержит атрибуты, используемые ASP.NET во время компиляции.

Атрибут `language` определяет язык для встраиваемых фрагментов кода — так называемых *сценариев*. По умолчанию этим языком является C#. Допускается также использование Visual Basic .NET (значение атрибута — `VB`) и JScript (значение атрибута — `JS`). Атрибут `Debug` указывает, следует ли в откомпилированную страницу помещать отладочную информацию. Атрибут `CodeBehind` определяет исходный pas-файл, ассоциированный с данной веб-страницей (см. следующий раздел). Атрибут `AutoEventWireup` указывает, будут ли события автоматически связываться с их обработчиками. Речь идет об обработчиках, определенных в сценариях, а не в ассоциированном pas-файле. Как мы увидим дальше, некоторые события не передаются в класс формы и, таким образом, недоступны в этом файле. Для обработки таких событий и приходится задействовать сценарии. В этом случае следует заменить значение атрибута на `true`. Наконец, атрибут `Inherits` указывает класс, объектом которого является веб-страница. Этот класс описан в ассоциированном pas-файле как наследник класса `System.Web.UI.Page`.

Атрибут CodeBehind

С помощью атрибута `CodeBehind` устанавливается связь между веб-страницей и ассоциированным pas-файлом. Автоматическое связывание страницы с pas-файлом является важнейшей особенностью ASP.NET, так как позволяет отделить задачу оформления веб-страницы от задачи реализации логики приложения. Как правило, программисты — плохие художники (сужу по своему личному опыту). Разделение двух задач позволяет более эффективно организовать совместную работу художников и программистов.

В pas-файле объявляется класс веб-страницы. Вот это объявление:

```
type
  TWebForm1 = class(System.Web.UI.Page)
  {$REGION 'Designer Managed Code'}
  strict private
    procedure InitializeComponent;
  {$ENDREGION}
  strict private
    procedure Page_Load(sender: System.Object; e: System.EventArgs);
  strict protected
    procedure OnInit(e: EventArgs); override;
  private
    { Private Declarations }
  public
    { Public Declarations }
  end;
```

В новом классе первоначально объявляются процедура `InitializeComponent` и обработчики событий `Load` и `OnInit`. По мере разработки логики приложения класс пополняется новыми членами.

Сценарии

Сценариями называются фрагменты кода, внедренные в текст веб-страницы между тегами <**script**> и </**script**>. Различают два вида сценариев: выполняемые браузером клиента и выполняемые веб-сервером. Первые могут содержать только код JavaScript, так как только его умеют интерпретировать все современные браузеры. Серверные сценарии могут содержать код C#, JScript или Visual Basic .NET. Они исполняются серверами IIS и Cassini. Если на пустую форму ASP.NET поместить элемент `Label`, то можно написать такой простой сценарий:

```
<script runat="Server">
  void Page_Load(Object obj, EventArgs e)
  {
  Label1.Text = "Hello, World!";
  }
</script>
```

Этот сценарий определяет обработчик события `Load` формы на языке C#. Если вы захотите воспроизвести пример, учтите, что сценарий можно располагать в любом месте HTML-кода, но редактор Delphi требует, чтобы он находился сразу за тегами <**title**></**title**>. Кроме того, не забудьте изменить значение атрибута `AutoEventWireup` в директиве `@ Page`.

10 Веб-формы и серверные элементы управления

Как уже отмечалось в предыдущей главе, веб-формы и серверные элементы управления являются основными элементами технологии ASP.NET.

Веб-формы

В технологии ASP.NET формы с атрибутом `Runat="Server"` называются *веб-формами*. Этот атрибут указывает на то, что форма обрабатывается сервером. Обработка формы означает, что данные передаются как от сервера в форму (браузер клиента), так и наоборот — от формы в сервер.

Основные особенности веб-форм:

- Содержимое формы создается сервером в ответ на запрос клиента.
- Содержимое формы доступно для просмотра в любом браузере клиента. Программист может вставлять в форму компоненты, учитывающие специфические особенности конкретного браузера, например Internet Explorer.
- Веб-форма реализуется в виде совокупности двух файлов: один (с расширением aspx) содержит HTML-текст, который в окне браузера интерпретируется в виде страницы с текстом и серверными элементами управления; второй (с расширением pas) реализует связанную с приложением логику и представляет собой модуль Delphi. При компиляции pas-файл преобразуется в динамически загружаемую библиотеку (DLL) и размещается в той же папке, что и aspx-файл.

В основе веб-формы лежит объект класса `Page` пространства имен `System.Web.UI`. В табл. 10.1 описываются наиболее важные свойства этого класса.

Таблица 10.1. Свойства класса Page

Свойство	Описание
`property Application: HttpApplicationState;`	Это свойство, предназначенное только для чтения, возвращает объект, содержащий текущий запрос клиента
`property Cache: Cache;`	Возвращает объект класса `Cache`, содержащий буфер запросов
`property ClientTarget: String;`	Возвращает автоматически определенные параметры или устанавливает нужные параметры браузера клиента
`property Context: HttpContext;`	Возвращает объект, содержащий параметры, связанные с текущей страницей
`property EnableViewState: Boolean;`	Возвращает или устанавливает признак, определяющий, будет ли страница сохранять состояние, которое было к моменту завершения работы с ней
`property ErrorPage: String;`	Содержит название страницы, в которой будет сохраняться протокол ошибок, связанных с текущей страницей
`property IsPostBack: Boolean;`	Содержит `True`, если страница загружается в первый раз
`property IsValid: Boolean;`	Возвращает результат вызова метода `Validation`
`property Request: HttpRequest;`	Возвращает запрос пользователя
`property Response: HttpResponse;`	Содержит ответ пользователю
`property Server: HttpServerUtility;`	Возвращает сервер, связанный с текущей страницей
`property Session: HttpSessionState;`	Возвращает объект, содержащий параметры текущего сеанса
`property Trace: TraceContext;`	Возвращает трассировку выполнения текущего запроса
`property User: IPrincipal;`	Возвращает параметры пользователя, запросившего текущую страницу
`property Validators: ValidatorCollection;`	Возвращает коллекцию компонентов, которые прошли тестирование методом `Validate`
`property ViewStateUserKey: String;`	Возвращает идентификатор пользователя, для которого страница хранит состояние вида
`property Visible: String;`	Возвращает `True`, если текущая страница была отрисована

Свойство `EnableViewState` по умолчанию имеет значение `True`, что означает способность формы и расположенных на ней серверных элементов управления

запоминать так называемое *состояние вида* (view state), то есть то состояние, которое было к моменту, когда пользователь завершил работу со страницей и вызвал другую. Если в пределах того же сеанса пользователь вновь вызовет прежнюю страницу, она будет в точности такой, какой он видел ее в последний раз. Каким образом реализуется запоминание состояния вида? Для этого в окне браузера выберите команду Вид ▸ Просмотр HTML-кода. Вы увидите дескриптор, описывающий скрытое поле:

```
<input type="hidden" name="__VIEWSTATE" value="..." />
```

Вместо точек в значении атрибута `value` будет более или менее длинный набор символов. Этот набор и есть хранилище состояния вида. Заметьте, что скрытого дескриптора `_VIEWSTATE` нет в исходном aspx-коде — его добавил и наполнил сервер.

Состояние вида сохраняется в пределах текущего сеанса, то есть до закрытия окна браузера.

Поскольку страница имеет свойство `ViewStateUserKey`, в котором указывается идентификатор пользователя, веб-страница может хранить индивидуальное состояние вида для каждого посетившего ее пользователя.

Свойство `Response` класса `HTTPResponse` содержит ответ на запрос пользователя. Класс `HTTPResponse` имеет многочисленные свойства и методы, позволяющие программисту непосредственно воздействовать на содержимое HTML-страницы. В частности, методом `Write` этого класса в формируемую страницу помещается произвольный HTML-текст. Следующий обработчик события `Load` страницы (файл Ch10\Response\WebForm1.aspx) создаст окно, показанное на рис. 10.1:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
```

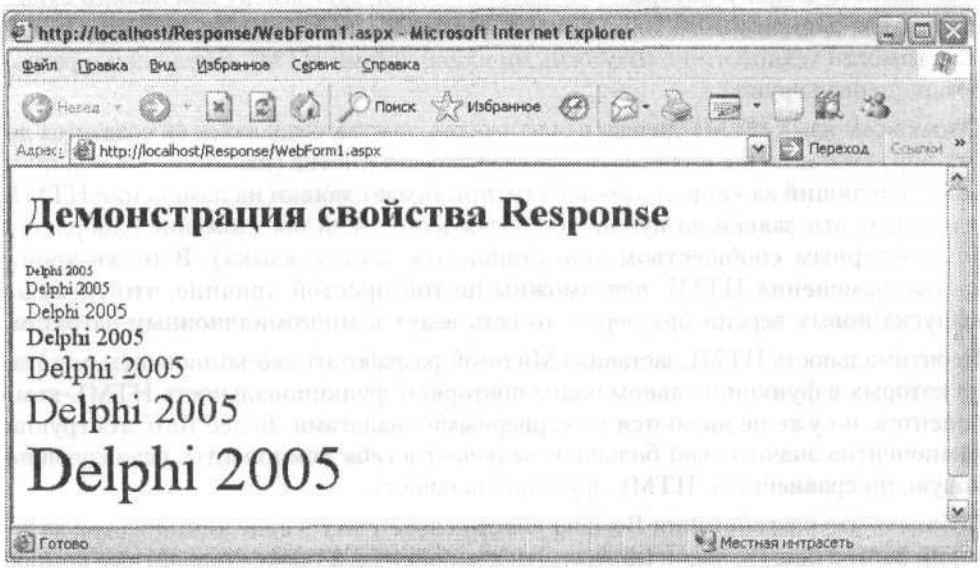


Рис. 10.1. Демонстрация метода Page.Response.Write

```
var
  k: Integer;
begin
  Page.Response.Write('<h1>Демонстрация свойства Response</h1>');
  for k := 1 to 7 do
    Page.Response.Write(System.&String.Format(
    '<font size={0}>Delphi 2005<br></font>', [k]));
end;
```

Свойство `Response` (точнее, класс `HTTPResponse`) имеет метод `Redirect`, с помощью которого можно переадресовать запрос клиента на другой сайт:

```
Page.Response.Redirect('http://piter.com');
```

Серверные элементы управления

При разработке технологии ASP.NET сотрудники Microsoft создали две группы серверных элементов управления: HTML-элементы (категория HTML Elements) и веб-элементы (категория Web Controls). Первая группа относится к пространству имен `System.Web.UI.HTMLControls`, вторая — к пространству `System.Web.UI.WebControls`.

Чем это вызвано? Дело в том, что первая группа фактически представляет собой серверные аналоги стандартных HTML-элементов. Как уже говорилось, обычный HTML-элемент превращается в серверный вставкой атрибута `Runat="Server"` и использованием атрибута `ID` вместо `NAME`. В состав компонентов .NET Framework включен компонент `HTMLGenericControl` (его нет в палитре компонентов Delphi), который реализует эту нехитрую замену для любого существующего элемента или для элемента, который еще *не создан*: именно для совместимости технологии с возможными изменениями HTML в будущем и разработана первая группа.

Однако сам язык HTML далеко не оптимален, так как создавался (и создается до сих пор) поэтапно и в его разработке участвовали многие (напомню, консорциум W3C, следящий за «порядком» в Сети, принимает заявки на изменение HTML и выносит эти заявки на публичное обсуждение; если предложение одобряется компьютерным сообществом, оно становится частью языка). В то же время частые изменения HTML невозможны по той простой причине, что требуют выпуска новых версий браузеров, то есть ведут к многомиллионным затратам.

Неоптимальность HTML заставила Microsoft разработать веб-компоненты, многие из которых в функциональном плане повторяют функциональность HTML-компонентов, но уже не являются их серверными аналогами. Более того, эта группа компонентов значительно больше и включает в себя компоненты, реализующие новую, по сравнению с HTML, функциональность.

Добавлю, что разработчики Borland внесли свою лепту в виде компонентов категорий Borland Data Provider и DB Web. Эти компоненты, а также стандартные компоненты для работы с базами данных рассмотрены в главе 12.

Компоненты категории HTML Elements

Компоненты категории HTML Elements являются серверными аналогами стандартных элементов управления языка HTML. Они перечислены в табл. 10.2.

Таблица 10.2. Серверные аналоги стандартных элементов управления

Название	Описание
HTML Label	Обычная надпись
HTML Button	Обычная кнопка
HTML TextBox	Поле ввода
HTML Text Area	Многострочное поле
HTML Password	Поле для ввода пароля
HTML Reset Button	Кнопка установки значений, используемых по умолчанию
HTML Submit Button	Кнопка пересылки данных из формы на сервер
HTML Image Button	Кнопка с изображением
HTML CheckBox	Флажок
HTML RadioButton	Переключатель
HTML DropDown	Раскрывающийся список
HTML ListBox	Список
HTML Hidden Field	Скрытое поле ввода
HTML File Upload	Поле ввода имени файла с кнопкой Обзор
HTML Anchor	Гиперссылка
HTML Table	Таблица
HTML Span	Область страницы, для которой используется особый стиль отрисовки
HTML Div	Область страницы, для которой используется особый стиль отрисовки
HTML Flow Panel	«Плавающая» область DIV
HTML Grid Panel	Фиксированная область DIV
HTML Horizontal Rule	Горизонтальная линия

Использование HTML-элементов в программах на Delphi вызывает определенные трудности. Дело в том, что многие из них (кнопки, например) требуют программной поддержки в виде сценариев. В результате теряется одно из главных преимуществ ASP.NET — отделение логики приложения от HTML-текста страницы. Кроме того, IIS не поддерживает язык программирования Delphi, так что сценарии приходится писать на C# или Visual Basic.

ПРИМЕЧАНИЕ

В приводимых далее примерах в некоторых случаях используются сценарии на языке Visual Basic .NET. В приложении Г приводится справка по этому языку программирования.

Компоненты категории Web Controls

В отличие от HTML-компонентов, компоненты категории Web Controls являются полноценными компонентами со своими методами, свойствами и событиями. Это позволяет связать с ними логику управления приложением.

ПРИМЕЧАНИЕ

Некоторые обработчики событий определены в окне инспектора объектов, но недоступны в программе. К таковым, например, относятся событие `SelectedItemChanged` для списков, событие `CheckedChange` для флажков и переключателей и т. п. Если понадобится программная реакция на такое событие, обработчик события необходимо писать в виде сценария. Таким образом, полное отделение логики программы от HTML-кода в элементах категории Web Controls тоже не достигается.

Элемент AdRotator

Элемент `AdRotator` предназначен для размещения на веб-странице рекламных баннеров. Файл, содержащий рекламную информацию, указывается в свойстве `AdvertismentFile`. В соответствующем XML-файле нужно указать сам рекламный файл, гиперссылку на страницу, к которой ведет этот баннер, альтернативный текст и некоторые другие параметры. Например (проект Ch10\AdRotator\AdRptator.dpr, рис. 10.2):

```
<Advertisements>
<Ad>
  <ImageUrl>click.gif</ImageUrl>
  <NavigateUrl>http://www.borland.com</NavigateUrl>
  <AlternateText>Borland.com</AlternateText>
  <Keyword>Computers</Keyword>
  <Impressions>80</Impressions>
</Ad>
</Advertisements>
```

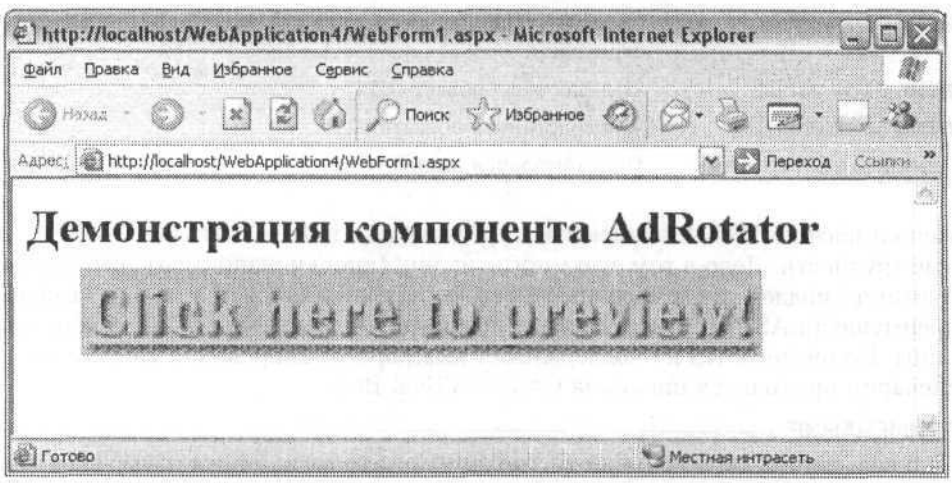


Рис. 10.2. Компонент AdRotator

Можно не определять свойство AdvertismentFile, а использовать обработчик события AdCreated:

```
procedure TWebForm1.AdRotator1_AdCreated(sender: System.Object;
            e: System.Web.UI.WebControls.AdCreatedEventArgs);
begin
  e.ImageUrl := 'click.gif';
  e.NavigateUrl := 'http://www.borland.com';
  e.AlternateText := 'Borland.com';
end;
```

Для нормальной работы компонента достаточно задать три параметра:

- ImageUrl — файл рекламного баннера;
- NavigateURL — гиперссылка на страницу, которая вызывается щелчком на баннере;
- AlternativeText — альтернативный текст, заменяющий изображение, если оно по каким-либо причинам недоступно.

Кроме того, можно задействовать два необязательных параметра:

- Keyword — определяет категории, которые могут использоваться для фильтрации баннеров;
- Impressions — число, указывающее на то, как часто баннер появляется на странице.

Кнопки

Категория Web Controls содержит три варианта кнопок: обычная (Button), графическая (ImageButton) и с гиперссылкой (LinkButton). Все три кнопки выполняют одну и ту же функцию: при щелчке на кнопке серверу передаются некоторые связанные с формой данные. Рисунок 10.3 создан с помощью формы, код которой представлен в листинге 10.1 (файл Ch10\ButtonsDemo\WebForm1.aspx).

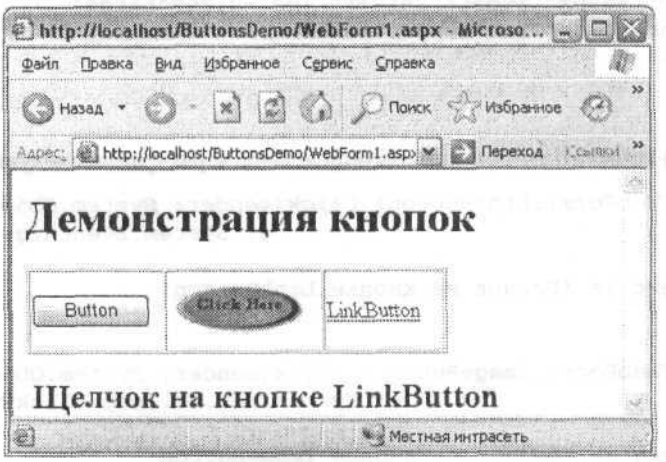


Рис. 10.3. Демонстрация кнопок

Листинг 10.1. Демонстрация кнопок

```
<%@ Page Language="c#" Debug="true" Codebehind="WebForm1.pas"
AutoEventWireup="false" Inherits="WebForm1.TWebForm1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1>Демонстрация кнопок</h1>
    <form runat="server">
      <table cellspacing="1" cellpadding="1" width="300" border="1">
        <tbody>
          <tr>
            <td style="WIDTH: 92px">
              <asp:Button id="Button1" runat="server" width="84px"
               text="Button"></asp:Button>
            </td>
            <td style="WIDTH: 108px">
              <asp:ImageButton id="ImageButton1" runat="server"
               imageurl="clickhere.gif"></asp:ImageButton>
            </td>
            <td>
              <asp:LinkButton id="LinkButton1" runat="server">
               LinkButton</asp:LinkButton>
            </td>
          </tr>
        </tbody>
      </table>
      <asp:Label id="Label1"
        style="Z-INDEX: 1; LEFT: 14px; POSITION: absolute;
        TOP: 150px" runat="server" font-size="Large"
        font-bold="True"></asp:Label>
    </form>
  </body>
</html>
```

Обработчики событий `Click` кнопок определены в pas-файле следующим образом:

```
procedure TWebForm1.LinkButton1_Click(sender: System.Object;
                                      e: System.EventArgs);
begin
  Label1.Text := 'Щелчок на кнопке LinkButton'
end;

procedure TWebForm1.ImageButton1_Click(sender: System.Object;
                         e: System.Web.UI.ImageClickEventArgs);
begin
  Label1.Text := 'Щелчок на кнопке ImageButton'
end;
```

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  Label1.Text := 'Щелчок на кнопке Button';
end;
```

Обратите внимание: аргументы обработчика кнопки `ImageButton` отличаются от аргументов обработчиков двух других кнопок. Связано это с тем, что в объекте `e: System.Web.UI.ImageClickEventArgs` передаются координаты `X` и `Y` указателя мыши в момент щелчка (рис. 10.4). Это может оказаться полезным, если кнопка имеет непрямоугольную форму (файл Ch10\ImageBtnDemo\WebForms1.aspx).

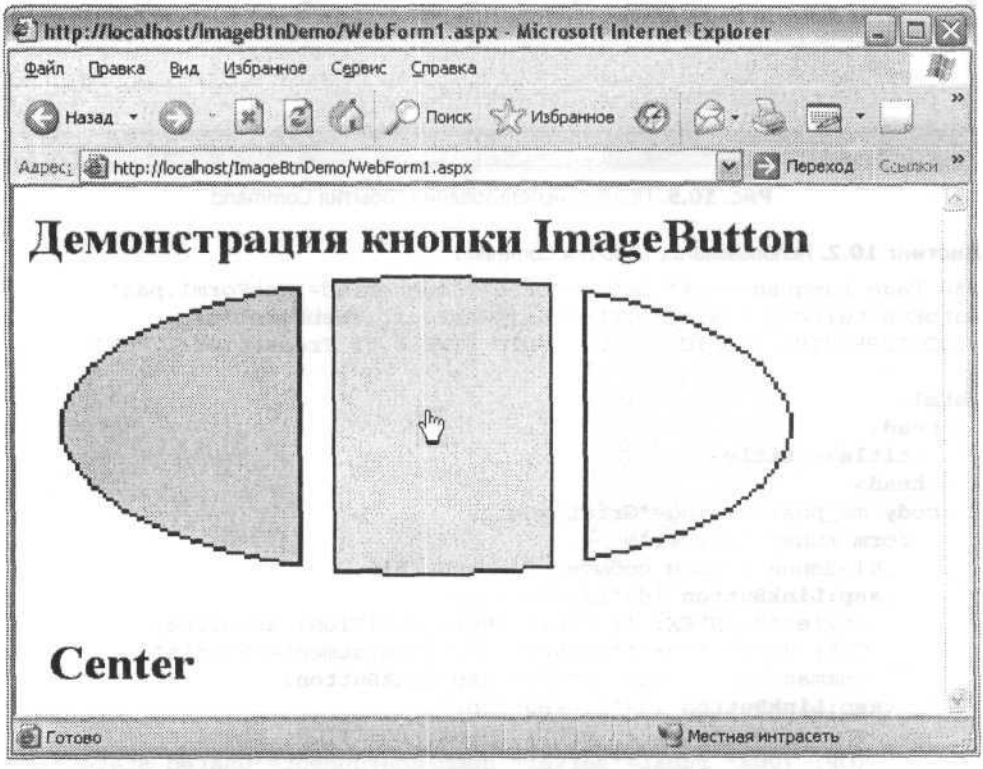


Рис. 10.4. Использование координат мыши для определения области щелчка на кнопке

Все кнопки имеют строковые свойства `CommandName` и `CommandArg`. Эти свойства отсылаются серверу при щелчке на кнопке связанным со щелчком событием `Command`. Значения этих свойств могут быть произвольными. Например, они могут использоваться для идентификации кнопки.

В следующем примере (файл Ch10\BtnCommand\WebForms1.aspx) создается окно, показанное на рис. 10.5. В окне три гиперкнопки имеют зашифрованные названия стран. При щелчке на кнопке появляется строка с расшифровкой названия (листинг 10.2).

Рис. 10.5. Пример использования события Command

Листинг 10.2. Использование события Command

```
<%@ Page Language="c#" Debug="true" Codebehind="WebForm1.pas"
AutoEventWireup="false" Inherits="WebForm1.TWebForm1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>
  <body ms_positioning="GridLayout">
    <form runat="server">
      <h1>Демонстрация события Command</h1>
      <asp:LinkButton id="LinkButton1"
        style="Z-INDEX: 1; LEFT: 46px; POSITION: absolute;
        TOP: 70px" runat="server" commandargument="Russia"
        commandname="Country">RU</asp:LinkButton>
      <asp:LinkButton id="LinkButton2"
        style="Z-INDEX: 2; LEFT: 150px; POSITION: absolute;
        TOP: 70px" runat="server" commandargument="Unated States"
        commandname="Country">US</asp:LinkButton>
      <asp:LinkButton id="LinkButton3"
        style="Z-INDEX: 3; LEFT: 254px; POSITION: absolute;
        TOP: 70px" runat="server" commandargument="South Africa"
        commandname="Country">SA</asp:LinkButton>
      <asp:Label id="Label1"
        style="Z-INDEX: 4; LEFT: 46px; POSITION: absolute;
        TOP: 110px" runat="server" font-bold="True"></asp:Label>
    </form>
  </body>
</html>
```

Все три кнопки (гиперссылки) имеют такой обработчик события Command:

```
procedure TWebForm1.LinkButton1_Command(sender: System.Object;
            e: System.Web.UI.WebControls.CommandEventArgs);
begin
  Label1.Text := 'Your choice is: ' + e.CommandArgument.ToString
end;
```

Переключатели

Для создания переключателей используются элементы RadioButton и RadioButtonList. Переключатели объединяются в группу. Каждый из них может быть установлен или сброшен. В группе переключателей в установленном состоянии в любой момент времени может быть только один переключатель: при установке другого переключателя первый автоматически сбрасывается.

Элемент RadioButton представляет собой одиночный переключатель. Он объединяется с другими переключателям с помощью своего строкового свойства GroupName. Если логическое свойство AutoPostBack содержит значение True, при установке переключателя данные формы, в которой он расположен, автоматически передаются серверу. Свойство Checked позволяет определить состояние переключателя (True, если установлен). Свойство Text определяет надпись переключателя, а свойство TextAlign — выравнивание надписи относительно границ элемента. С переключателем связано событие CheckedChanged, которое возникает при изменении его свойства Checked.

Элемент RadioButtonList представляет собой группу переключателей. Его удобно использовать для отображения полей таблицы базы данных или коллекций. Свойства элемента представлены в табл. 10.3.

Таблица 10.3. Свойства элемента RadioButtonList

Свойство	Описание
`property AutoPostBack: Boolean;`	Если содержит True, то при установке переключателя на сервер автоматически пересылает данные формы, в которой он расположен
`property CellPadding: Integer;`	Указывает количество пикселов между границей элемента и переключателем
`property CellSpasing: Integer;`	Устанавливает расстояние между отдельными переключателями
`property DataMember: String;`	Идентифицирует связываемую с элементом таблицу в источнике данных
`property DataSource: Object;`	Идентифицирует источник данных для элемента
`property DataTextField: String;`	Идентифицирует поле с данными для элемента
`property Items: ListItemCollection;`	Содержит коллекцию элементов списка
`property RepeatColumns: Integer;`	Определяет количество столбцов

продолжение

Таблица 10.3 (продолжение)

Свойство	Описание
`property RepeatDirection: RepeatDirection;`	Указывает направление, в котором должны повторяться переключатели
`property SelectedIndex: Integer;`	Возвращает индекс установленного переключателя
`property SelectedItem: String;`	Возвращает имя установленного переключателя
`property TextAlign: TextAlign;`	Определяет выравнивание надписи относительно границ переключателей

Переключатели имеют событие `CheckedCanged`, возникающее при изменении состояния элемента. Однако обработчик этого события может быть реализован только в виде сценария.

Большинство серверных элементов управления имеют свойство `AutoPostBack`, позволяющее передать серверу изменившиеся данные формы. Сервер, в свою очередь, может изменить страницу и вернуть ее браузеру пользователя. В следующем примере установка того или иного переключателя приводит к автоматическому изменению цвета формы (рис. 10.6). Соответствующий код представлен в листинге 10.3 (файл Ch10\RadioButtonListAutoPostBack\WebForm1.aspx).

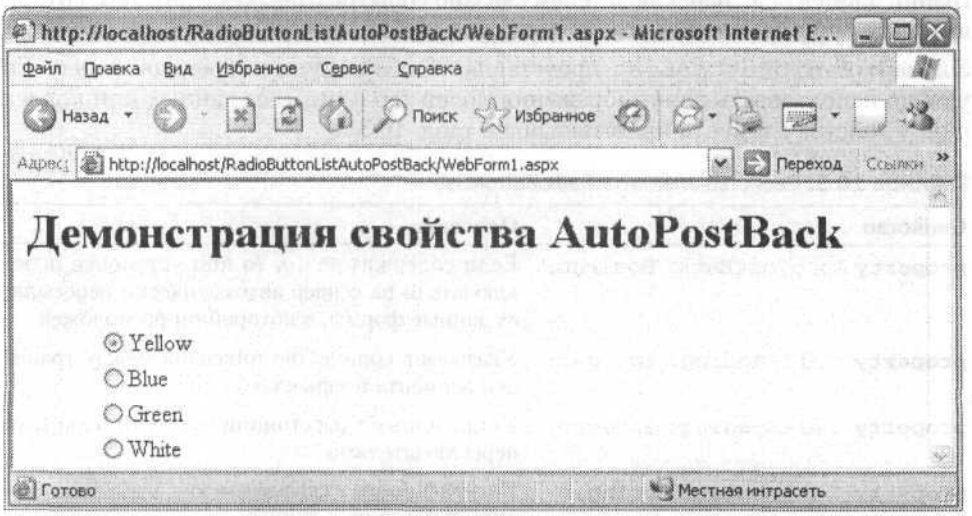


Рис. 10.6. Демонстрация свойства AutoPostBack

Листинг 10.3. Демонстрация свойства AutoPostBack

```
<html>
  <head>
    <script language="vb" runat="Server">
      Dim strBackColor As String = "white"
      Sub RadioButtonList1_SelectedIndexChanged(s As Object,
                                                e As EventArgs)
```

```
        strBackColor = RadioButtonList1.SelectedItem.Text
      End Sub
    </script>
  </head>
  <body bgcolor="<%=strBackColor%>">
    <form runat="server">
      <h1>Демонстрация свойства AutoPostBack</h1>
      <asp:RadioButtonList id="RadioButtonList1"
        style="Z-INDEX: 1; LEFT: 54px; POSITION: absolute;
        TOP: 94px" runat="server" autopostback="True"
       onselectedindexchanged=
                "RadioButtonList1_SelectedIndexChanged">
        <asp:ListItem value="Yellow">Yellow</asp:ListItem>
        <asp:ListItem value="Blue">Blue</asp:ListItem>
        <asp:ListItem value="Green">Green</asp:ListItem>
        <asp:ListItem value="White"
            selected="True">White</asp:ListItem>
      </asp:RadioButtonList>
    </form>
  </body>
</html>
```

В этом коде имеется сценарий на языке Visual Basic, в котором объявляются глобальная переменная `strBackColor` и обработчик `RadioButton1_SelectedIndexChanged`. Глобальная переменная используется в качестве значения атрибута `BgColor` тега `<body>`:

```
<body bgcolor="<%=strBackColor%>">
```

Обработчик изменяет значение этой переменной, и данные формы отсылаются серверу, сервер возвращает данные в браузер, в результате меняется цвет формы.

Флажки

Флажок обычно используется для представления логических значений (Да или Нет, Включено или Выключено и т. д.). Флажки, как и переключатели, можно объединить в группу, но, в отличие от переключателей, в одной группе может быть одновременно установлено несколько флажков.

Для представления флажков на страницах ASP.NET служат два элемента управления: `CheckBox` и `CheckBoxList`.

Элемент `CheckBox` создает единственный флажок. Его свойства полностью повторяют свойства элемента `RadioButton` за исключением свойства `GroupName`, так как флажок может не включаться в группу.

Элемент `CheckBoxList` создает группу флажков. Его свойства и события повторяют свойства и события элемента `RadioButtonList`.

В следующем примере (файл Ch10\CheckBoxList\WebForm1.aspx) группа `CheckBoxList1` содержит три флажка, свойства `Value` которых имеют значения 1, 2 и 3. При щелчке на кнопке подсчитывается сумма значений установленных флажков (рис. 10.7).

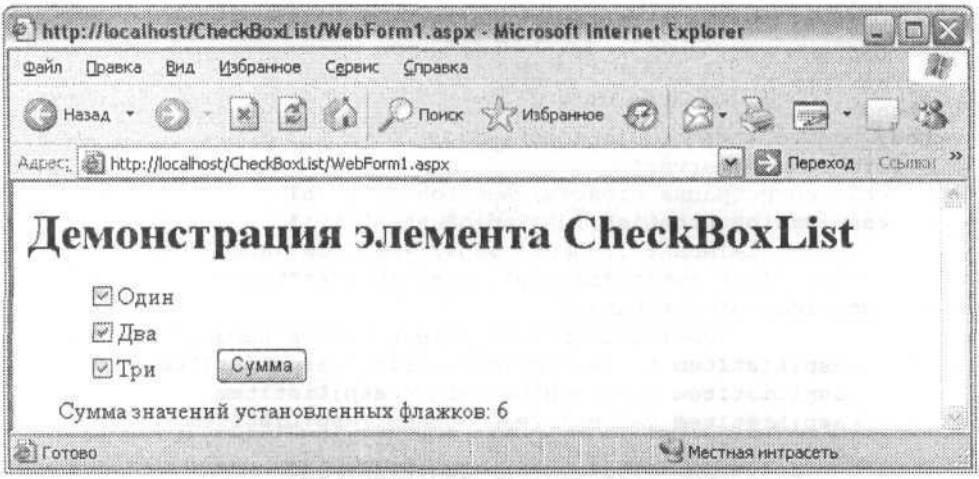


Рис. 10.7. Демонстрация компонента CheckBoxList

Обработчик щелчка на кнопке имеет следующий вид:

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  S: Integer;
  CB: ListItem;
begin
  S := 0;                                  // Начальное значение суммы
  for CB in CheckBoxList1.Items do         // Цикл по флажкам
    if CB.Selected then                    // Флажок установлен?
      S := S + Int32.Parse(CB.Value); //-Да. Учитываем его значение
  Label1.Text := 'Сумма значений установленных флажков: ' +
                                                        S.ToString
end;
```

Значение `Value` флажка определено как строка, поэтому при суммировании оно преобразуется к целому числу. Для преобразования используется функция `Parse` типа данных `Int32`.

Флажки имеют событие `CheckedChange`, обработчик которого может быть реализован только в виде сценария.

Элемент Label

Элемент `Label` (надпись) используется для вывода текста и является экземпляром класса `System.Web.UI.WebControls.Label`. Свойство `Text` содержит отображаемый в надписи текст. С помощью свойств `BorderColor`, `ForeColor` и `BackColor` можно указать цвет границы, текста и фона надписи.

Элемент TextBox

Элемент `TextBox` предназначен для ввода текста. Тип элемента определяется свойством `TextMode`, которое может принимать одно из следующих значений:

- `Single` — стандартное однострочное поле;
- `MultiLine` — многострочное поле;
- `Password` — поле для ввода пароля.

Свойство `MaxLength` устанавливает максимальное количество отображаемых символов. Свойство `Columns` определяет ширину поля в символах, а свойство `Rows` — высоту поля в количестве отображаемых строк (имеет значение только для многострочного поля).

Для элемента определено событие `TextChanged`, которое доступно только с помощью сценария.

Списки

Элементы `ListBox` и `DropDownList` представляют собой списки. Элементами списка являются экземпляры класса `ListItem`, совокупность которых образует коллекцию `Items` элемента.

Список `ListBox` всегда раскрыт и демонстрирует все предусмотренные в нем элементы. Доступ к элементам списка `DropDownList` осуществляется только после щелчка на кнопке раскрытия списка.

Свойство `SelectedItem` возвращает выбранный элемент списка, а свойство `SelectedValue` — связанную с элементом строку. Если свойство `SelectedMode` имеет значение `Multiple`, список позволяет выбирать несколько значений.

Списки имеют событие `SelectedIndexChange`, обработчик которого реализуется в виде сценария.

В следующем примере, код которого представлен в листинге 10.4 (файл Ch10\ListBox\WebForm1.aspx), на форме располагается список с названиями цветов. Изменение выбора в списке приводит к соответствующему изменению цвета надписи (рис. 10.8).

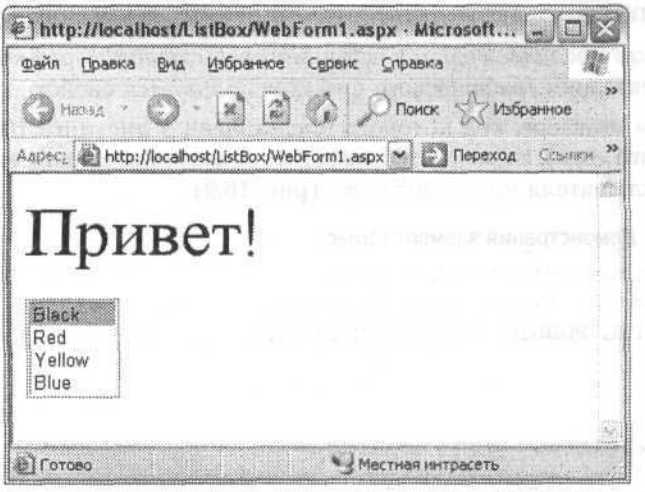


Рис. 10.8. Использование обработчика события SelectedIndexChanged списка

Листинг 10.4. Демонстрация списков

```
<html>
  <head>
    <script language="vb" runat="Server">
    Dim strColor As String = "black"
    Sub ListBox1_SelectedIndexChanged(s As Object,
                                      e As EventArgs)
      strColor = ListBox1.SelectedItem.Text
    End Sub
  </script>
  </head>
  <body ms_positioning="GridLayout"><font
      color="<%=strColor%>" size="8">Привет!</font>
    <form runat="server">
      <asp:ListBox id="ListBox1" runat="server"
        onselectedindexchanged="ListBox1_SelectedIndexChanged"
        autopostback="True">
        <asp:ListItem value="Black"
                      selected="True">Black</asp:ListItem>
        <asp:ListItem value="Red">Red</asp:ListItem>
        <asp:ListItem value="Yellow">Yellow</asp:ListItem>
        <asp:ListItem value="Blue">Blue</asp:ListItem>
      </asp:ListBox>
    </form>
  </body>
</html>
```

Обратите внимание: свойство `AutoPostBack` элемента `ListBox1` имеет значение `True`. Только в этом случае смена выбранного в списке элемента приведет к немедленному изменению цвета текста.

Элемент Image

Элемент `Image` предназначен для вставки на веб-страницу графического изображения. Для указания графического файла используется свойство `ImageURL`.

В следующем примере, код которого представлен в листинге 10.5 (файл Ch10\Image\WebForm1.aspx), элемент отображает один из трех рисунков, выбранных с помощью переключателя `RadioButton1` (рис. 10.9).

Листинг 10.5. Демонстрация элемента Image

```
<%@ Page Language="c#" Debug="true" Codebehind="WebForm1.pas"
AutoEventWireup="false" Inherits="WebForm1.TWebForm1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>
  <body ms_positioning="GridLayout">
```

```
    <form runat="server">
      <h1>Демонстрация компонента Image</h1>
      <asp:Image id="Image1" style="Z-INDEX: 1; LEFT: 14px;
                 POSITION: absolute; TOP: 62px"
        runat="server" height="299px" width="627px"></asp:Image>
      <asp:RadioButtonList id="RadioButtonList1"
        style="Z-INDEX: 2; LEFT: 102px; POSITION: absolute;
        TOP: 358px" runat="server">
        <asp:ListItem value="Голубые холмы.jpg">
                                  Голубые холмы</asp:ListItem>
        <asp:ListItem value="Закат.jpg">Закат</asp:ListItem>
        <asp:ListItem value="Зима.jpg">Зима</asp:ListItem>
      </asp:RadioButtonList>
      <asp:Button id="Button1" style="Z-INDEX: 3; LEFT: 262px;
                 POSITION: absolute; TOP: 374px"
        runat="server" text="Показать"></asp:Button>
    </form>
  </body>
</html>
```

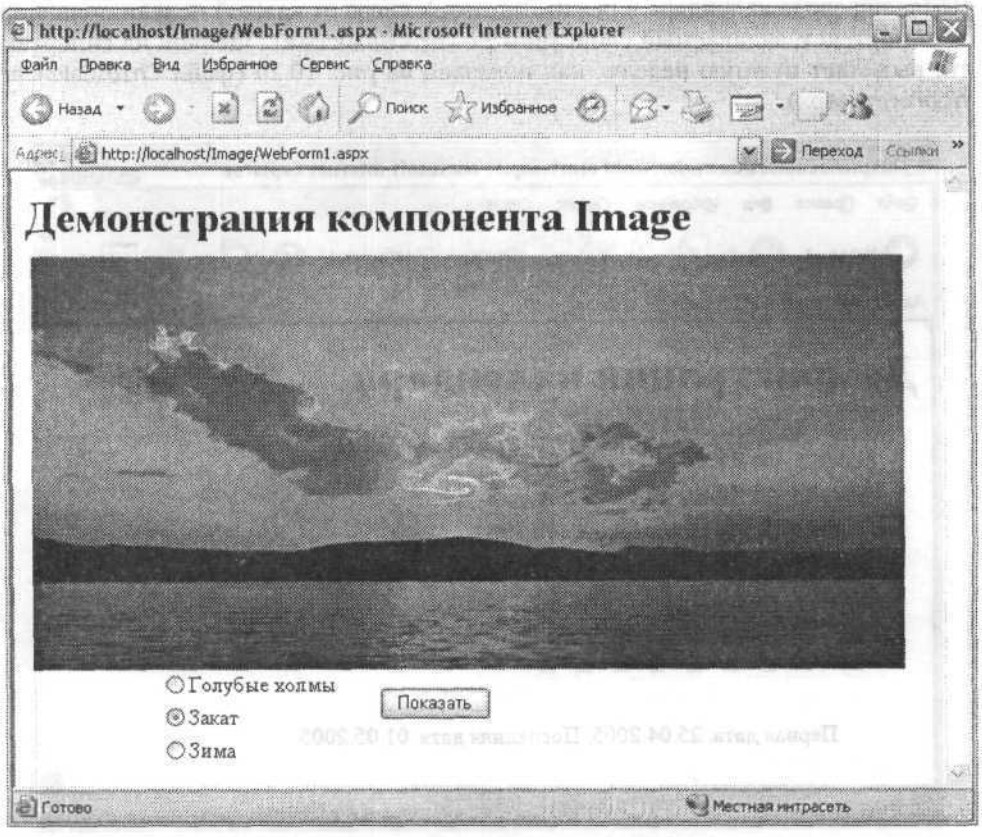


Рис. 10.9. Демонстрация элемента Image

В модуле WebForm1.pas определен такой обработчик щелчка на кнопке Показать:

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  Image1.ImageUrl := RadioButtonList1.SelectedValue
end;
```

Элемент Calendar

Элемент `Calendar` предназначен для показа календаря. С его помощью можно выбрать нужную дату, которая доступна программе в свойстве `SelectedDay`. Свойство `SelectionMode` определяет режим выбора и может иметь одно из следующих значений:

- `Day` — выбирается один день (значение по умолчанию);
- `DayWeek` — выбирается один день или неделя;
- `DayWeekMonth` — выбирается день, неделя или месяц;
- `None` — выбор запрещен.

После перевода календаря в режим `DayWeek` слева от каждой недели появляется гипертекст, задаваемый свойством `SelectWeekText`: щелчок на гипертексте выделяет нужную неделю, как показано на рис. 10.10 (файл Ch10\Calendar\WebForm1.aspx).

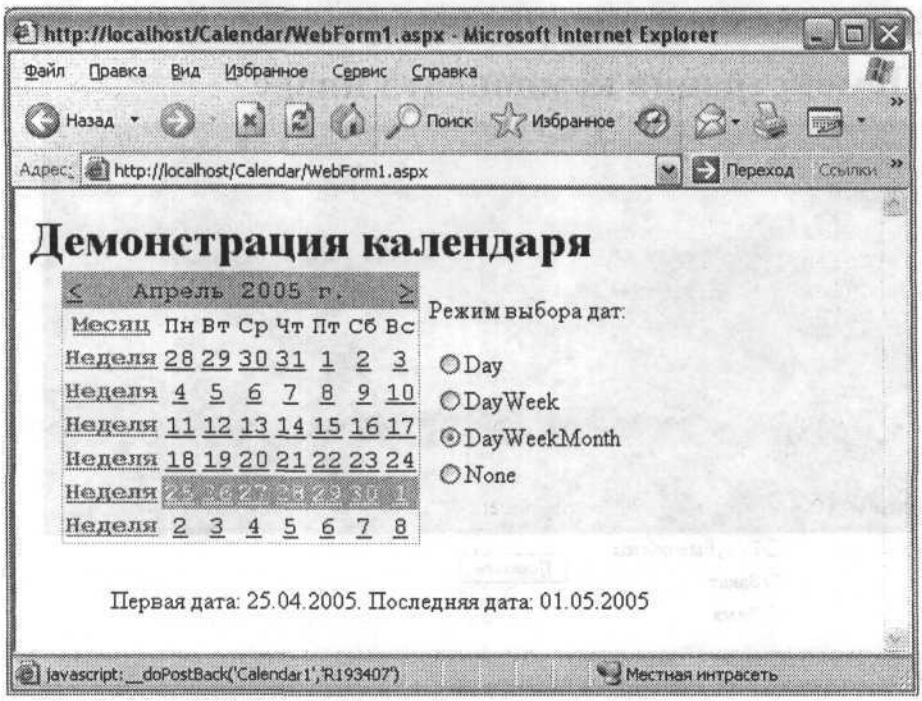


Рис. 10.10. Выделение в календаре недели

В режиме DayWeekMonth, кроме того, в левом верхнем углу появляется гипертекст, задаваемый свойством SelectMonthText. Щелчок на этой гиперссылке выделяет нужный месяц. В обоих случаях диапазон выбранных дат возвращается в коллекции свойства SelectedDates.

Рисунок 10.10 создан программой, в которой определены следующие обработчики событий:

```
procedure TWebForm1.Calendar1_SelectionChanged(sender:
                          System.Object; e: System.EventArgs);
begin
  case Calendar1.SelectionMode of
    CalendarSelectionMode.None: Label2.Text := '';
    CalendarSelectionMode.Day: Label2.Text := 'Выбрана дата '
      + DateToStr(Calendar1.SelectedDate);
  else
    Label2.Text := 'Первая дата: ' +
        DateToStr(Calendar1.SelectedDates[0]) +
        '. Последняя дата: ' + DateToStr(Calendar1.SelectedDates[
        Calendar1.SelectedDates.Count - 1]);
  end;
end;

procedure TWebForm1.RadioButtonList1_SelectedIndexChanged(sender:
                          System.Object; e: System.EventArgs);
var
  k: ListItem;
  i: Integer;
begin
  for k in RadioButtonList1.Items do
    if k.Selected then i := StrToInt(k.Value);
  case i of
  0: Calendar1.SelectionMode := CalendarSelectionMode.None;
  1: Calendar1.SelectionMode := CalendarSelectionMode.Day;
  2: Calendar1.SelectionMode := CalendarSelectionMode.DayWeek;
  3: Calendar1.SelectionMode := CalendarSelectionMode.DayWeekMonth;
  end;
end;
```

Помимо описанных, элемент имеет многочисленные свойства оформительского характера, некоторые из которых перечислены в табл. 10.4.

Таблица 10.4. Свойства элемента Calendar

Свойство	Описание
`property CellPading: Integer;`	Указывает расстояние в пикселах между содержимым ячейки и ее границами
`property CellSpacing: Integer;`	Определяет расстояние в пикселах между ячейками

продолжение

Таблица 10.4 (продолжение)

Свойство	Описание
`property DayNameFormat: DayNameFormat;`	Устанавливает формат представления дней недели: `FirstLetter` — первая буква названия дня; `FirstTwoLetters` — первые две буквы; `Full` — полное название; `Short` — краткое название (по умолчанию)
`property NextMonthText: String;`	Указывает текст для гиперссылки на следующий месяц
`property NextPrevFormat: NextPrevFormat;`	Указывает формат гиперссылок на предыдущий и следующий месяцы: `CustomText` — текст в свойствах `NextMonthText` и `PrevMonthText` (по умолчанию); `FullMonth` — полное название месяца; `ShortMonth` — краткое название месяца
`property PrevMonthText: String;`	Указывает текст для гиперссылки на предыдущий месяц
`property ShowDayHeader: Boolean;`	Если содержит `True`, выводится строка с названиями дней недели
`property ShowGridLines: Boolean;`	Если содержит `True`, ячейки отделяются линиями
`property ShowNextPrevMonth: Boolean;`	Если содержит `True`, выводятся гиперссылки на предыдущий и следующий месяцы
`property ShowTitle: Boolean;`	Если содержит `True`, выводится заголовок
`property TitleFormat: TitleFormat;`	Определяет формат заголовка: `Month` — название месяца; `MonthYear` — название месяца и год
`property TodaysDate: DateTime;`	Содержит текущую дату
`property VisibleDate: DateTime;`	Содержит показываемый месяц

При отрисовке очередной ячейки календаря возникает событие `DayRender`. Программист в обработчике события может изменить стандартный вид ячейки.

Обработчик события `DayRender` в параметре `e: DayRenderEventArgs` получает свойства, некоторые из которых перечислены в табл. 10.5.

Таблица 10.5. Свойства, получаемые в параметре DayRenderEventArgs

Свойство	Описание
`property Cell: TableCell;`	Ячейка таблицы с визуализируемым днем
`property Cell.ColumnSpan: Integer;`	Количество столбцов таблицы, которое должна занимать ячейка
`property Cell.HorizontalAlign: HorizontalAlign;`	Горизонтальное выравнивание текста ячейки: `Center` — по центру; `Justify` — по левому и правому краям одновременно; `Left` — по левому краю; `NoSet` — не установлено (по умолчанию); `Right` — по правому краю

Свойство	Описание
`property Cell.RowSpan: Integer;`	Количество строк таблицы, которое должна занимать ячейка
`property Cell.Text: String;`	Текст в ячейке
`property Cell.VerticalAlign: VerticalAlign;`	Вертикальное выравнивание текста ячейки: `Bottom` — по нижнему краю; `Middle` — по центру; `NoSet` — не установлено (по умолчанию); `Top` — по верхнему краю
`property Cell.Wrap: Boolean;`	Если содержит `True`, разрешается перенос текста на новую строку
`property Day: CalendarDay;`	Объект, содержащий визуализируемый день
`property Day.Date: DateTime;`	Визуализируемая дата
`property Day.DayNumberText: String;`	Визуализируемый текст
`property Day.IsOtherMonth: Bolean;`	Если содержит `True`, дата не входит в текущий месяц
`property Day.IsSelectable: Boolean;`	Если содержит `True`, дата выбрана
`property Day.IsToday: Boolean;`	Если содержит `True`, отображается текущая дата
`property Day.IsWeekend: Boolean;`	Если содержит `True`, дата относится к субботе или воскресенью

В следующем примере (файл Ch10\CalendarRender\WebForm1.aspx) выделяются все субботы и воскресенья, а также текущий день (рис. 10.11).

Обработчик выделения текущей даты и нерабочих дней выглядит следующим образом:

```
procedure TWebForm1.Calendar1_DayRender(sender: System.Object;
            e: System.Web.UI.WebControls.DayRenderEventArgs);
begin
  if e.Day.IsToday then                  // Текущий день?
    e.Cell.BackColor := Color.Orange  //-Да
  else if e.Day.IsWeekend then           // Конец недели?
    e.Cell.BackColor := Color.Yellow; //-Да
end;
```

Элемент Literal

Элемент управления `Literal` предназначен для вывода на веб-страницу небольших фрагментов текста. В отличие от элемента `Label`, свойство `Text` элемента `Literal` не может содержать HTML-текст.

Чаще всего элемент `Literal` используется для точного указания места вывода текста. В следующем примере, код которого представлен в листинге 10.6 (файл Ch10\LiteralDemo\WebForm1.aspx), элементы `Literal` вставляются в ячейки таблицы

(рис. 10.12). Поскольку такие элементы недоступны в обработчике `Page_Load` формы WebForm1.pas, в файле WebForm1.aspx используется сценарий на языке Visual Basic .NET.

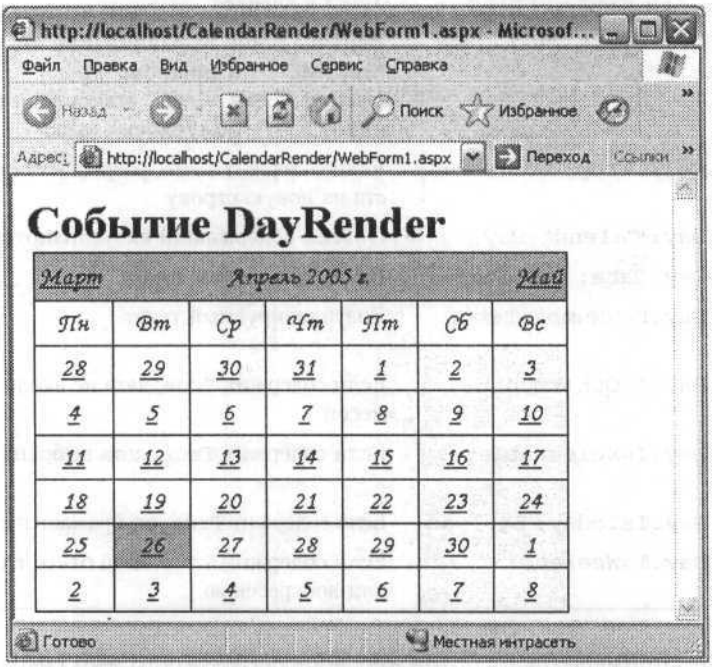


Рис. 10.11. Выделение текущей даты, а также нерабочих дней

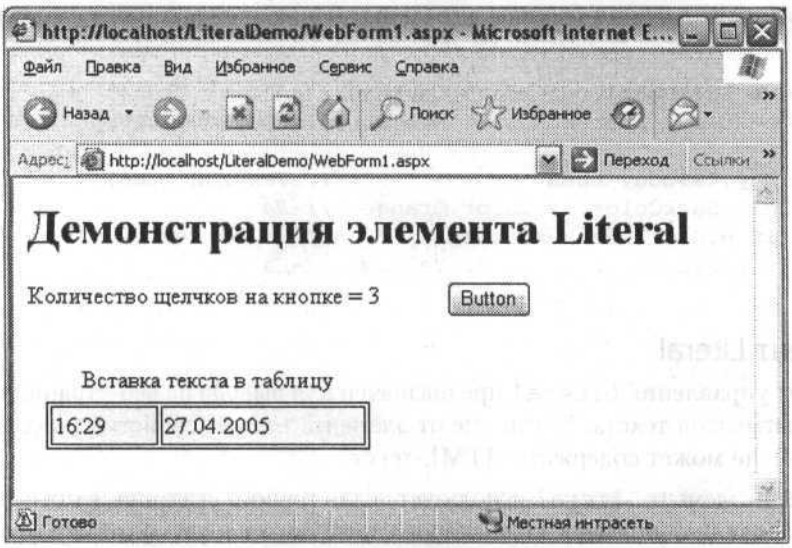


Рис. 10.12. Демонстрация элементов Literal

Листинг 10.6. Демонстрация элементов Literal

```
<%@ Page CodeBehind="WebForm1.pas" Inherits="WebForm1.TWebForm1" %>
<html>
  <head>
    <title></title>
    <script language="vb" runat="server">
      Sub Page_Load(s As Object, e As EventArgs)
        litTime.Text = DateTime.Now.ToShortTimeString()
        litDate.Text = DateTime.Now.ToShortDateString()
      End Sub
    </script>
  </head>
  <body onload="Page_Load" ms_positioning="GridLayout">
    <form runat="server">
      <h1>Демонстрация элемента Literal</h1>Количество щелчков на
    кнопке =
      <asp:Literal id="Literal1" runat="server" text="0">
      </asp:Literal>
      <asp:Button id="Button1"
        style="Z-INDEX: 1; LEFT: 294px; POSITION: absolute;
        TOP: 70px" runat="server" text="Button"></asp:Button>
      <asp:Table id="Table1"
        style="Z-INDEX: 2; LEFT: 22px; POSITION: absolute;
        TOP: 126px" runat="server" gridlines="Vertical"
        backcolor="Yellow" bordercolor="Black" borderwidth="1px"
        borderstyle="Solid"width="219px"
        caption="Вставка текста в таблицу" height="33px">
        <asp:TableRow>
          <asp:TableCell><asp:Literal id="litTime" runat="Server"/>
          </asp:TableCell>
          <asp:TableCell><asp:Literal id="litDate" runat="server"/>
          </asp:TableCell>
        </asp:TableRow>
      </asp:Table>
    </form>
  </body>
</html>
```

Как видно из приведенного кода, в форме используются три элемента `Literal`. Элемент `Literal1` помещен на форму обычным перетаскиванием из палитры компонентов, а вот элементы `litTime` и `litDate` вставлены вручную (редактированием текста файла **WebForm1.aspx**). В результате они не видны в файле **WebForm1.pas**, а программный доступ к ним возможен только с помощью сценария.

Заметьте, что эксперт создания приложения ASP.NET вставляет в начало HTML-текста страницы такие строки:

```
<%@ Page Language="c#" Debug="true" Codebehind="WebForm1.pas"
AutoEventWireup="false" Inherits="WebForm1.TWebForm1"%>
<%@ Register tagprefix="borland" Namespace="Borland.Data.Web"
Assembly="Borland.Data.Web"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

В них указывается язык программирования C# (атрибут Language) и запрещается автоматическое возникновение события onload (атрибут AutoEventWireup). Эти строки нужно заменить такой:

```
<%@ Page CodeBehind="WebForm1.pas" Inherits="WebForm1.TWebForm1" %>
```

В результате сохраняется связь с файлом WebForm1.pas и становится работоспособным обработчик щелчка на кнопке.

В дескриптор <**body**> добавлен атрибут onload="Page_Load"; если этого не сделать, событие onload обработано не будет.

Поскольку элемент Literal1 вставлен в форму обычным способом, он доступен в файле WebForm1.pas:

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  k: Integer;
begin
  k := Int32.Parse(Literal1.Text) + 1;
  Literal1.Text := k.ToString;
end;
```

Элемент Panel

Элемент Panel используется для группировки нескольких элементов управления. Используя свойство Visible, панель вместе с расположенными на ней элементами можно прятать и показывать, как это сделано в проекте Ch10\PanelDemo\PanelDemo.dpr (*рис. 10.13*).

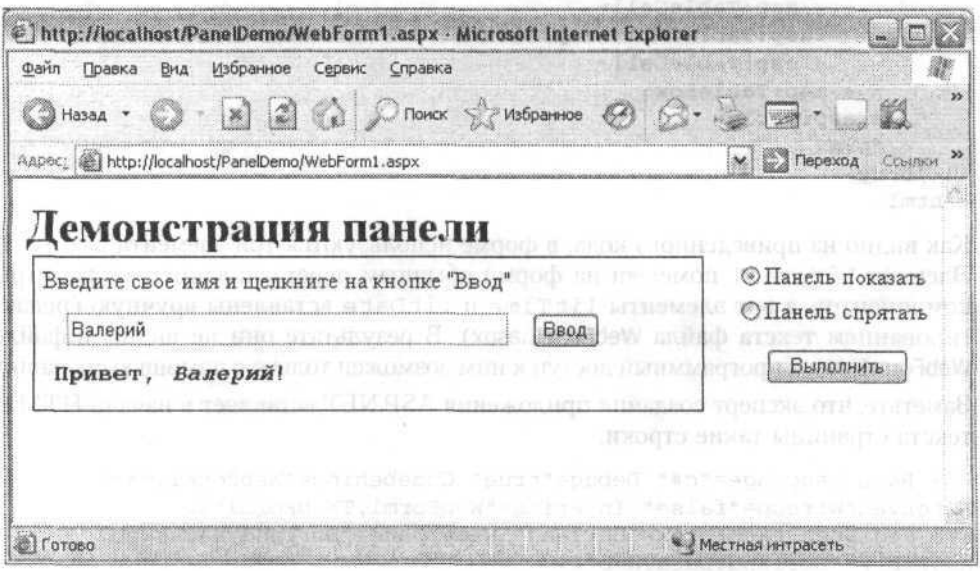


Рис. 10.13. Демонстрация панели

В модуле проекта определены такие обработчики:

```
procedure TWebForm1.btName_Click(sender: System.Object;
                                 e: System.EventArgs);
begin
  lbName.Text := 'Привет, <i>' + tbName.Text + '</i>!';
end;

procedure TWebForm1.btHide_Click(sender: System.Object;
                                 e: System.EventArgs);
begin
  if rbHide.Items[0].Selected then
    Panel1.Visible := True
  else
    Panel1.Visible := False
end;
```

Замечу, что для элемента RadioButtonList в окне инспектора объектов указано событие SelectedItemChanged, возникающее при установке другого переключателя. Однако в модуль Delphi это событие не передается. В этом нетрудно убедиться:

```
procedure TWebForm1.rbHide_SelectedIndexChanged(sender:
                          System.Object; e: System.EventArgs);
begin
  Panel1.Visible := not Panel1.Visible
end;
```

Несмотря на то что обработчик определен, он не реагирует на установку другого переключателя, поэтому изменение видимости панели осуществляется в обработчике щелчка на кнопке Выполнить.

Любопытной особенностью элемента является отсутствие у него свойства Text, хотя определены такие свойства, как Font, ForeColor, HorizontalAlignment, Wrap. Связано это с тем, что при вставке элемента в форму между тегами **<asp:Panel>** и **</asp:Panel>** вставляется текст с именем панели, так что перечисленные свойства относятся к этому тексту. Изменить или удалить текст можно только непосредственным редактированием aspx-файла.

Элемент Table

Элемент Table предназначен для создания таблиц. Строки таблицы являются экземплярами класса TableRow, а их совокупность хранится в коллекции свойства Rows. Каждая строка состоит из ячеек, которыми являются экземпляры класса TableCell.

В табл. 10.6 указаны наиболее важные свойства элемента.

Свойство Rows — основное свойство элемента. С помощью редактора этого свойства можно определить строки и ячейки таблицы на этапе создания формы, а с помощью методов Add, Insert, Remove свойства Rows — на этапе прогона.

Таблица 10.6. Свойства элемента Table

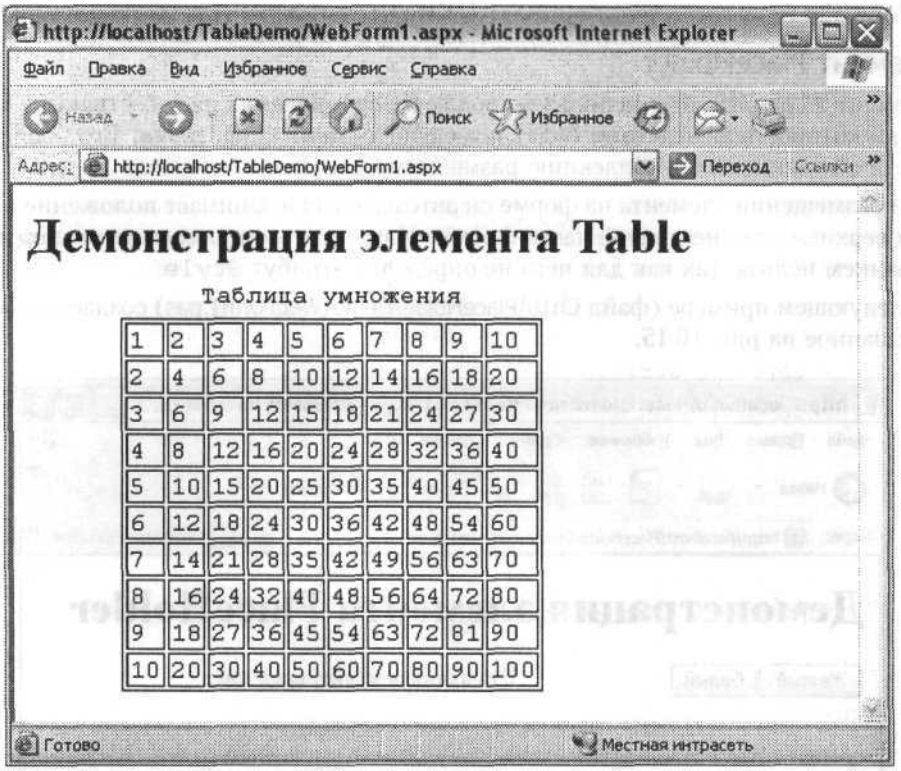
Свойство	Описание
`property BackColor: Color;`	Определяет умалчиваемый цвет фона ячеек и границ
`property BackImageUrl: String;`	Задает имя файла с фоновым рисунком
`property BorderColor: Color;`	Определяет цвет линий границ
`property BorderStyle: BorderStyle;`	Определяет стиль границ рядов: `NotSet` — не установлен (по умолчанию); `None` — нет границы; `Dotted` — линия из точек; `Dashed` — линия из тире; `Solid` — сплошная линия; `Double` — двумя линиями; `Groove` — в виде желоба; `Ridge` — в виде гребня; `Inset` — эффект тиснения; `Outset` — эффект выпуклости
`property BorderWidth: Integer;`	Определяет толщину границ строк
`property Caption: String;`	Задает заголовок таблицы
`property CaptionAlign: CaptionAlign;`	Определяет выравнивание заголовка: `Bottom` — снизу и по центру; `Left` — влево; `NotSet` — не определено (по умолчанию); `Right` — вправо; `Top` — сверху и по центру
`property CellPadding: Integer;`	Расстояние в пикселах между текстом и границами ячейки
`property CellSpacing: Integer;`	Расстояние в пикселах между соседними ячейками
`property Font: Font;`	Шрифт заголовка
`property ForeColor: Color;`	Цвет заголовка
`property GridLines: GridLines;`	Определяет стиль линий, разделяющих соседние ячейки: `Both` — по вертикали и горизонтали; `Horizontal` — по горизонтали; `None` — нет линий; `Vertical` — по вертикали
`property Rows: TableRowsCollection;`	Содержит коллекцию объектов класса `TableRow`

Класс `TableRow` имеет собственные оформительские свойства `BackColor`, `BorderColor`, `BorderStyle`, `BorderWith`, `Font`, `ForeColor`, `HorizontalAlign`, которые аналогичны по назначению одноименным свойствам таблицы, но относятся только к одной ее строке. По умолчанию значения этих свойств такие же, как и у таблицы, но программист может их изменять по своему усмотрению, выделяя ту или иную строку собственным цветом, стилем границ, шрифтом и т. д.

Центральным свойством класса `TableRow` является свойство `Cells`, определяющее коллекцию ячеек — объектов класса `TableCell`. Любопытно, что количество ячеек в каждой строке может быть разным, что позволяет создавать таблицы причудливой формы. Каждая ячейка имеет свой набор оформительских свойств,

это дает возможность выделять отдельные ячейки. В классе `TableCell` определено свойство `Text`, задающее демонстрируемый в ячейке текст.

В проекте Ch10\TableDemo\TableDemo.dpr создается окно, показанное на рис. 10.14.



Демонстрация элемента Table

Таблица умножения

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Рис. 10.14. Демонстрация таблицы

В модуле проекта определен такой обработчик события `Page_Load`:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
var
  i, j, k: Integer;
  TR: TableRow;
  TC: TableCell;
begin
  for i := 1 to 10 do                    // Цикл по строкам
  begin
    TR := TableRow.Create;               // Создаем новую строку
    Table1.Rows.Add(TR);                 // Вставляем ее в таблицу
    for j := 1 to 10 do                  // Цикл по ячейкам
    begin
      TC := TableCell.Create;            // Создаем ячейку
      k := i * j;
```

```
      TC.Text := k.ToString;              // Помещаем в нее текст
      Table1.Rows[i - 1].Cells.Add(TC);// Добавляем в строку
    end;
  end;
end;
```

Элемент PlaceHolder

Элемент `PlaceHolder` используется для резервирования на веб-странице места, на которое впоследствии будет помещен элемент управления. Его свойство `Controls` определяет коллекцию размещаемых на нем элементов.

При размещении элемента на форме он автоматически занимает положение в левом верхнем углу незанятой части формы. Изменить это его положение перетаскиванием нельзя, так как для него не определен атрибут `Style`.

В следующем примере (файл Ch10\PlaceHolderDemo\WebForm1.pas) создается окно, показанное на рис. 10.15.

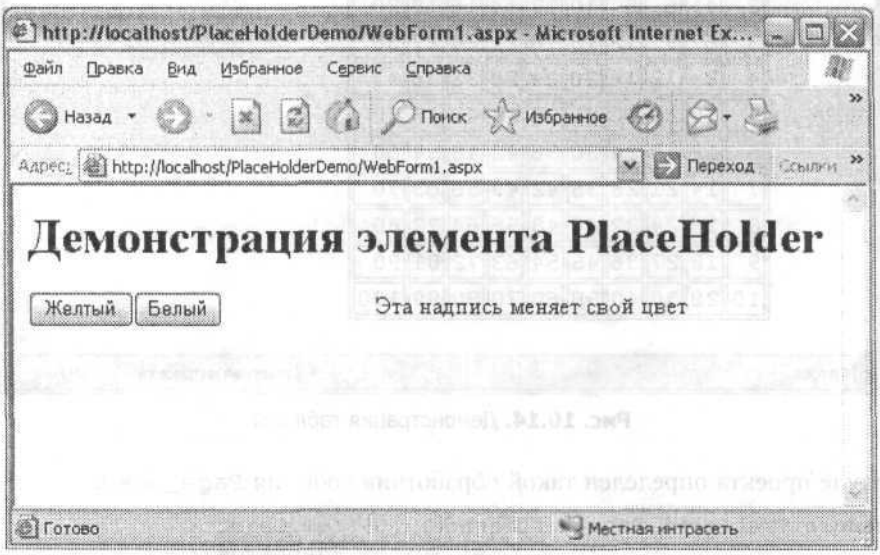


Рис. 10.15. Кнопки в окне помещены на элемент PlaceHolder

На этапе конструирования на форме размещаются заголовок, обрамленный тегами <h1> и </h1>, элемент `PlaceHolder` и элемент `Label`. В обработчике события `Load` формы в элемент `PlaceHolder` вставляются две кнопки:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
var
  Bt: Button;
begin
  Bt := Button.Create;                // Создаем первую кнопку
  Bt.Text := 'Желтый';                // Надпись на ней
```

```
  Include(Bt.Click, BtnYellow_Click);// Обработчик события Click
  PlaceHolder1.Controls.Add(Bt);     // Помещаем на PlaceHolder
  Bt := Button.Create;               // Создаем вторую кнопку
  Bt.Text := 'Белый';
  Include(Bt.Click, BtnWhite_Click);
  PlaceHolder1.Controls.Add(Bt);
end;

procedure TWebForm1.BtnYellow_Click(S: &Object; e: EventArgs);
begin
  Label1.BackColor := Color.Yellow;
end;

procedure TWebForm1.BtnWhite_Click(S: &Object; e: EventArgs);
begin
  Label1.BackColor := Color.White;
end;
```

Обратите внимание на два момента. Во-первых, кнопка `Bt` в обработчике `Page_Load` создается дважды. Если не обратиться к конструктору для создания второй кнопки, объект `Bt` будет ссылаться на прежнюю область памяти и на элемент `PlaceHolder` будет дважды помещена только вторая кнопка. Во-вторых, в приложениях ASP.NET связывать с динамически создаваемым объектом обработчик события можно только с помощью перекрытой процедуры `Include`.

Элемент HyperLink

Элемент `HyperLink` реализует гиперссылку на веб-странице. Для этого у него определено свойство `NavigateURL`, в которое нужно поместить требуемый URL-адрес. При щелчке на элементе происходит смена страниц в окне браузера. Если свойство `NavigateURL` не определено (содержит пустую строку), элемент полностью повторяет функциональность элемента `Label`. Если в свойство `ImageURL` поместить маршрут доступа к графическому файлу, элемент будет демонстрировать соответствующее изображение, щелчок на котором позволит перейти на страницу, определяемую свойством `NavigateURL`.

Элемент XML

Элемент XML предназначен для отображения XML-документов в соответствии с указанным XSL-стилем. Отображаемый документ задается свойством `DocumentSource`, XSL-файл — свойством `TransformSource`.

11 Проверка данных

В интерактивных системах важную роль приобретает контроль над возможными действиями пользователя. Типичный пример — пользователь щелкнул на кнопке вызова другой формы, оставив незаполненными поля, предназначенные для обязательного заполнения (для регистрации, например). В категории Web Controls есть группа элементов, предназначенных для проверки данных:

- `RequiredFieldValidator` — проверяет, введены ли в элементе управления данные при потере им фокуса ввода;
- `RangeValidator` — контролирует попадание введенного значения в заданный диапазон;
- `RegularExpressionValidator` — проверяет соответствие введенных данных заданной маске или регулярному выражению;
- `CompareValidator` — сравнивает введенное значение с заданным;
- `CustomValidator` — позволяет программисту написать собственный код для проверки данных;
- `ValidationSummary` — отображает сводку обо всех ошибках ввода.

В этой главе рассматриваются перечисленные компоненты.

Две формы проверки

Существует две формы проверки — на стороне сервера и на стороне клиента. Проверка на стороне сервера требует дополнительного трафика, так как данные отсылаются серверу, который проверяет их и возвращает клиенту результат проверки. В случае клиентской проверки в HTML-страницу вставляется сценарий на языке JavaScript. Этот сценарий интерпретируется клиентским браузером, который и принимает решение о корректности данных. Клиентская проверка снижает загрузку сети и значительно сокращает время реакции на ввод данных, однако не

все браузеры могут интерпретировать сценарии JavaScript, к тому же разные версии одного и того же браузера поддерживают разные версии этого языка, что не всегда гарантирует правильную работу.

Разработчики ASP.NET разработали «умные» элементы проверки, которые генерируют как клиентский, так и серверный код. Если браузер клиента поддерживает JavaScript, ему автоматически пересылается нужный сценарий, в противном случае проверка реализуется сервером.

Следует помнить, что клиентская проверка данных работает только в некоторых браузерах, в частности в Internet Explorer версии 4.0 и выше.

Настройка клиентской проверки

Элементы проверки данных в случае клиентской проверки используют сценарии, хранящиеся в файле WebUIValidation.js. Этот файл по умолчанию устанавливается в папку Inetpub\wwwroot\aspnet_client\system_web\1_1_4322 сервера при установке на машине программного обеспечения технологии ASP.NET. Если файл перенести в другую папку, нужно соответствующим образом откорректировать файл machine.config в папке Windows\Microsoft.NET\Framework\v1.1.432\Config.

В папке Windows\Microsoft.NET\Framework\v1.1.432 хранится утилита aspnet_regiis.exe, предназначенная для установки/удаления библиотечного файла WebUIValidation.js. Эта утилита запускается из командной строки с ключом `-c` для установки библиотеки и с ключом `-e` для ее удаления.

Отключение клиентской проверки

Если по каким-либо причинам необходимо отключить клиентскую проверку веб-страницы, в ее начало следует поместить следующую директиву:

```
<%@ Page ClientTarget="downlevel" %>
```

Технология ASP.NET позволяет отключать клиентскую проверку не для всей страницы, а лишь для некоторых элементов проверки: каждый такой элемент имеет логическое свойство `EnabledClientScript`, в которое в этом случае следует поместить значение `False`.

Элемент RequiredFieldValidator

Элемент `RequiredFieldValidator` проверяет, введены ли необходимые данные в контролируемый им элемент. Как правило, он используется для контроля над содержимым полей `TextBox`, но его можно использовать и с любым другим элементом, способным получать фокус ввода.

Свойства элемента представлены в табл. 11.1.

Элемент имеет метод `Validate`, который осуществляет проверку и обновляет свойство `IsValid`. Программный вызов этого метода обычно не требуется, так как он вызывается автоматически при попытке смены страницы.

Таблица 11.1. Свойства элемента RequiredFieldValidator

Свойство	Назначение
`property ControlToValidate: String;`	Содержит имя контролируемого элемента
`property Display: ValidatorDisplay;`	Определяет место вывода сообщения об ошибке, содержащегося в свойстве `Text`: `Static` — текст появляется на месте размещения элемента; `Dynamic` — текст появляется на свободном месте страницы; `None` — текст не появляется
`property EnabledClientScript: Boolean;`	Разрешает или запрещает клиентскую проверку
`property Enabled: Boolean;`	Разрешает или запрещает любую проверку
`property ErrorMessage: String;`	Содержит сообщение, которое выводится в элементе `ValidationSummary`, если свойство `Text` не установлено
`property InitValue: String;`	Содержит начальное значение контролируемого элемента
`property IsValid: Boolean;`	Содержит `True`, если проверка корректности данных прошла успешно
`property Text: String;`	Определяет сообщение об ошибке

В следующем примере (файл Ch11\RequiredFieldValidater\WebForm1.aspx) на форме размещаются два поля и кнопка вызова другой станицы (рис. 11.1).

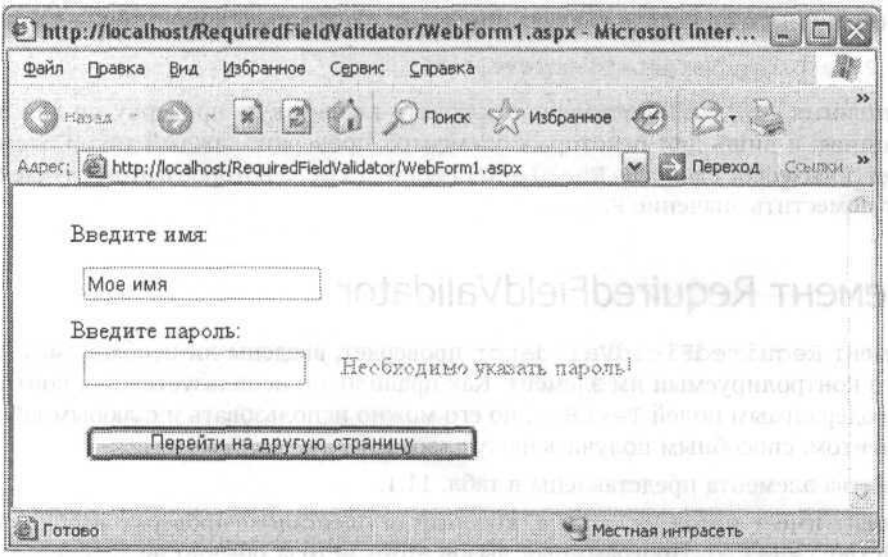


Рис. 11.1. Демонстрация элемента RequiredFieldValidator

На форме есть также два элемента RequiredFieldValidator, каждый из которых контролирует содержимое одного из полей. В обработчике щелчка на кнопке анализируется результат проверки:

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  if RequiredFieldValidator1.IsValid and
                    RequiredFieldValidator2.IsValid then
    Response.Redirect('WebForm2.aspx');
end;
```

В каждом приложении ASP.NET определен объект Response, содержащий ответ на запрос клиента. Метод Redirect этого объекта осуществляет переадресацию запроса на другую страницу. Помимо объекта Response в ASP.NET определены также объекты Request, Server, Application и Session.

Элемент RangeValidator

Элемент RangeValidator позволяет проверить, попадает ли введенное пользователем значение в заданный диапазон. В качестве границ диапазона могут использоваться числа, даты, денежные суммы и строки. Свойства MinimumValue и MaximumValue хранят границы диапазона. Свойство Type определяет тип данных и может иметь значение Integer, Double, Currency, Date или String. Остальные свойства элемента такие же, как у элемента RequiredFieldValidator.

ПРИМЕЧАНИЕ

Дата может вводиться только в формате 20/1/2005 или 20-1-2005.

Рисунок 11.2 иллюстрирует использование элемента RangeValidator (файл Ch11\RangeValidator\WebForm1.aspx).

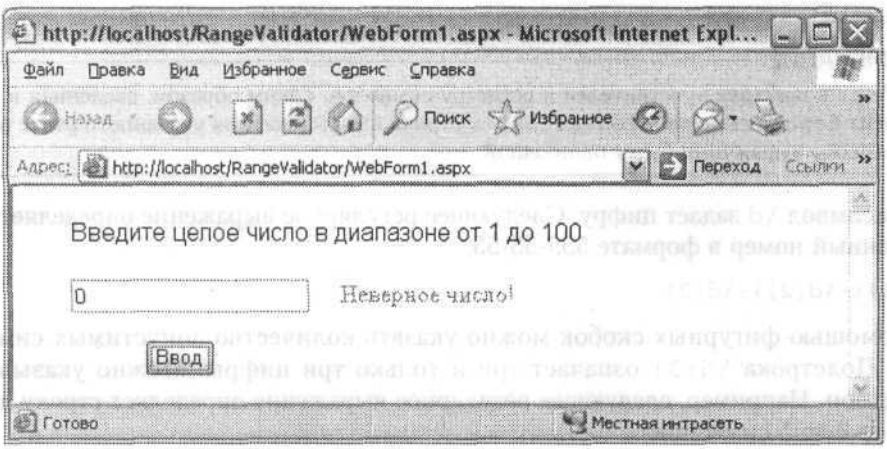


Рис. 11.2. Пример использования элемента RangeValidator

Элемент RegularExpressionValidator

Элемент `RegularExpressionValidator` сравнивает введенные пользователем данные с некоторым регулярным выражением. В качестве такового используется строка, задающая шаблон ввода. Эта строка помещается в свойство `ValidationExpression` элемента. Остальные свойства элемента такие же, как у элемента `RequiredFieldValidator`.

В простейшем случае регулярное выражение — это просто строка. В этом случае ввод пользователя должен просто повторить эту строку. Такая проверка, например, применяется для контроля правильности введенного пароля (см. рис. 11.1). В этом примере (файл **Ch11\RegularExpessionValidator\WebForm1.aspx**) свойство `ValidationExpression` содержит строку `Delphi`. Более сложные проверки реализуются вставкой в строку шаблона специальных метасимволов.

ПРИМЕЧАНИЕ

В окне инспектора объектов в строке свойства `ValidationExpession` имеется кнопка, щелчок на которой открывает список стандартных для ASP.NET шаблонов.

Метасимвол `\w` определяет любой *словообразующий символ* (латинскую букву, знак подчеркивания или цифру). Метасимвол `\w+` определяет любое количество словообразующих символов. Следующее регулярное выражение определяет произвольный латинский текст:

```
(\w|\s)+
```

Здесь метасимвол `\s` задает пробел, метасимвол `|` означает выбор одного из двух метасимволов, а скобки группируют метасимволы, так что метасимвол `+` относится к заключенному в скобки выражению. Показанное ниже выражение определяет произвольный латинский текст, начинающийся символами `begin` и заканчивающийся символами `end`:

```
begin(\w|\s)+end
```

ПРИМЕЧАНИЕ

Текст в шаблоне чувствителен к регистру символов. Таким образом, введенная в элемент `RegularExpressionValidator` строка `BEGIN_end` для указанного ранее регулярного выражения будет ошибочной.

Метасимвол `\d` задает цифру. Следующее регулярное выражение определяет телефонный номер в формате 555-55-55:

```
\d{3}\-\d{2}\-\d{2}
```

С помощью фигурных скобок можно указать количество допустимых символов. Подстрока `\d{3}` означает три и только три цифры. Можно указывать диапазон. Например, следующее регулярное выражение определяет строку длиной от 2 до 5 символов:

```
\w{2,5}
```

ПРИМЕЧАНИЕ

Элемент `RegularExpressionValidator` никак не реагирует на отсутствие данных, поэтому его обычно дополняют элементом `RequiredFieldValidator`.

Элемент CompareValidator

Элемент `CompareValidator` выполняет сравнение данных в поле формы с другими данными. Сравниваемыми данными могут быть числа, даты, денежные суммы, строки. Помимо свойств элемента `RequiredFieldValidator` у элемента `CompareValidator` есть свойства `ControlToCompare`, `ValueToCompare`, `Operator` и `Type`. Первое определяет второй элемент с данными: сравнение осуществляется между ним и элементом `ControlToCompare`. Например, при вводе диапазона дат начальная дата не должна превышать конечную. Свойство `Operator` задает операцию сравнения и может иметь одно из следующих значений: `Equal` — равно, `NotEqual` — не равно, `GreaterThen` — больше, `GreaterThenEqual` — больше или равно, `LessThen` — меньше, `LessThenEqual` — меньше или равно, `CompareDateCheck` — проверить правильность даты. Свойство `Type` устанавливает тип сравниваемых данных и может иметь значение `Integer`, `Double`, `Currency`, `Date` или `String`.

ПРИМЕЧАНИЕ

Дата может вводиться только в формате 20/1/2005 или 20-1-2005.

На рис. 11.3 показано окно программы, в которой элемент `CompareValidator` используется для сравнения дат (файл **Ch11\CompareValidator\WebForm1.aspx**). Специфичные для элемента свойства имеют такие значения: `ControlToValidate = TextBox1` (начальная дата), `ControlToCompare = TextBox2`, `Operator = GreaterTnenEqual`, `Type = Date`.

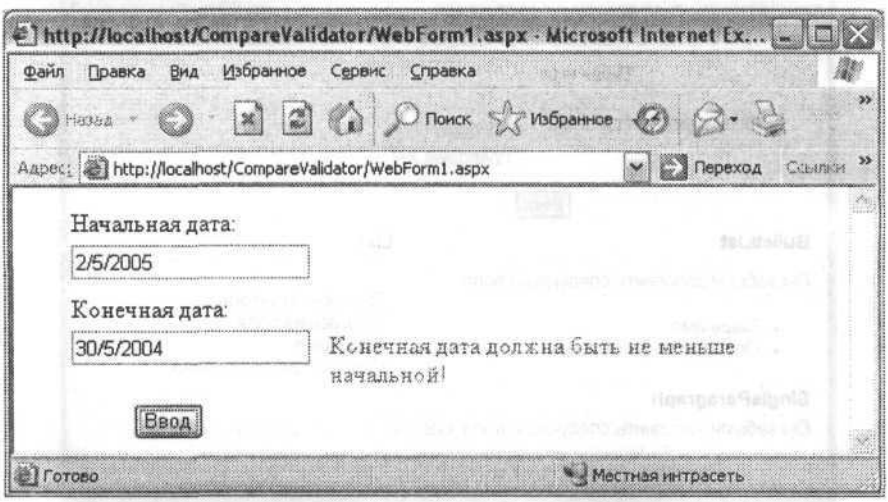


Рис. 11.3. Демонстрация элемента CompareValidator

Элемент ValidationSummary

Элемент `ValidationSummary` создает отчет обо всех ошибках в данных веб-формы. Его особенно удобно применять на громоздких формах с множеством элементов контроля. Специфические свойства элемента перечислены в табл. 11.2.

Таблица 11.2. Свойства элемента ValidationSummary

Свойство	Назначение
`property DisplayMode: ValidationSummaryDisplayMode;`	Устанавливает формат сообщений об ошибках: `BulletList` — в маркированном списке; `List` — в простом списке; `SingleParagraph` — в одном абзаце
`property EnabledClientScript: Boolean;`	Разрешает/запрещает клиентскую проверку данных
`property Enabled: Boolean;`	Разрешает/запрещает любую проверку
`property HeaderText: String;`	Определяет заголовок отчета
`property ShowMessageBox: Boolean;`	Если содержит `True`, отчет выводится во всплывающем окне
`property ShowSummary: Boolean;`	Разрешает/запрещает вывод отчета

На рис. 11.4 показаны все три формата отчета об ошибках, устанавливаемые свойством `DisplayMode` (файл Ch11\ValidationSummary\WebForm1.aspx).

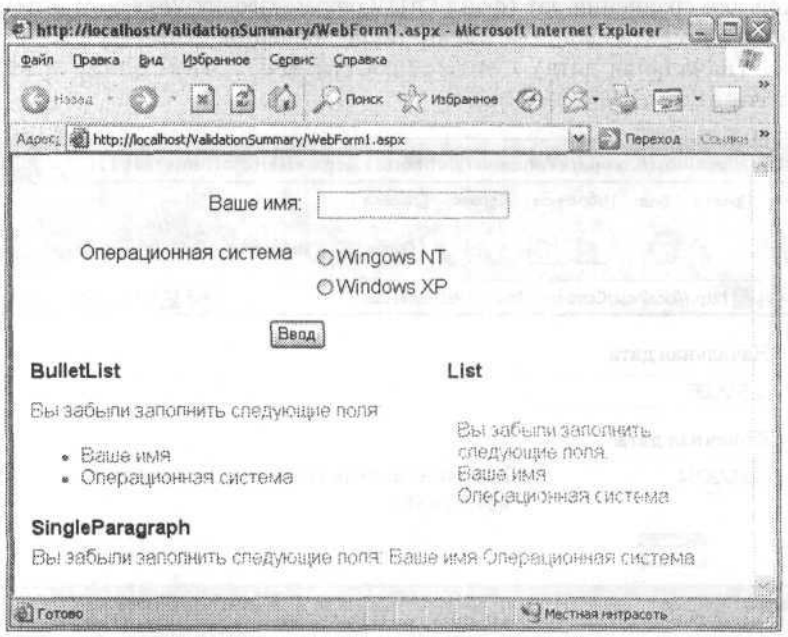


Рис. 11.4. Возможные форматы отчета

На рис. 11.5 показано всплывающее окно с отчетом об ошибках (свойство ShowMessageBox элемента имеет значение True).

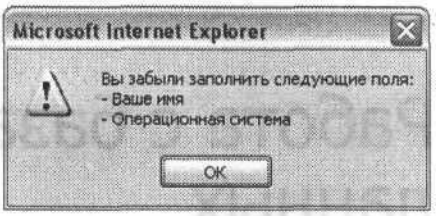


Рис. 11.5. Всплывающее окно с отчетом

Элемент CustomValidator

Элемент CustomValidator позволяет программисту самому организовать проверку данных как на стороне сервера, так и в клиентском браузере. Для этого в первом случае он должен написать обработчик события ServerValidate, а во втором — создать сценарий на языке JavaScript (напомню, что этот сценарий интерпретирует браузер клиента, который, в отличие от IIS, не обязан «понимать» язык C# или Visual Basic .NET). Как обработчик, так и функция JavaScript принимают два параметра: первый содержит объект CustomValidator и имеет тип Object, а второй инкапсулирует связанные с событием параметры в виде объекта класса ServerValidateEventArgs. Этот класс имеет два свойства: логическое IsValid и строковое Value. Обработчик анализирует свойство Value и возвращает результат в свойстве IsValid.

В следующем примере (файл Ch11\CustomValidator\WebForm1.aspx) используется такой обработчик:

```
procedure TWebForm1.CustomValidator1_ServerValidate(
          source: System.Object; args:  ServerValidateEventArgs);
var
  k: Integer;
begin
  k := Int32.Parse(args.Value);
  args.IsValid := not odd(k);
end;
```

Результат показан на рис. 11.6.

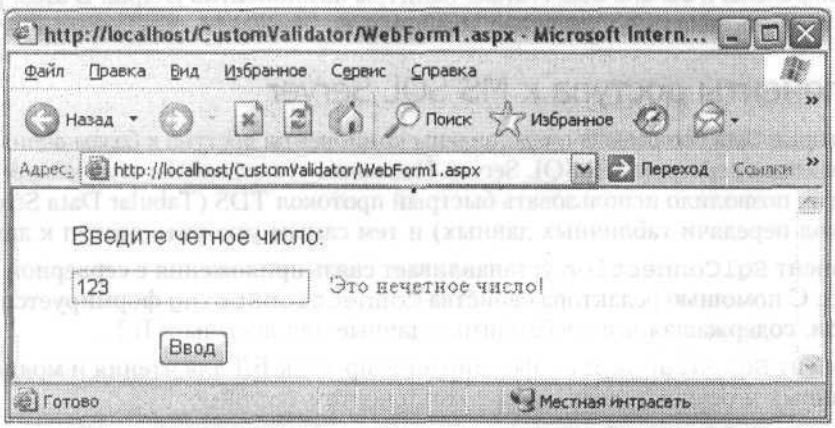


Рис. 11.6. Демонстрация элемента CustomValidator

12 Работа с базами данных

Публикация баз данных в Интернете является одной из наиболее интересных задач, решаемых с помощью технологии ASP.NET. В Delphi 2005 есть возможность использовать как стандартные для .NET Framework средства работы с данными, так и средства, разработанные корпорацией Borland. Последние сосредоточены в категориях Borland Data Provider и DB Web палитры компонентов. В этой главе рассматриваются обе группы средств.

Доступ к данным

Доступ к данным в рамках приложений ASP.NET реализуется с помощью технологии ADO.NET, достаточно подробно рассмотренной в главе 2. При создании приложений ASP.NET компоненты этой технологии сосредоточены в категориях Data Components и Borland Data Provider палитры компонентов Delphi. В этом разделе кратко описываются указанные компоненты.

Компоненты доступа к MS SQL Server

В категории Data Components представлены компоненты доступа к базам данных, обслуживаемым сервером MS SQL Server. Напомню, что отделение этих компонентов от других позволило использовать быстрый протокол TDS (Tabular Data Stream — протокол передачи табличных данных) и тем самым ускорить доступ к данным.

Компонент `SqlConnection` устанавливает связь приложения с серверной базой данных. С помощью редактора свойства `ConnectionString` формируется строка связи, содержащая все необходимые данные для доступа к БД.

Компонент `SqlDataAdapter` формирует запросы к БД для чтения и модификации данных и осуществляет связь с компонентом `DataSet`.

Компонент `SqlCommand` используется для выполнения транзакций или хранимых процедур.

Компоненты доступа к другим источникам данных

Для доступа к другим источника данных используются компоненты BdpConnection, BdpDataAdapter и DataSet. Связной компонент BdpConnection позволяет взаимодействовать с промышленными серверами MS SQL Server, DB2, Oracle, Sybase, InterBase, а также с таблицами MS Access. Адаптер BdpDataAdapter формирует SQL-запросы к одной или нескольким таблицам БД, представлениям или хранимым процедурам. Этот компонент формирует все четыре SQL-запроса, которые хранятся в его свойствах DeleteCommand, InsertCommand, SelectCommand и UpdateCommand. При представлении данных на веб-страницах часто бывает удобнее вместо адаптера применять компонент BdpCommand, который реализует единственную команду, хранящуюся в свойстве CommandText. При этом, если команда возвращает данные, используется его метод ExecuteReader или ExecuteScalar, в противном случае — ExtcuteNonQuery. Набор данных (класс DataSet) предоставляет удобный механизм чтения и обновления данных и инкапсулирует в себе в общем случае множество таблиц и связей между ними. При необходимости можно включить еще два компонента: ODBCConnection и ODBCDataAdapter. Эти компоненты рассчитаны на доступ к данным по технологии ODBC, но по умолчанию не устанавливаются. Для их установки следует выбрать команду Components ▸ Install .NET Components, разыскать их в списке компонентов и установить соответствующие флажки. После закрытия окна компоненты появятся в категории Data Components.

Визуализация данных

Визуализировать данные, хранимые в компоненте DataSet, можно либо с помощью стандартных компонентов категории Web Controls, либо с помощью компонентов категории DB Web. Последние разработаны корпорацией Borland и, как показывает практика, намного удобнее стандартных. Для их связи с наборами данных DataSet используются компоненты-клапаны DBWebDataSource, играющие такую же роль, как и компоненты TDataSource в VCL-приложениях.

Простой пример

Рассмотрим пример, в котором создается окно, показанное на рис. 12.1 (файл Ch12\DataPublicDemo\WebForm1.aspx).

Как видно из рисунка, программа публикует таблицу BOOKS из БД «Книголюб».

1. Начните новый проект ASP.NET. Если вы еще не создавали соединение с БД C:\BIBLDATA\IB_BIBL.GDB, создайте его. Для этого в правой верхней панели окна менеджера проектов щелкните на вкладке со значком Data Explorer для вызова окна одноименной утилиты. В этом окне щелкните правой кнопкой на узле Interbase и в контекстном меню выберите команду Add New Connection. В появившемся окне введите имя нового соединения, например IBConn. После этого раскройте узел Interbase, щелкните правой кнопкой мыши на новом соединении,

в контекстном меню выберите команду Modify Connection и настройте соединение так, как показано на рис. 12.2.

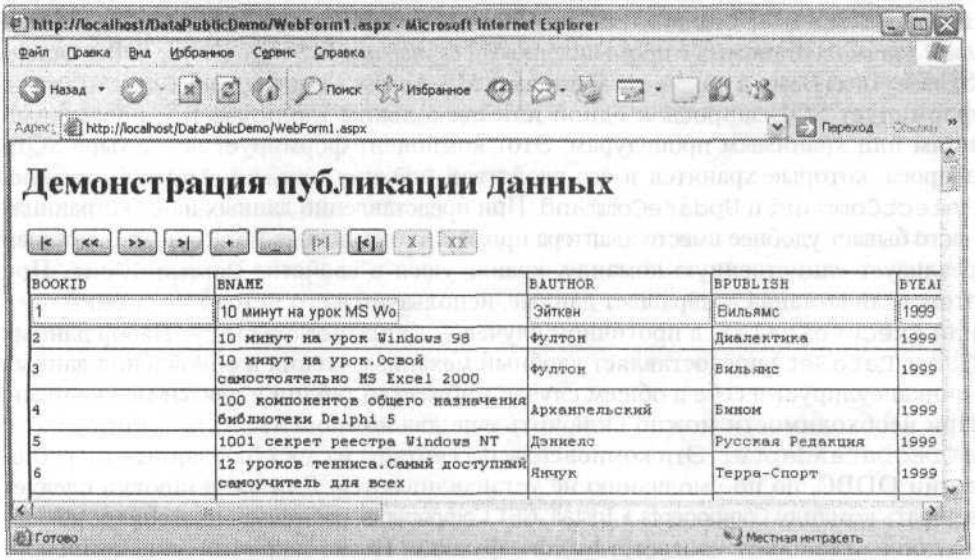


Рис. 12.1. Окно работающей программы

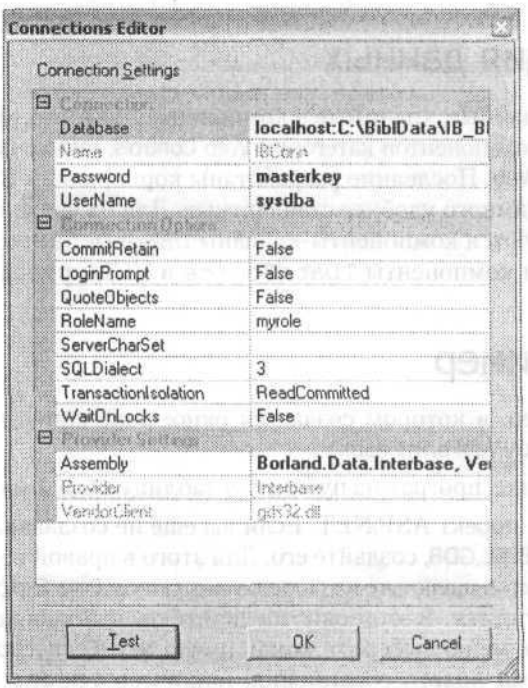


Рис. 12.2. Настройка соединения с БД «Книголюб»

ВНИМАНИЕ

При работе с таблицами InterBase в технологии ASP.NET маршруту доступа к файлу БД должен предшествовать префикс с DNS-именем узла сети, на котором запущен сервер InterBase, например: `localhost:C:\Data\MyData.gdb`. Если префикс не указать, то на этапе редактирования формы сетка будет заполняться данными, но при прогоне программы появится сообщение Unavailable database (База данных недоступна).

2. В окне Data Explorer раскройте узел соединения с БД «Книголюб», затем раскройте узел Tables и перетащите таблицу `BOOKS` в окно формы. В нижней части формы появятся два компонента: связной компонент `BdpConnection1`, настроенный на связь с БД IB_BIBL.GDB, и адаптер `BdpDataAdapter1`, настроенный на отображение всех полей таблицы `BOOKS`.
3. Присоедините к ним компонент `DataSet` (категория Data Components).
4. В свойство `DataSet` адаптера поместите ссылку на компонент `DataSet1` и активизируйте адаптер (`Active = True`).
5. Присоедините к проекту компонент `DBWebDataSource` (категория DB Web) и в его свойство `DataSource` поместите ссылку на таблицу `BOOKS`.
6. Поместите на форму компоненты `DBWebNavigator` и `DBWebGrid` (категория DB Web) и настройте их: в свойства `DataSource` поместите ссылки на компонент `DBWebDataSource1`, а в свойства `TableName` — ссылки на таблицу `BOOKS`. Сетка должна наполниться данными.

Итак, мы решили довольно сложную задачу, не написав ни одной строчки кода[1]! Как мы увидим дальше, использование классических элементов ASP.NET из категорий Data Components и Web Controls для решения той же задачи требует значительно бóльших усилий. Другим немаловажным преимуществом компонентов категории DB Web является то, что они «живут» (наполняются данными) не только на этапе прогона программы, но и на этапе конструирования формы.

Стандартные элементы для работы с базами данных

В этом разделе рассматриваются возможности стандартных серверных элементов применительно к базам данных. Компоненты категории DB Web описаны в разделе «Элементы категории DB Web».

[1] На самом деле это не так: если вы запустите проект, то увидите, что в компоненте `DBWebNavigator` кнопка подтверждения сделанных изменений недоступна. Чтобы полностью реализовать все возможные действия (вставка, удаление и изменение записей), следует определить обработчик события `DBWebDataSource.OnApplyChangesRequest` (подробнее см. раздел «Элемент DBWebDataSource»).

Привязка данных

Все компоненты технологии ASP.NET унаследованы от класса `Control`, который передает своим потомкам метод `DataBind`. Этот метод осуществляет так называемую *привязку данных*, представляющую собой динамическую установку значения некоторого свойства элемента управления. Таким образом, любой из рассмотренных в главе 11 серверных элементов управления можно при желании связать с данными из БД или XML-файла.

ПРИМЕЧАНИЕ

На веб-странице может находиться множество элементов управления, связанных с данными. Чтобы не обращаться к методу `DataBind` каждого элемента, вызовите этот метод для страницы, которая также является наследником класса `Control`. Связывание страницы с данными приведет к автоматическому связыванию всех расположенных на ней элементов управления.

В качестве примера рассмотрим программу, которая создает окно, показанное на рис. 12.3 (файл Ch12\DataBindingDemo\WebForm1.aspx).

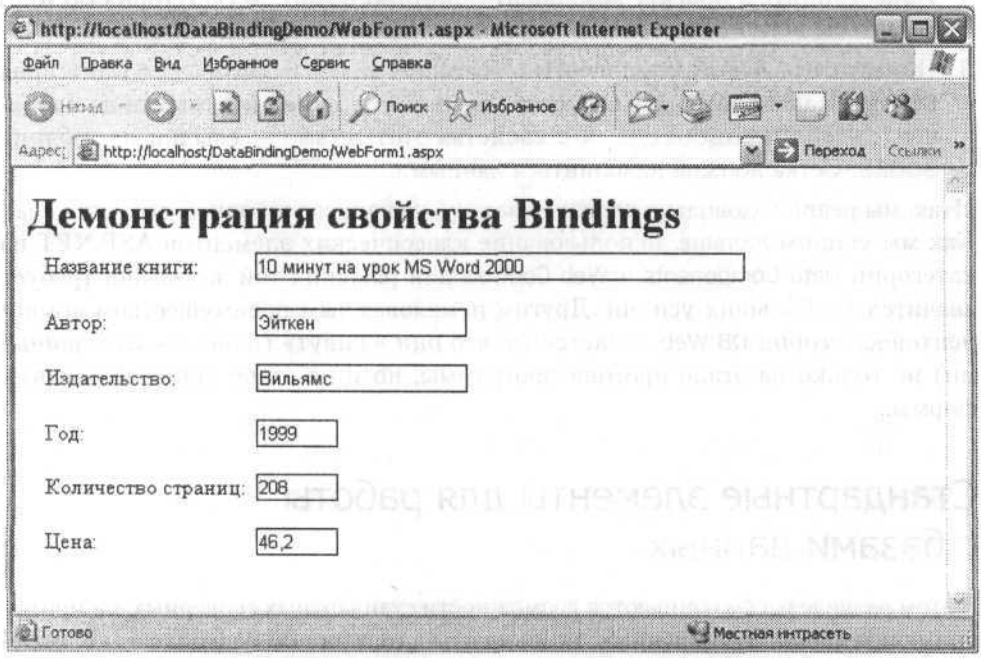


Рис. 12.3. Отображение данных в элементах TextBox

В шести элементах `TextBox` этого окна отображаются данные из шести полей первой записи таблицы `BOOKS` базы данных IB_BIBL.GDB. С веб-страницей связаны не показанные на рисунке компоненты `BdpConnection`, `BdpDataAdapter` и `DataSet`. Для каждого элемента `TextBox` щелчок на кнопке свойства `DataBindings` открывает окно, показанное на рис. 12.4.

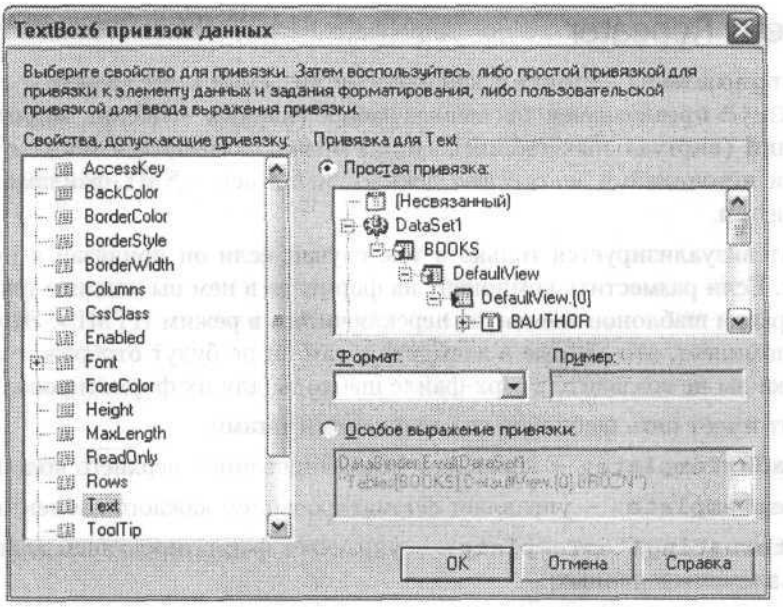


Рис. 12.4. Окно редактора свойства DataBindings

В левом списке этого окна выбирается свойство, допускающее привязку, а в правом — поле с данными. Чтобы наполнить поля ввода данными, используется такой обработчик:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
begin
  if not Self.IsPostBack then
    Self.DataBind;
end;
```

Проверка свойства `IsPostBack` приложения делается для того, чтобы данные в поля ввода загружались единственный раз — при первой загрузке формы.

ПРИМЕЧАНИЕ

Любое приложение ASP.NET автоматически снабжается обработчиком события `Load`, которое возникает при загрузке страницы. Он доступен в pas-файле и по умолчанию ничего не делает.

Программа носит чисто иллюстративный характер, поэтому возможное изменение данных в полях никак не влияет на собственно данные в таблице. Если бы мы захотели действительно изменять табличные данные, нам нужно было бы предусмотреть кнопку, обработчик щелчка на которой вызывал бы метод `AcceptChanges` набора данных `DataSet`. Еще одна особенность программы — в ней не предусмотрены средства навигации по таблице. Проще всего решить эту задачу можно с помощью компонентов категории DB Web, которые рассматриваются в разделе «Элементы категории DB Web».

Элемент Repeater

Элемент управления `Repeater`, как и рассматриваемые далее элементы `DataList` и `DataGrid`, предназначен для вывода наборов данных — таблиц, запросов, представлений (виртуальных таблиц). Но он может связываться также с любыми другими источниками *множества* данных, например с XML-файлами, коллекциями и т. п.

Элемент визуализируется только в том случае, если он привязан к источнику данных. Если разместить компонент на форме, то в нем вы увидите такой текст: «Для правки шаблонов элементов переключитесь в режим HTML». Это сообщение напоминает, что данные в элементе и сам он не будут отображаться до тех пор, пока вы не создадите в aspx-файле шаблоны для их форматирования.

Элемент имеет пять шаблонов со следующими тегами:

- **<HeaderTemplate>** — управляет форматированием верхнего колонтитула;
- **<ItemTemplate>** — управляет форматированием каждого элемента данных;
- **<AlternatingItemTemplate>** — управляет форматированием альтернативных элементов данных;
- **<SeparatorTemplate>** — управляет разделителем между элементами данных;
- **<FooterTemplate>** — управляет форматированием нижнего колонтитула.

В типичном варианте вывода данных в форме таблицы в верхнем колонтитуле содержится тег объявления таблицы **<table>** и теги **<tr>**, **<th>**, **</th>**, **</tr>**, определяющие форматирование шапки (первой строки) таблицы. В шаблоне **<ItemTemplate>** размещаются теги **<tr>**, **<td>**, **</td>**, **</tr>**, определяющие обычную строку таблицы. Наконец, в шаблоне **<FooterTemplate>** размещается тег окончания таблицы **</table>**. Для привязки данных в обычной строке таблицы между тегами **<td>** и **</td>** помещаются специальные теги привязки:

```
<%#Container.DataItem("ИмяПоля")%>
```

Здесь *ИмяПоля* — имя поля таблицы базы данных.

ПРИМЕЧАНИЕ

Если при прогоне программы источник данных по каким-либо причинам недоступен, отладчик ASP.NET выдает неверную диагностику: он выделяет строку с тегами привязки и заявляет, что эта строка ссылается на свойство, в то время как требуется метод (рис. 12.5).

Должен сказать, что использовать элемент `Repeater` в приложениях Delphi непросто. Мне, например, не удалось добиться его нормальной работы по обычной схеме: поместить в свойство `DataSource` ссылку на набор данных и в обработчике `Page_Load` вызвать его метод `DataBind`. В следующем примере (файл Ch12\RepeaterDemo\WebForm1.aspx) элемент выводит три поля таблицы `Authors` базы данных Pubs (демонстрационная БД, поставляемая с сервером MS SQL Server). Окно программы показано на рис. 12.6.

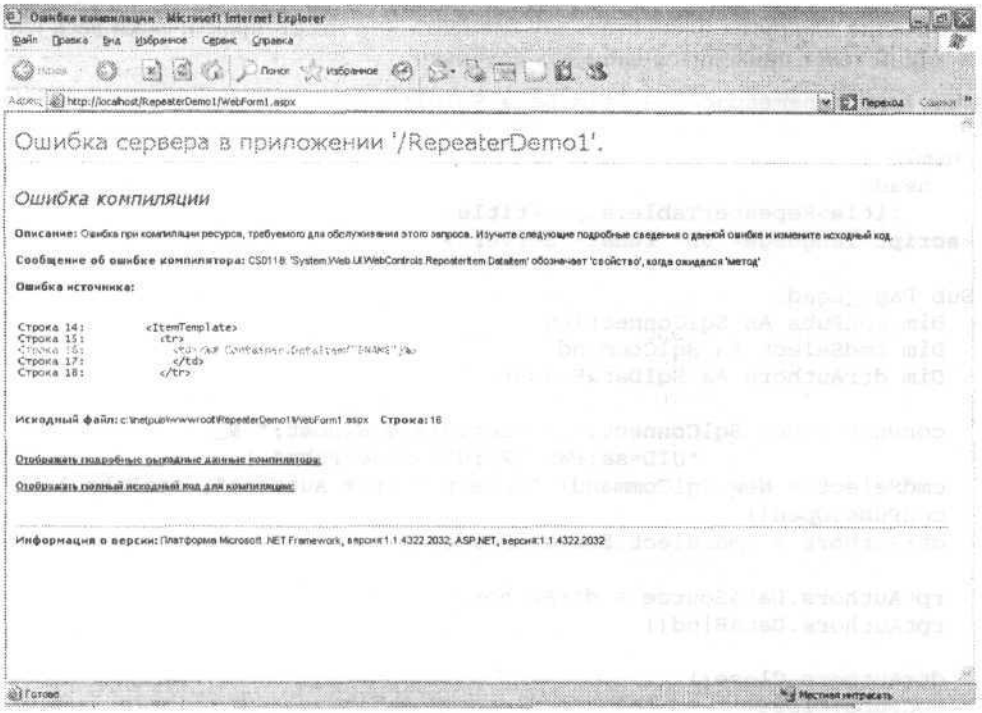


Рис. 12.5. Сообщение отладчика

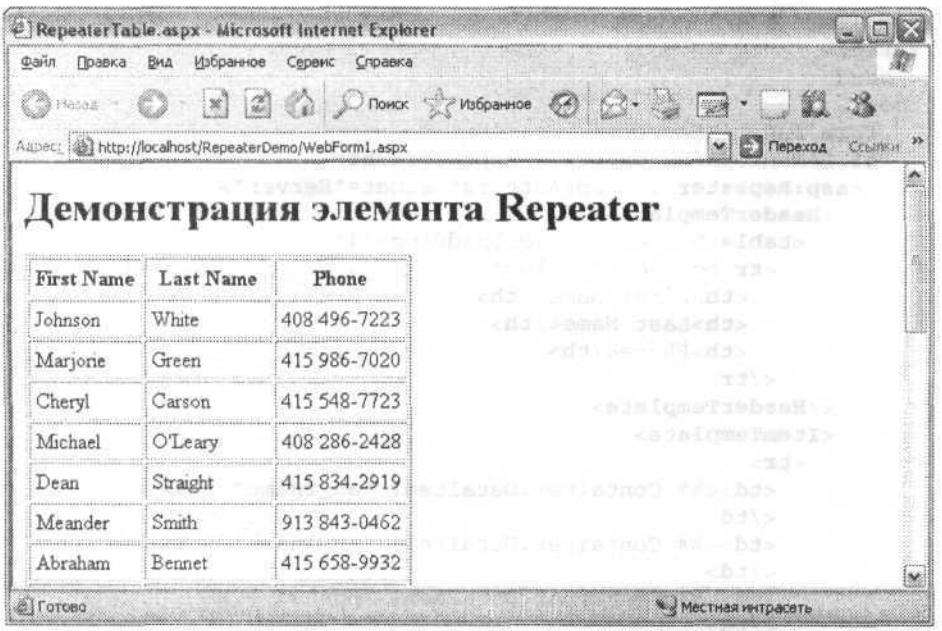


Рис. 12.6. Демонстрация элемента Repeater

В отличие от других элементов, для элемента Repeater практически весь следующий текст приходится набирать вручную:

```
<%@ Import namespace="System.Data.SqlClient"%>

<html>
  <head>
    <title>RepeaterTable.aspx</title>
<script language="VB" runat="Server">

Sub Page_Load
  Dim conPubs As SqlConnection
  Dim cmdSelect As SqlCommand
  Dim dtrAuthors As SqlDataReader

  conPubs = New SqlConnection( "Server=localhost;" &_
                    "UID=sa;PWD=123;Database=Pubs" )
  cmdSelect = New SqlCommand( "Select * From Authors", conPubs )
  conPubs.Open()
  dtrAuthors = cmdSelect.ExecuteReader()

  rptAuthors.DataSource = dtrAuthors
  rptAuthors.DataBind()

  dtrAuthors.Close()
  conPubs.Close()
End Sub


</script>
  </head>
  <body>
    <form runat="Server">
    <h1>Демонстрация элемента Repeater</h1>
      <asp:Repeater id="rptAuthors" runat="Server">
        <HeaderTemplate>
          <table border="1" cellpadding="4">
            <tr bgcolor="yellow">
              <th>First Name</th>
              <th>Last Name</th>
              <th>Phone</th>
            </tr>
        </HeaderTemplate>
        <ItemTemplate>
          <tr>
            <td><%# Container.DataItem( "au_fname" ) %>
            </td>
            <td><%# Container.DataItem( "au_lname" ) %>
            </td>
            <td><%# Container.DataItem( "phone" ) %>
            </td>
          </tr>
```

```
        </ItemTemplate>
        <AlternatingItemTemplate>
          <tr bgcolor="yellow">
            <td><%# Container.DataItem( "au_fname" ) %>
            </td>
            <td><%# Container.DataItem( "au_lname" ) %>
            </td>
            <td><%# Container.DataItem( "phone" ) %>
            </td>
          </tr>
        </AlternatingItemTemplate>
        <FooterTemplate>
          </table>
        </FooterTemplate>
      </asp:Repeater>
    </form>
  </body>
</html>
```

В обработчике `Page_Load` данные привязываются к элементу динамически. Сначала создается соединение `conAuthors` класса `SqlConnection` — в строке соединения указаны пользователь `sa` и пароль `123` (как указать нужный пароль, описано в разделе «Работа с транзакциями» главы 4). Затем создается объект `cmdSelect` класса `SqlCommand`, в который помещается запрос, выбирающий все записи из таблицы `Authors`. Наконец, создается объект `dtrAuthors` класса `SqlDataReader`, который и устанавливается как источник данных для элемента `rptAuthors`.

Элемент DataList

Элемент `DataList` предназначен для вывода множества данных в виде списка. Подобно элементу `Repeater`, он имеет шаблоны, управляющие форматированием отдельных его частей. Таким образом, объем ручного кодирования в нем приблизительно такой же, как и в случае элемента `Repeater`. В отличие от последнего, он имеет множество свойств оформительского характера, позволяющих указать заголовок, шрифт, цвет, наличие разделительных линий и т. д.

В следующем примере (файл Ch12\DataListDemo\WebForm1.aspx) в элементе `DataList` выводится содержимое поля `Title` таблицы `Titles` из БД Pubs (рис. 12.7).

Как и в примере RepeaterDemo.aspx, практически весь следующий текст нужно набирать вручную:

```
<%@ Import namespace="System.Data.SqlClient"%>

<html>
  <head>
    <script language="VB" runat="Server">
  Sub Page_Load
  Dim conPubs As SqlConnection
  Dim cmdSelect As SqlCommand
  Dim dtrTitles As SqlDataReader
```

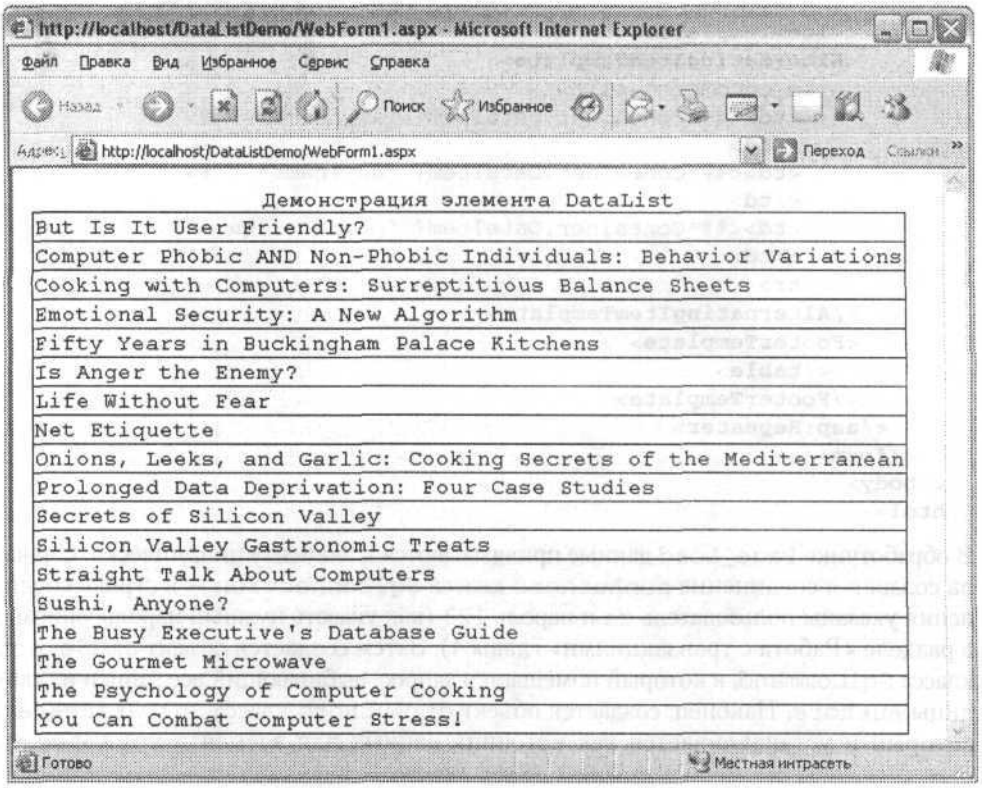


Рис. 12.7. Демонстрация элемента DataList

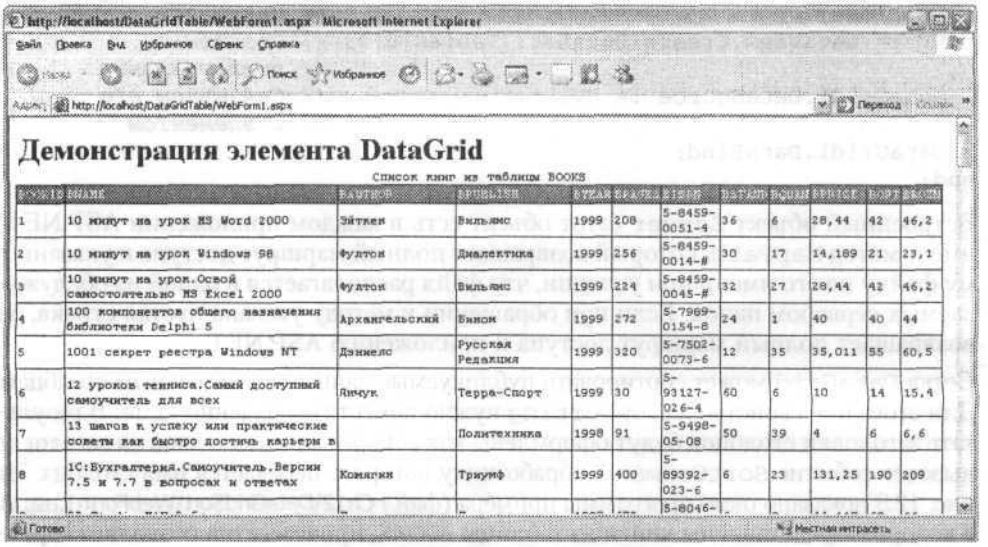
```
  conPubs = New SqlConnection( "Server=localhost;UID=sa;" &_
                               "PWD=123;Database=Pubs" )
  cmdSelect = New SqlCommand( "Select Title From Titles", conPubs )
  conPubs.Open()
  dtrTitles = cmdSelect.ExecuteReader()

  dlstTitles.DataSource = dtrTitles
  dlstTitles.DataBind()

  dtrTitles.Close()
  conPubs.Close()
End Sub
    </script>
  </head>
  <body ms_positioning="GridLayout">
    <form runat="server">
      <asp:DataList id="dlstTitles"
        style="Z-INDEX: 1; LEFT: 14px; POSITION: absolute;
        TOP: 14px"  runat="server" caption=
         "Демонстрация элемента DataList"
```

```
        font-names="Courier New" borderstyle="Solid" borderwidth=
        "1px" bordercolor="Black" backcolor="#FFFF80"
        gridlines="Horizontal">
        <ItemTemplate><%# Container.DataItem("Title")%>
        </ItemTemplate>
      </asp:DataList>
    </form>
  </body>
</html>
```

Поскольку обработчик `Page_Load` практически такой же, как и примере RepeaterDemo.aspx, я не буду его комментировать. Обратите внимание на многочисленные атрибуты тега определения элемента. Все они созданы в окне инспектора объектов на стадии редактирования формы — это безусловное достоинство элемента. Кроме того, у элемента в окне инспектора объектов определены свойства `HeaderStyle`, `ItemStyle`, `AlternatingItemStyle` и т. д., которые дают доступ к соответствующим шаблонам и позволяют установить свойства шаблонов еще на этапе редактирования.

Элемент DataGrid

Элемент `DataGrid` предназначен для вывода данных из наборов данных или XML-файлов в виде сетки. В отличие от рассмотренных ранее элементов `Repeater` и `DataList`, его умалчиваемые свойства таковы, что позволяют свести объем ручного кодирования к минимуму. В следующем примере (файл Ch12\DataGridTable\WebForm1.aspx) в сетке выводятся данные из таблицы `BOOKS` БД «Книголюб» (рис. 12.8).



Демонстрация элемента DataGrid

Список книг из таблицы BOOKS

BOOKID	NAME	AUTHOR	PUBLISH	BYEAR	BPAGES	BISBN	BPRICE	BQUAN	BPRICE	BQNT	BSUM
1	10 минут на урок MS Word 2000	Зиткев	Вильямс	1999	208	5-8459-0051-4	36	6	28,44	42	46,2
2	10 минут на урок Windows 98	Фултон	Диалектика	1999	256	5-8459-0014-#	30	17	14,189	21	23,1
3	10 минут на урок.Освой самостоятельно MS Excel 2000	Фултон	Вильямс	1999	224	5-8459-0045-#	32	27	28,44	42	46,2
4	100 компонентов общего назначения библиотеки Delphi 5	Архангельский	Бином	1999	272	5-7989-0154-8	24	1	40	40	44
5	1001 секрет реестра Windows NT	Дэниелс	Русская Редакция	1999	320	5-7502-0073-6	12	35	35,011	55	60,5
6	12 уроков тенниса.Самый доступный самоучитель для всех	Янчук	Терра-Спорт	1999	30	5-93127-026-4	60	6	10	14	15,4
7	13 шагов к успеху или практические советы как быстро достичь карьеры в		Политехника	1998	91	5-9498-05-08	50	39	4	6	6,6
8	1С:Бухгалтерия.Самоучитель.Версии 7.5 и 7.7 в вопросах и ответах	Комягин	Триумф	1999	400	5-89392-023-6	6	23	131,25	190	209
						5-8046-					

Рис. 12.8. Демонстрация элемента DataGrid

После размещения на форме компонентов BdpConnection, BdpDataAdapter, DataSet и их настройки (см. раздел «Простой пример») был создан следующий обработчик события Load:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
begin
  DataGrid1.DataBind;
end;
```

Все оформительские свойства элемента доступны в окне инспектора объектов и позволяют придать данным нужный вид еще на этапе редактирования.

Источником данных может быть XML-файл. В этом случае следует воспользоваться методом DataSet.ReadXML:

```
implementation
uses System.IO;  // В этом пространстве имен определен класс
                 // StreamReader

procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
// Загружает XML-файл в НД и связывает его с сеткой
var
  SR: StreamReader;
  DV: DataView;
  FileName: String;
begin
  DataSet1.Tables.Clear;                          // Очищаем НД
  FileName := Server.MapPath('books.xml');        // Полное имя файла
  SR := StreamReader.Create(FileName);            // Создаем поток
  DataSet1.ReadXML(SR);                           // Читаем данные в НД
  SR.Close;                                       // Закрываем поток
  DV := DataView.Create(DataSet1.Tables[0]); // Создаем
                                                  // представление
  DataGrid1.DataSource := DV;                     // Связываем его
                                                  // с элементом
  DataGrid1.DataBind;
end;
```

Встроенный объект Server (этот объект есть в каждом приложении ASP.NET) имеет метод MapPath, который возвращает полный маршрут доступа к указанному файлу по его имени при условии, что файл располагается в одной из обслуживаемых сервером папок. Если при обращении к методу указана пустая строка, он возвращает полный маршрут доступа к приложению ASP.NET.

Сетка DataGrid может сортировать публикуемые данные по отдельным столбцам. Для этого в ее свойство AllowSorting нужно поместить значение True. В результате заголовки столбцов будут оформлены как гиперссылки, а щелчок на заголовке вызовет событие SortCommand, обработчику которого передается имя столбца. На рис. 12.9 показано окно программы примера (файл Ch12\DataGridSort\WebForm1.aspx), в котором представлены книги из таблицы BOOKS, причем сетка позволяет сортировать книги по любому столбцу.

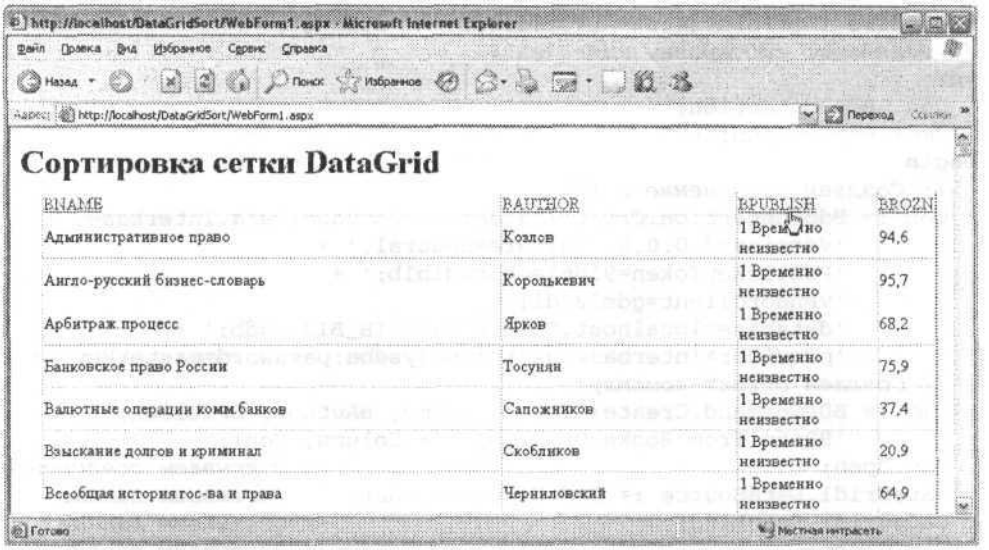


Рис. 12.9. Пример сортировки по столбцу BPublish

Для воспроизведения примера поместите на пустую форму элемент DataGrid и напишите такие строки кода:

```
type
  TWebForm1 = class(System.Web.UI.Page)
  ...
  private
    procedure SortDataGrid(Column: String);
    { Private Declarations }
  public
    { Public Declarations }
  end;

implementation

procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
// Сортирует книги по названию в момент загрузки страницы
begin
  if not IsPostBack then
    SortDataGrid('BName');
end;

procedure TWebForm1.DataGrid1_SortCommand(source: System.Object;
    e: System.Web.UI.WebControls.DataGridSortCommandEventArgs);
// Сортирует книги по выбранному столбцу
begin
  SortDataGrid(e.SortExpression);
end;
```

```
procedure TWebForm1.SortDataGrid(Column: String);
// Реализует сортировку книг
var
  Con: BdpConnection;
  Cmd: BdpCommand;
begin
  // Создаем соединение с БД:
  Con := BdpConnection.Create('assembly=Borland.Data.Interbase, ' +
         'Version=2.0.0.0, Culture=neutral,' +
         'PublicKeyToken=91d62ebb5b0d1b1b;' +
         'vendorclient=gds32.dll;' +
         'database=localhost:C:\BiblData\IB_BIBL.gdb;' +
         'provider=Interbase;username=sysdba;password=masterkey');
  // Создаем объект-команду:
  Cmd := BDPCommand.Create('Select BName, BAuthor, BPublish, ' +
         'BRozn From Books Order By ' + Column, Con);
  Con.Open;                                    // Открываем соединение
  DataGrid1.DataSource := Cmd.ExecuteReader; // Считываем данные
  DataGrid1.DataBind;                          // Связываем данные
  Con.Close;                                   // Закрываем соединение

end;
```

Сетка может содержать столбцы разных типов, в том числе:

- `BoundColumn` — обычный столбец для отображения данных;
- `HyperLinkColumn` — вывод записей в виде гиперссылок;
- `TemplateColumn` — вывод записей с форматированием по шаблону;
- `ButtonColumn` — вывод записей в виде кнопок;
- `EditCommandColumn` — вывод команд редактирования `Edit`, `Update` и `Cancel`.

По умолчанию элемент `DataGrid` выводит все столбцы набора данных в формате `BoundColumn`. Но если в свойство `AutoGenerateColumns` поместить значение `False`, столбцы можно создавать вручную, что дает разработчику более полный контроль над видом элемента.

Отключать автоматическое создание столбцов и добавлять вручную столбцы `BoundColumn` можно в том случае, когда не все поля НД нуждаются в опубликовании. Если, например, использовать стандартный прием перетаскивания на веб-страницу таблицы `BOOKS`, в НД появятся все поля этой таблицы. Однако для прайс-листа многие из них не нужны. В этом случае в свойство `AutoGenerateColumns` следует поместить значение `False` и щелчком на кнопке в свойстве `Columns` вызвать окно редактора столбцов (рис. 12.10).

С помощью редактора можно выбрать тип столбца и указать его заголовок. На рис. 12.10 показан процесс формирования первого столбца прайс-листа. В строке Выражение форматир. данных можно ввести регулярное выражение, определяющее формат выводимых данных. В частности, символы `{0:c}` предписывают выводить вещественные числа в денежном формате. Эти символы используются для форматирования данных в поле `BRozn`. Окно с прайс-листом показано на рис. 12.11 (файл Ch12\DataGridColumn\WebForm1.aspx).

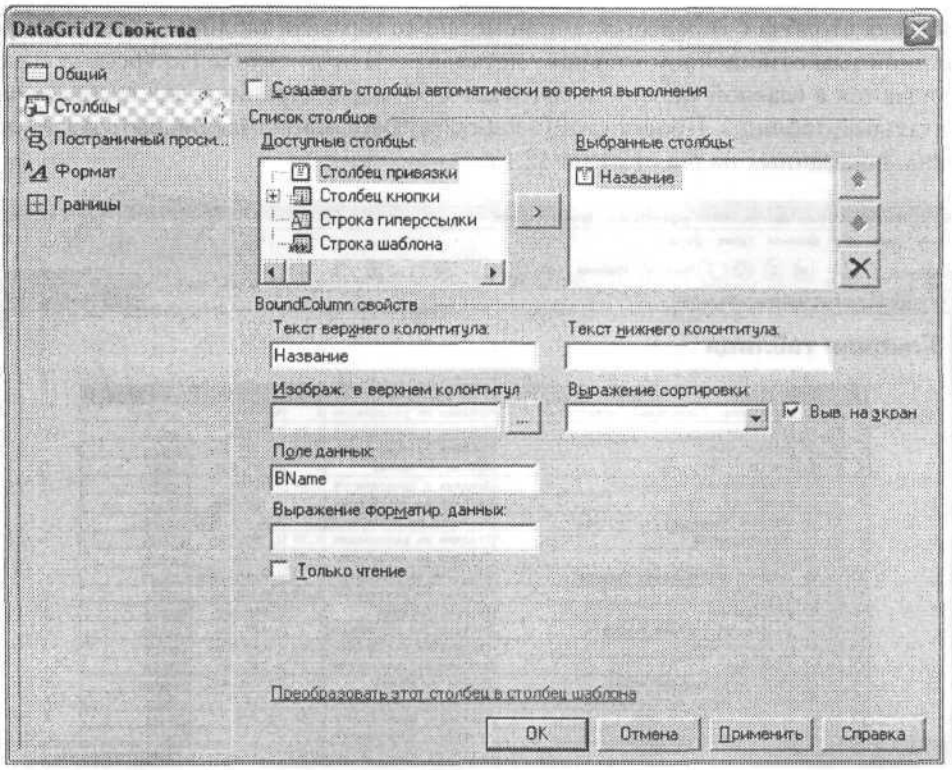
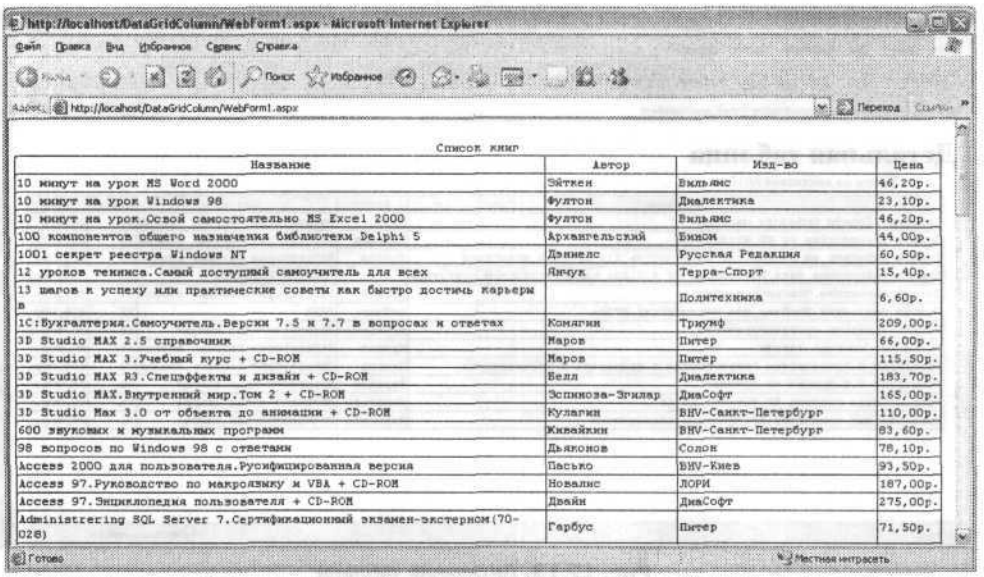


Рис. 12.10. Окно редактора столбцов



Название	Автор	Изд-во	Цена
10 минут на урок MS Word 2000	Эйткен	Вильямс	46,20р.
10 минут на урок Windows 98	Фултон	Диалектика	23,10р.
10 минут на урок.Освой самостоятельно MS Excel 2000	Фултон	Вильямс	46,20р.
100 компонентов общего назначения библиотеки Delphi 5	Архангельский	Бином	44,00р.
1001 секрет реестра Windows NT	Даниелс	Русская Редакция	60,50р.
12 уроков техникса.Самый доступный самоучитель для всех	Янчук	Терра-Спорт	15,40р.
13 шагов к успеху или практические советы как быстро достичь карьерн в		Политехника	6,60р.
1С:Бухгалтерия.Самоучитель.Версии 7.5 и 7.7 в вопросах и ответах	Комягин	Триумф	209,00р.
3D Studio MAX 2.5 справочник	Маров	Питер	66,00р.
3D Studio MAX 3.Учебный курс + CD-ROM	Маров	Питер	115,50р.
3D Studio MAX R3.Спецэффекты и дизайн + CD-ROM	Белл	Диалектика	183,70р.
3D Studio MAX.Внутренний мир.Том 2 + CD-ROM	Эспиноза-Зиглар	ДиаСофт	165,00р.
3D Studio Max 3.0 от объекта до анимации + CD-ROM	Кулагин	BHV-Санкт-Петербург	110,00р.
600 звуковых и музыкальных программ	Живайкин	BHV-Санкт-Петербург	83,60р.
98 вопросов по Windows 98 с ответами	Дьяконов	Солон	78,10р.
Access 2000 для пользователя.Русифицированная версия	Пасько	BHV-Киев	93,50р.
Access 97.Руководство по макроязыку и VBA + CD-ROM	Новалис	ЛОРИ	187,00р.
Access 97.Энциклопедия пользователя + CD-ROM	Двайн	ДиаСофт	275,00р.
Administrering SQL Server 7.Сертификационный экзамен-экстерном(70-028)	Гарбус	Питер	71,50р.

Рис. 12.11. Прайс-лист

Вставка столбца с гиперссылками помогает отобразить таблицы, связанные реляционным отношением главная—детальная. В этом случае гиперссылки размещаются в главной таблице и содержат URL-адрес страницы для отображения детальной таблицы. Проект Ch12\DataGridMastDet\DataGridMastDet.dpr создает два окна, показанных на рис. 12.12 и 12.13.

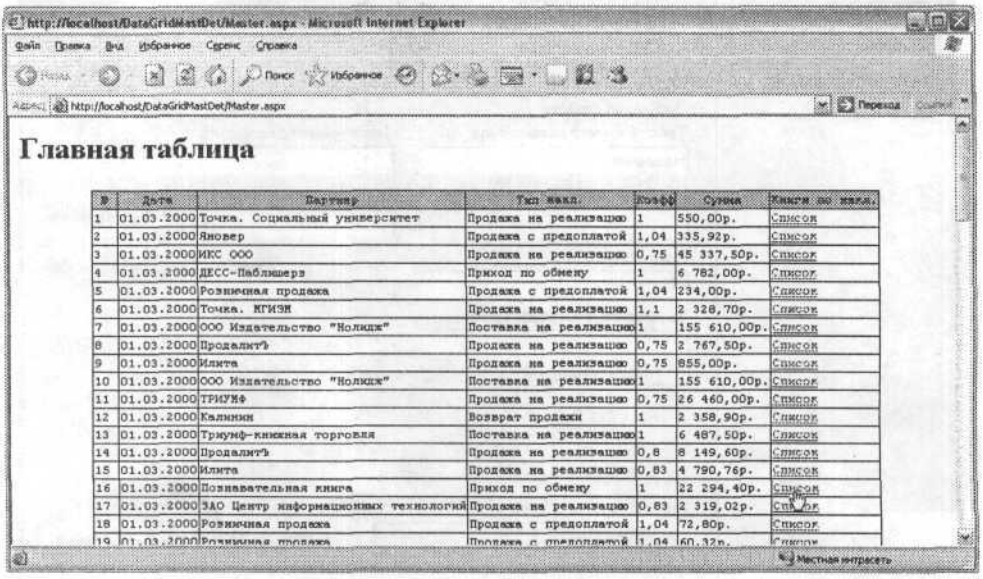


Рис. 12.12. Главная таблица

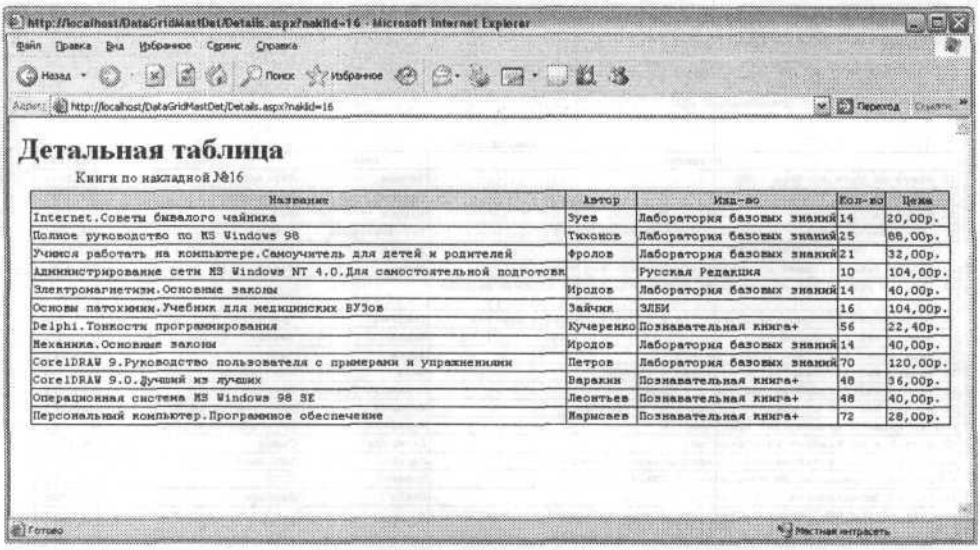


Рис. 12.13. Детальная таблица

На рис. 12.12 показана таблица NAKLS. Правая колонка сетки содержит гиперссылки такого рода (рис. 12.14):

```
Details.aspx?naklid={0}
```

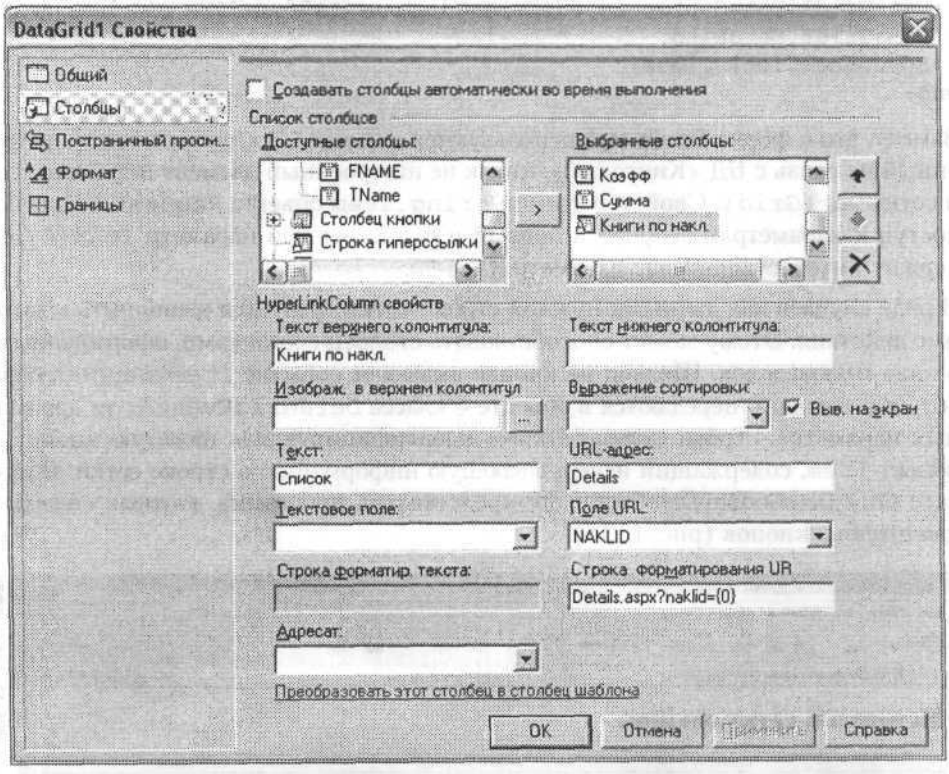


Рис. 12.14. Назначение гиперссылки

ПРИМЕЧАНИЕ

Столбцы типов, отличных от BoundColumne, доступны в окне редактора столбцов ниже узла (возможно, пустого) Столбец привязки.

В этой гиперссылке перед знаком вопроса указана веб-страница, предназначенная для отображения детальных данных, а после знака вопроса — передаваемые ей данные. Эти данные задаются именем (naklid) и значением {0}. Такая строка форматирования означает, что в качестве значения будет передаваться значение поля, выбранного в списке Поле URL. Обработчик Page_Load страницы Detailse.aspx имеет такой вид:

```
procedure TWebForm2.Page_Load(sender: System.Object;
                              e: System.EventArgs);
begin
  Label1.Text := 'Книги по накладной №' +
                                 Request.QueryString.Item[0];
```

```
  BdpCommand1.CommandText := 'SELECT BNAME, BAUTHOR, BPublish, ' +
    'MQUAN, MPRICE FROM MOVES, BOOKS WHERE MBOOK=BOOKID ' +
    'and NAKLID=' + Request.QueryString.Item[0];
  BdpConnection1.Open;
  DataGrid1.DataSource := BdpCommand1.ExecuteReader;
  DataGrid1.DataBind;
  BdpConnection1.Close;
end;
```

Замечу, что в форме Details.aspx используются элемент BdpConnection1, настроенный на связь с БД «Книголюб», никак не настроенный элемент BdpCommand1 и сетка DataGrid1. Свойство QueryString.Item объекта Request открывает доступ к параметрам запроса, а первый и единственный параметр Item[0] содержит интересующий нас параметр NaklID.

В ряде случаев над данными нужной строки сетки требуется выполнить некоторые действия. Этому может способствовать столбец с кнопками, оформленными в виде гиперссылок. Щелчок на кнопке вызывает событие ItemChanged, обработчику которого передаются в объекте e класса DataGridEventArgs два важных параметра: строка CommandName, идентифицирующая нажатую кнопку, и объект Item, содержащий исчерпывающую информацию о строке сетки. В проекте Ch12\DataGridBtn\DataGrigBtn.dpr представлена программа, которая содержит два столбца кнопок (рис. 12.15).

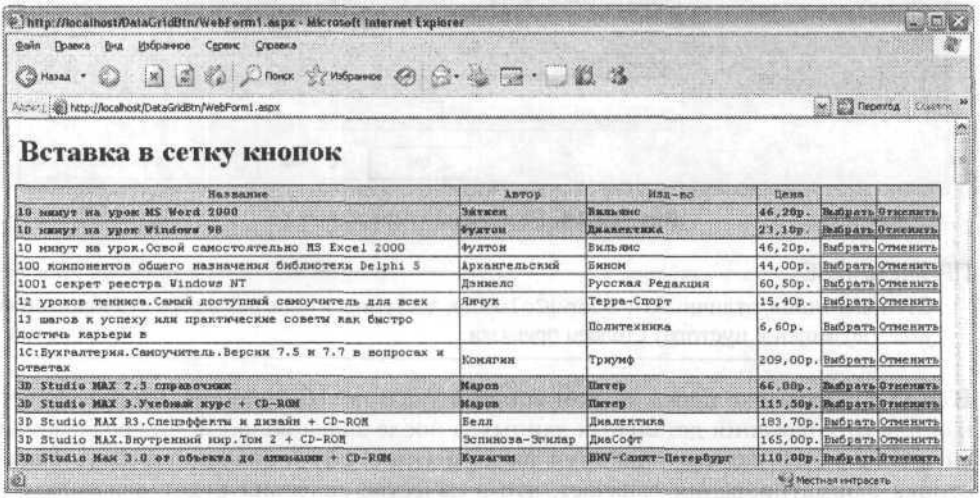


Рис. 12.15. Сетка с кнопками

Щелчок на кнопке Выбрать выделяет ряд фоновым цветом и выводит текст в нем полужирным шрифтом. Щелчок на кнопке Отменить отменяет выделение. Все действия реализует такой обработчик:

```
procedure TWebForm1.DataGrid1_ItemCommand(source: System.Object;
          e: System.Web.UI.WebControls.DataGridCommandEventArgs);
```

```
begin
  if e.CommandName = 'Выбрать' then
  begin
    e.Item.BackColor := System.Drawing.Color.LightGreen;
    e.Item.Font.Bold := True
  end else begin
    e.Item.BackColor := System.Drawing.Color.White;
    e.Item.Font.Bold := False
  end
end;
```

Свойство `CommandName` определяется содержимым поля **Имя команды** в окне редактора колонок элемента `DataGrid` (рис. 12.16).

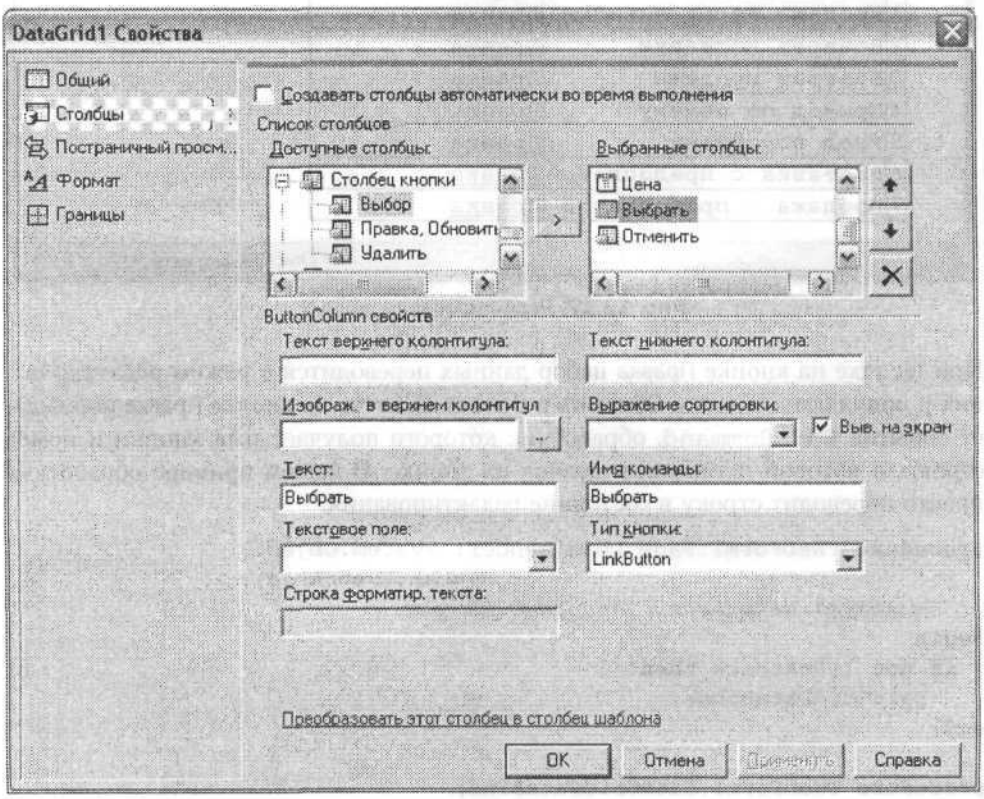


Рис. 12.16. Определение имени связанной с кнопкой команды

Наконец, элемент `DataGrid` не только может отображать данные, но и позволяет их редактировать. Для этого в состав его столбцов следует включить столбец с кнопками (гиперссылками) **Правка**, **Обновить** и **Отмена**. Следует отметить, что эти кнопки (проект Ch12\DataGridEdit\DataGridEdit.dpr) предоставляют лишь необходимые интерфейсные средства, но не осуществляют изменения в исходном наборе данных (рис. 12.17).

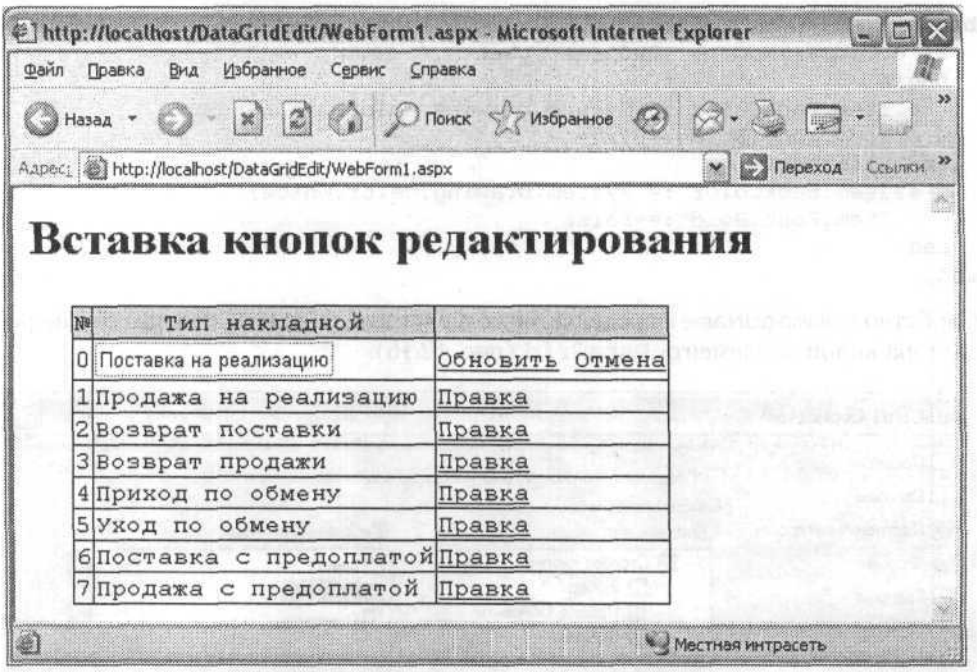


Рис. 12.17. Редактирование данных

При щелчке на кнопке Правка набор данных переводится в режим редактирования и появляются кнопки Обновить и Отмена. Щелчок на кнопке Правка порождает событие EditCommand, обработчик которого получает имя кнопки и номер строки, в которой произошел щелчок на кнопке. В нашем примере обработчик просто переводит строку в состояние редактирования:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
// Реализует начальное связывание данных
begin
  if not IsPostBack then
    DataGridDataBind;
end;

procedure TWebForm1.DataGridDataBind;
// Получает и связывает данные
var
  Cmd: BdpCommand;
begin
  // Создаем команду (элемент BdpConnection1 присоединен к проекту
  // на этапе конструирования формы и связан с БД "Книголюб"):
  Cmd := BdpCommand.Create(
           'SELECT * FROM TYPES', BdpConnection1, NIL);
```

```
  BdpConnection1.Open;                          // Открываем соединение
  DataGrid1.DataSource := Cmd.ExecuteReader; // Читаем данные
  DataGrid1.DataBind;                           // Связываем данные
  BdpConnection1.Close;                         // Закрываем соединение
end;

procedure TWebForm1.DataGrid1_EditCommand(source: System.Object;
        e: System.Web.UI.WebControls.DataGridCommandEventArgs);
// Щелчок на кнопке "Правка"
begin
  DataGrid1.EditItemIndex := e.Item.ItemIndex; // Переводим строку
  DataGridDataBind;                             // в режим
                                                // редактирования
end;
```

Для действительного изменения данных или для отказа от изменений предусмотрены связанные с элементом события `CancelCommand` и `UpdateCommand`, которые возникают при щелчках на кнопках **Отмена** и **Обновить** соответственно:

```
procedure TWebForm1.DataGrid1_UpdateCommand(source: System.Object;
          e: System.Web.UI.WebControls.DataGridCommandEventArgs);
// Подтверждает изменения (кнопка "Обновить"). Ключевое поле TypeID
// не может изменяться (оно только для чтения)
var
   TB: TextBox;
   Cmd: BdpCommand;
   ID, Name: String;
begin
   ID := e.Item.ItemIndex.ToString;       // Значение ключевого поля
   TB := e.Item.Cells[1].Controls[0] as TextBox; // Элемент
                                                 // редактирования
   Name := TB.Text;                  // Новый текст поля TName
  // Создаем команду изменения значения:
   Cmd := BdpCommand.Create('UPDATE TYPES SET TNAME=''' + Name +
                ''' WHERE TYPEID=' + ID, BdpConnection1, NIL);
   BdpConnection1.Open;              // Открываем соединение
   Cmd.ExecuteNonQuery;              // Выполняем изменение
   BdpConnection1.Close;             // Закрываем соединение
   DataGrid1.EditItemIndex := -1; // Прекращаем редактирование
                                     // строки
   DataGridDataBind;
end;

procedure TWebForm1.DataGrid1_CancelCommand(source: System.Object;
          e: System.Web.UI.WebControls.DataGridCommandEventArgs);
// Щелчок на кнопке "Отмена"
begin
  DataGrid1.EditItemIndex := -1;  // Отменяем редактирование строки
  DataGridDataBind;
end;
```

Элементы категории DB Web

Разработчики Delphi дополнили стандартные компоненты для работы с базами данных новыми, которые существенно упрощают доступ к данным и не требуют от программиста дополнительных усилий на ручное кодирование aspx-файлов. По сравнению со стандартными элементами они имеют следующие преимущества:

- не требуют связывания данных методом `DataBind`;
- представляют данные на этапе редактирования формы;
- автоматически подтверждают изменения в данных;
- дают средства навигации по набору данных.

Элемент DBWebDataSource

Элемент `DBWebDataSource` играет ту же роль, что и компонент `TDataSource` в VCL-приложениях. Он является клапаном, через который остальные компоненты категории DB Web получают доступ к данным. В дополнение к этому своему основному назначению элемент осуществляет привязку данных методом `DataBinding` к обслуживаемым им элементам отображения данных (`DBWebGrid`, `DBWebLabel`, `DBWebImage` и т. п.).

В табл. 12.1 представлены некоторые свойства элемента.

Таблица 12.1. Свойства элемента DBWebDataSource

Свойство	Описание
`property AutoRefresh: Boolean;`	Если содержит `False` (по умолчанию), данные обновляются только по требованию клиента, в противном случае — когда загружается страница
`property AutoUpdateCashe: Boolean;`	Если содержит `False` (по умолчанию), измененные данные записываются в XML-файл только в обработчике события `OnApplyChanges`, в противном случае — при каждом изменении (при этом свойство `UseUniquiFileName` игнорируется)
`property CascadingDeletes: CscadingDeleteOptions;`	Определяет изменение таблиц главная—детальная (см. далее)
`property CascadingUpdates: CascadingUpdateOptions;`	Определяет возможность изменения внешнего ключа в таблицах главная—детальная (см. далее)
`property DataSource: Object;`	Содержит ссылку на набор данных
`property ErrorOption: ErrorHtmlOptions;`	Определяет обработку ошибок (см. далее)
`property UseUniqueFileName: Boolean;`	Если содержит `True`, при записи XML-файла будет создаваться уникальное имя

Свойство	Описание
`property XMLFileName: String;`	Имя читаемого или записываемого файла XML
`property XCQShemaFile; String;`	Имя файла, содержащего схему XML-файла

Свойство `CascadingDeletes` определяет реакцию на удаление записи в таблицах главная–детальная и может принимать одно из следующих значений:

- `NoMasterDelete` (по умолчанию) – запрещает удаление записи главной таблицы. Это значение устанавливается, когда сервер накладывает на главную таблицу ограничение внешнего ключа. В этом случае пользователь должен самостоятельно удалить ненужные записи детальной таблицы и подтвердить удаление с помощью, например, метода `BdpDetailAdapter.AutoUpdate`, затем удалить запись в главной таблице и подтвердить изменение с помощью метода `BdpMasterAdapter.AutoUpdate`;
- `ServerCascadeDelete` – устанавливается, когда сервер настроен на поддержку каскадных удалений. В этом случае до подтверждения изменений пользователь может отменить удаление, и при этом восстанавливаются главная запись и все связанные с ней детальные записи. Если сервер не поддерживает каскадные изменения, детальные записи сохранятся после уничтожения главной записи;
- `ServerNoForeignKey` – устанавливается, когда сервер не накладывает на главную таблицу ограничение внешнего ключа. В этом случае уничтожение главной записи вызывает автоматическое уничтожение всех связанных с ней детальных записей. Если до подтверждения изменений пользователь отменит удаление, главная и детальная таблицы восстанавливают свой первоначальный вид. Если на главную таблицу наложено ограничение внешнего ключа, это значение при уничтожении главной записи вызовет ошибку.

Свойство `CascadingUpdates` определяет возможность каскадных изменений внешнего ключа в таблицах главная–детальная и может принимать одно из следующих значений:

- `NoMasterUpdate` (по умолчанию) – запрещает изменение ключевого поля главной таблицы;
- `ServerCascadeUpdate` – устанавливается, когда сервер поддерживает каскадное изменение ключевого поля в записях детальной таблицы;
- `ServerNoForeignKey` – разрешает изменение ключевого поля главной записи и автоматически корректирует ключевые поля в записях детальной таблицы. Это значение можно устанавливать только в том случае, когда сервер не накладывает на главную таблицу ограничения внешнего ключа.

Свойство `ErrorOption` определяет реакцию на возникшую ошибку и может содержать одно из следующих значений:

- `logOnSamePage` (по умолчанию) – сообщение об ошибке появляется на вебстранице;
- `logOnErrorPage` – сообщение появляется на отдельной странице с кнопкой OK;
- `logWithErrorEvent` – сообщение не появляется, так как установлен обработчик события `OnError`.

Элемент DBWebDataSource может оперировать XML-файлами. При этом, если свойство XMLFileName содержит имя существующего файла, этот файл становится базовым источником данных для элемента. Если свойство содержит имя несуществующего файла, в момент загрузки страницы файл будет создан и наполнен данными из набора данных DataSet.

Элемент не имеет специфических методов. Его события перечислены в табл. 12.2.

Таблица 12.2. События элемента DBWebDataSource

Событие	Описание
`property OnAfterSetChanges: WebControlsEvent;`	Возникает после подтверждения изменений
`property OnApplyChangesRequest: WenControlsEvent;`	Возникает, когда пользователь требует подтверждения сделанных им изменений
`property OnAutoApplyRequest: WebControlsEvent;`	Возникает, когда пользователь требует автоматического подтверждения делаемых им изменений
`property OnError: OnErrorEvent;`	Возникает при обнаружении ошибки в процессе обработки запроса. Обработчику передается список всех сообщений об ошибках и список всех предупреждающих сообщений
`property OnGetPostCollecton: PostCollecionEvent;`	Возникает перед получением изменений на странице клиента (см. далее)
`property OnRefreshRequest: WebControlsEvebt;`	Возникает при необходимости обновить данные
`property OnScroll: OnScrollEvent;`	Некоторые компоненты (DBWebGrid, DBWebTextBox и DBWebListBox) поддерживают прокрутку данных. Объект класса OnScrollEvent содержит имя таблицы, текущую запись и запись, которая была текущей до прокрутки

Важную роль играет событие OnApplyChangesRequest, так как именно в обработчике этого события можно вызвать метод AutoUpdate адаптера BdpDataAdapter и, таким образом, подтвердить сделанные изменения. Метод имеет такую сигнатуру:

```
procedure AutoUpdate(DS: DataSet; TableName: String;
                                  DataMode: BdpDataMode);
```

Здесь DS — набор данных; TableName — имя обновляемой таблицы; DataMode — параметр, определяющий обновляемые записи (All — все записи; Changed — измененные записи; Key — записи с заданными значениями ключевого поля). Например:

```
BdpAdapter1.AutoUpdate(DataSet1, 'TYPES', BdpDataMode.All);
```

ПРИМЕЧАНИЕ

Только после определения обработчика `OnApplyChangesRequist` становится доступной кнопка Apply в навигаторе `DBWebNavigator`.

Обработчик события `OnGetPostCollection` получает три параметра: `sender` — объект, возбудивший событие, `e` — объект класса `PostCollectionEventArgs` с параметрами, `StopUpdate` — логическая переменная. Класс `PostCollectionEventArgs` порожден от `EventArgs` и обогащает своего родителя членом `PostCollection` класса `NameValueCollection`.

Элемент DBWebNavigator

Элемент `DBWebNavigator` используется совместно с другими элементами, такими, например, как сетка, и обеспечивает удобные средства навигации по набору данных, а также операции обновления, подтверждения изменений (см. ранее), отмены изменений, вставки и удаления записи. Он от клапана `DBDataSource`, ссылку на который хранит его одноименное свойство, получает данные из НД, указанного в свойстве `TableName`.

Элемент имеет 10 (по количеству кнопок) пар свойств *XXXX*`ButtonIcon` типа **`String`** и *XXXX*`ButtonVisible` типа `Boolean`. Первое определяет изображение на поверхности кнопки, а второе — наличие соответствующей кнопки в элементе. Свойство `ButtonType` указывает тип изображений на кнопках. Оно может иметь одно из следующих значений:

- `ButtonIcons` — на кнопках демонстрируются изображения, заданные в свойствах *XXXX*`ButtonIcon`; для согласования размеров изображения и размеров кнопки следует использовать графические файлы в формате GIF;
- `ButtonSymbols` (по умолчанию) — демонстрируются такие комбинации символов: |< (перейти к первой записи), << (к предыдущей записи), >> (к следующей записи), >| (к последней записи), + (вставить запись), _ (удалить текущую запись), [>] (подтвердить изменения), [<] (обновить данные), x (отменить последнее изменение), xx (отменить все изменения);
- `ButtonText` — показываются надписи, поясняющие связанные с кнопкой действия (First, Pervious, Next, Last, Insert, Delete, Apply, Refresh, Undo, UndoAll).

На рис. 12.18 элемент показан в режиме `ButtonSymbols` (верхний ряд кнопок) и `ButtonText` (нижний ряд кнопок).

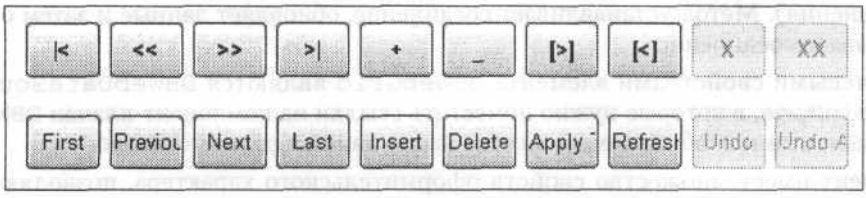


Рис. 12.18. Иллюстрация свойства ButtonType

Элемент DBWebGrid

Элемент DBWebGrid предназначен для отображения данных в сетке (рис. 12.19).

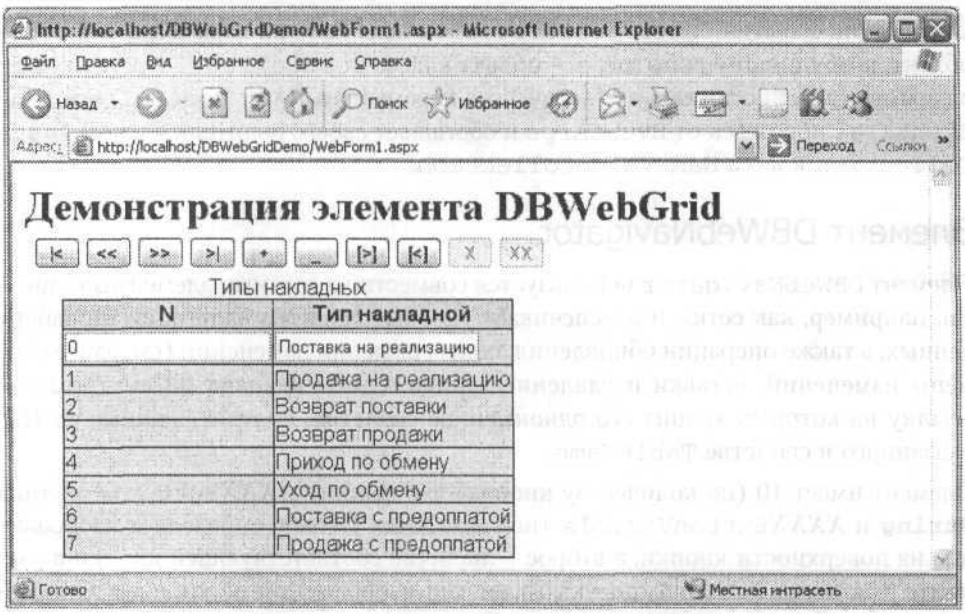


Рис. 12.19. Демонстрация элемента DBWebGrid

ПРИМЕЧАНИЕ

Для подтверждения сделанных изменений необходимо определить обработчик события OnApplyChangesRequist клапана DBWebDataSource (см. файл Ch10\DBWebGridDemo\WebForm1.pas).

Вот пример обработчика:

```
procedure TWebForm1.DBWebDataSource1_OnApplyChangesRequest(
   sender: System.Object; e: Borland.Data.Web.WebControlEventArgs);
begin
  BdpDataAdapter1.AutoUpdate(dataSet1, 'TYPES', BdpUpdateMode.All);
end;
```

Обратите внимание: к моменту вызова метода AutoUpdate соединение с источником данных разорвано (это легко проверить, проанализировав свойство State соединения). Метод устанавливает соединение, обновляет данные и затем опять разрывает соединение.

Ключевыми свойствами элемента DBWebGrid являются DBWebDataSource и TableName, в которые нужно поместить ссылки на компонент-клапан DBWebDataSource и отображаемый в сетке набор данных соответственно.

Элемент имеет множество свойств оформительского характера, позволяющих разнообразить вид отображаемых данных.

Большинство свойств элемента унаследовано от родительского класса DataGrid. Собственные свойства DBWebGrid представлены в табл. 12.3.

Таблица 12.3. Свойства элемента DBWebGrid

Свойство	Описание
`property AccessKey: String;`	Определяет символ быстрого доступа к элементу (в сочетании с клавишей Alt)
`property AllowPagging: Boolean;`	Управляет разбиением данных на страницы (см. далее)
`property DBDataSourse: IDBDataSourse;`	Содержит ссылку на элемент-клапан класса DBWebDataSourse
`property HandleColumnEvents: Boolean;`	Если содержит True (по умолчанию), события OnEditCommand и OnCancelCommand обрабатываются автоматически
`property HandlePaggingEvents: Boolean;`	Если содержит True (по умолчанию), событие OnPageIndexChange обрабатывается автоматически
`property ReadOnly: Boolean;`	Если содержит True, элемент не может редактировать данные
`property TableName: String;`	Содержит имя отображаемой таблицы DataTable

Если свойство AllowPagging содержит значение True, записи автоматически разбиваются на страницы, когда общее количество записей в наборе данных больше значения, указанного в свойстве PageSize. В противном случае свойство PageSize игнорируется и выводятся все записи НД или то их количество, которое определяется свойством MaxRecords адаптера. Если установлено разбиение на страницы, внизу сетки появляются значки, щелчки на которых меняют отображаемую в сетке страницу данных.

Элемент имеет метод UpdatePageIndex, позволяющий управлять курсором таблицы.

Визуализирующие элементы

В категории DB Web собраны визуализирующие элементы, которые в функциональном плане повторяют обычные элементы категории Web Controls, но разработаны специально для визуализации баз данных. Их главная особенность заключается в том, что они обеспечивают привязку данных автоматически (эта привязка реализуется клапаном DBWebDataSource, с помощью которого элементы DBWeb*XXXX* получают доступ к данным). Вторая особенность — предъявление данных не только на этапе прогона, но и на этапе конструирования формы.

Все визуализирующие элементы имеют свойства DBDataSource, TableName и ColumnName, в которые помещаются, соответственно, ссылка на клапан класса DBWebDataSource, имя таблицы и имя поля таблицы, содержимое которого и визуализирует тот или иной элемент.

Интересной особенностью списочных элементов (DBWebListBox, DBWebDropDownList, DBWebRadioButtonList и т. п.) является их способность демонстрировать данные из так называемых *подстановочных* (lookup) полей. Подстановочными называются поля, содержащие ссылки на поля других таблиц. Например, в таблице MOVES поле MBOOK содержит номер записи из таблицы BOOKS, в которой определены все параметры книги (название, автор, издательство и т. д.). Для работы с подстановочными полями списочные элементы имеют свойства LookupTable, DataTextField и DataValueField: в первом указывается подстановочная таблица, во втором — поле, содержимое которого будет отображаться в элементе, третье определяет ключевое поле для связи с подстановочным.

ПРИМЕЧАНИЕ

Правильное отображение подстановочных полей гарантируется после установки так называемых пакетов обновления Delphi 2005 Architect (www.borland.com/downloads/registered/download_delphi.html) объемом около 50 Мбайт.

В следующем примере (Ch12\DBWebControls\WebForm1.aspx) демонстрируется применение различных визуализирующих элементов для отображения данных из таблицы NAKLS (рис. 12.20).

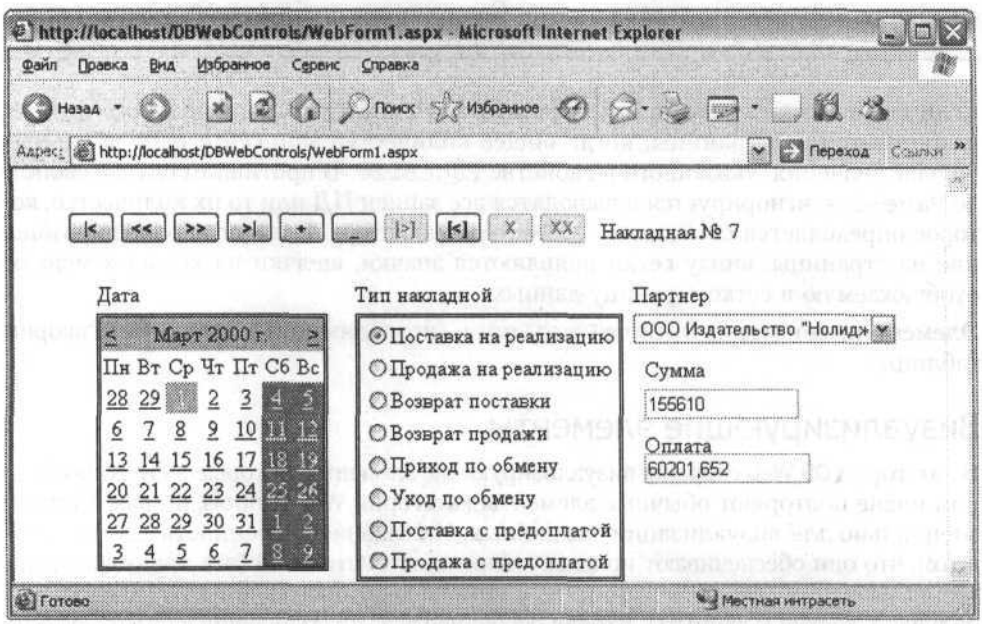


Рис. 12.20. Пример использования визуализирующих элементов

В примере используются невидимые на рисунке компоненты класса BdpDataAdapter для получения данных из главной (NAKLS) и двух подстановочных (TYPES и FIRMS) таблиц. Помимо трех адаптеров используются также невидимые

компоненты классов BdpConnection, DataSet и BdpDataSource. Свойства визуализирующих компонентов настроены следующим образом:

- компонент DBWebNavigator1:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
- компонент DBWebLabel1:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
  - ColumnName = NAKLID;
- компонент DBWebCalendar1:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
  - ColumnName = NDATE;
- компонент DBWebRadioButtonList1:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
  - ColumnName = NTYPE;
  - LookupTable = TYPES;
  - DataTextField = TNAME;
  - DataValeuField = TYPEID;
- компонент DBWebDropDownlist1:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
  - ColumnName = NFIRM;
  - LookupTable = FIRMS;
  - DataTextField = FNAME;
  - DataValeuField = FIRMID;
- компонент DBWebTextBox1:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
  - ColumnName = NSUM;
- компонент DBWebTextBox2:
  - DBDataSource = DBWebDataSource1;
  - TableName = NAKLS;
  - ColumnName = NPAEDSUM.

13 Веб-службы

Веб-службы представляют собой относительно новое явление в Интернете. Всемирная сеть в ее сегодняшнем виде основана на том, что пользователи посещают различные веб-страницы с помощью своих веб-браузеров. До недавнего времени единственным способом предоставления информации в Интернете являлось ее размещение на веб-странице, которую пользователи могли просматривать. Однако часто возникает необходимость получить по Сети доступ к удаленному программному обеспечению. Такой доступ и реализуется веб-службами. Как объявила корпорация Microsoft, с переходом на технологию .NET она решила сделать акцент не на разработке программного обеспечения, а на предоставлении программных услуг. При этом новый программный продукт предлагается на рынке в виде веб-службы. Таким образом, приложение сдается в аренду, а от пользователя для его применения и обновления требуется постоянное подключение к Интернету.

Эта глава посвящена применению Delphi 2005 и платформы .NET для создания и использования веб-служб.

Создание веб-служб

Процедуру создания простой веб-службы продемонстрируем на основе заготовок, вставленных в файл службы в виде комментариев. Эта служба возвращает строку «Hello World!».

1. Создайте новую веб-службу. Для этого выберите команду File ▸ New ▸ Others и в узле Delphi for .NET Projects выберите значок ASP.NET Web Service Application (рис. 13.1). Назовите проект WSHelloWorld.
2. На странице WebService1.pas раскомментируйте следующее объявление в разделе **interface**:

```
//    (*
      // Sample Web Service Method
      [WebMethod]
      function HelloWorld: string;
//    *)
    end;
```

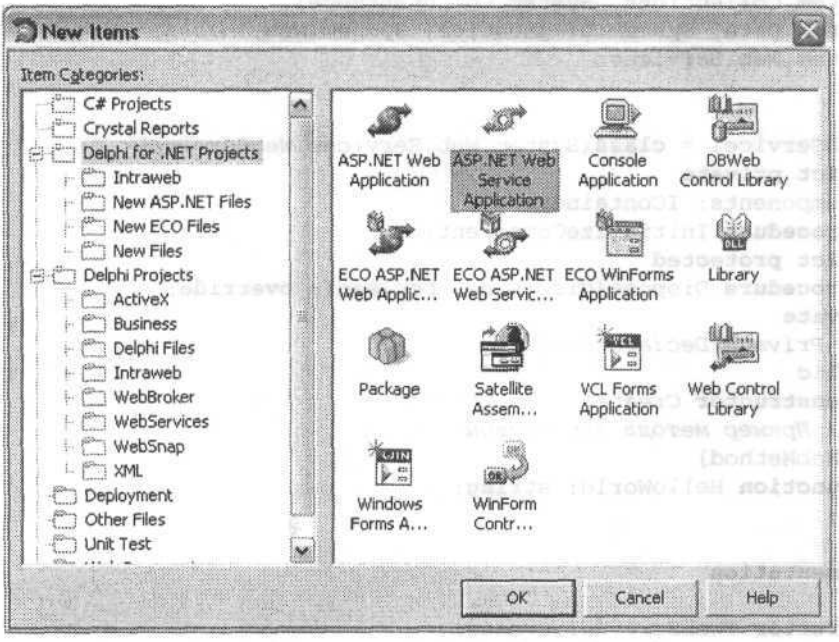


Рис. 13.1. Выбор значка ASP.NET Web Service Application

3. На той же странице раскомментируйте такое объявление в разделе **implementation**:

```
// Sample Web Service Method
// The following method is provided
// to allow for testing a new web service.
//(*
function TWebService1.HelloWorld: string;
begin
  Result := 'Hello World';
end;
//*)
```

4. Откомпилируйте проект.

Наша простая веб-служба создана! К проекту добавлены файлы WebService1.asmx, WebService1.pas, bin\WSHelloWorld.dll. Последний, собственно, и содержит веб-службу в виде динамически загружаемой библиотеки.

Содержимое файла WebService1.asmx показано в листинге 13.1.

Листинг 13.1. Файл WebService1.asmx

```
unit WebService1;

interface

uses
  System.Collections, System.ComponentModel,
  System.Data, System.Diagnostics, System.Web,
  System.Web.Services;

type
  TWebService1 = class(System.Web.Services.WebService)
  strict private
    components: IContainer;
    procedure InitializeComponent;
  strict protected
    procedure Dispose(disposing: boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
    // Пример метода веб-службы
    [WebMethod]
    function HelloWorld: string;
  end;

implementation

constructor TWebService1.Create;
begin
  inherited;
  InitializeComponent;
end;

procedure TWebService1.Dispose(disposing: boolean);
begin
  if disposing and (components <> nil) then
    components.Dispose;
  inherited Dispose(disposing);
end;

// Пример реализации веб-метода
function TWebService1.HelloWorld: string;
begin
  Result := 'Hello World';
end;
end.
```

Класс `TWebService1` происходит от `System.Web.Services.WebService`. Такое наследование обеспечивает доступ к таким популярным объектам, как `Application`,

Context, Session и т. д. Главное, что отличает этот класс от других, — наличие атрибута [WebMethod] перед объявлением метода HelloWorld в разделе **implementation**. Именно этот атрибут обеспечивает доступ к службе извне.

При запросе URL-адреса веб-службы без параметров ASP.NET возвращает описание службы, как показано на рис. 13.2.

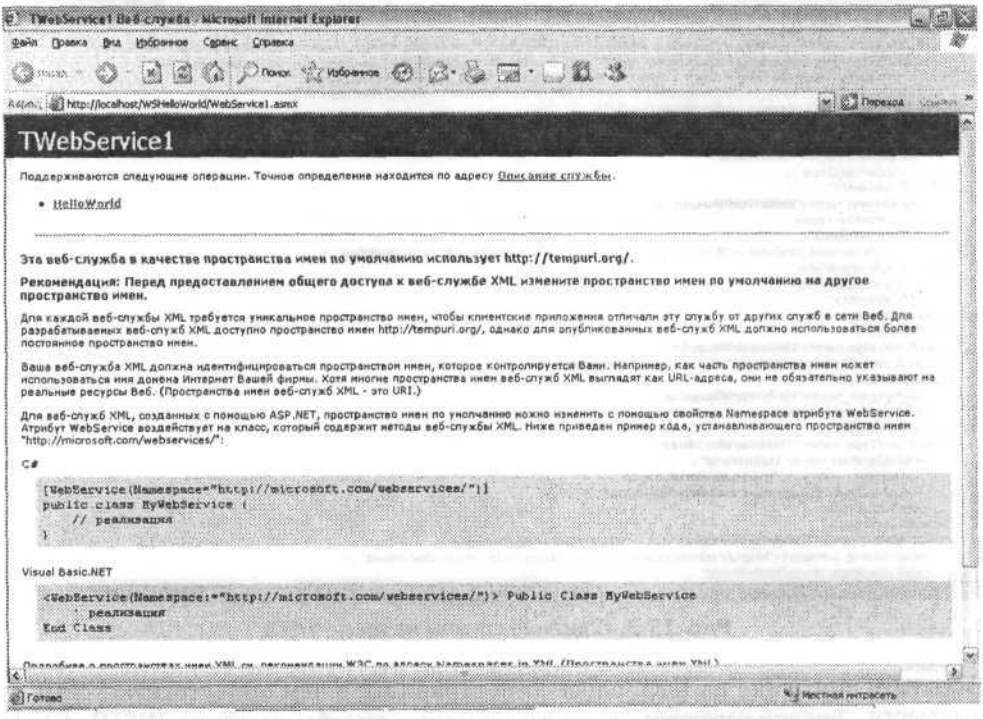


Рис. 13.2. Результат обращения к веб-службе WSHelloWorld

Как видим, страница содержит название класса TWebService1 и перечень объявленных в нем методов веб-службы (в нашем примере — единственный метод HelloWorld). На остальной части страницы выводится информация о необходимости замены умалчиваемого пространства имен http://tempuri.org уникальным, отличающим ее от других одноименных служб в Сети (если, разумеется, служба будет опубликована в Интернете). В качестве пространства имен рекомендуется использовать домен вашей фирмы (или любое другое имя, обеспечивающее уникальность, — пространство имен не обязательно должно совпадать с реальным URL-адресом какого-либо сетевого ресурса). Для изменения пространства имен рекомендуется использовать свойство Namespace атрибута [WebService]. Этот атрибут должен предшествовать объявлению класса веб-службы. Например:

```
[WebMethod(Namespace='http://MyFirm.ru/WebServices')]
WebService1 = class(System.Web.Services.WebService)
```

После щелчка на ссылке Описание службы в окне, показанном на рис. 13.1, появляется страница, представленная на рис. 13.3.

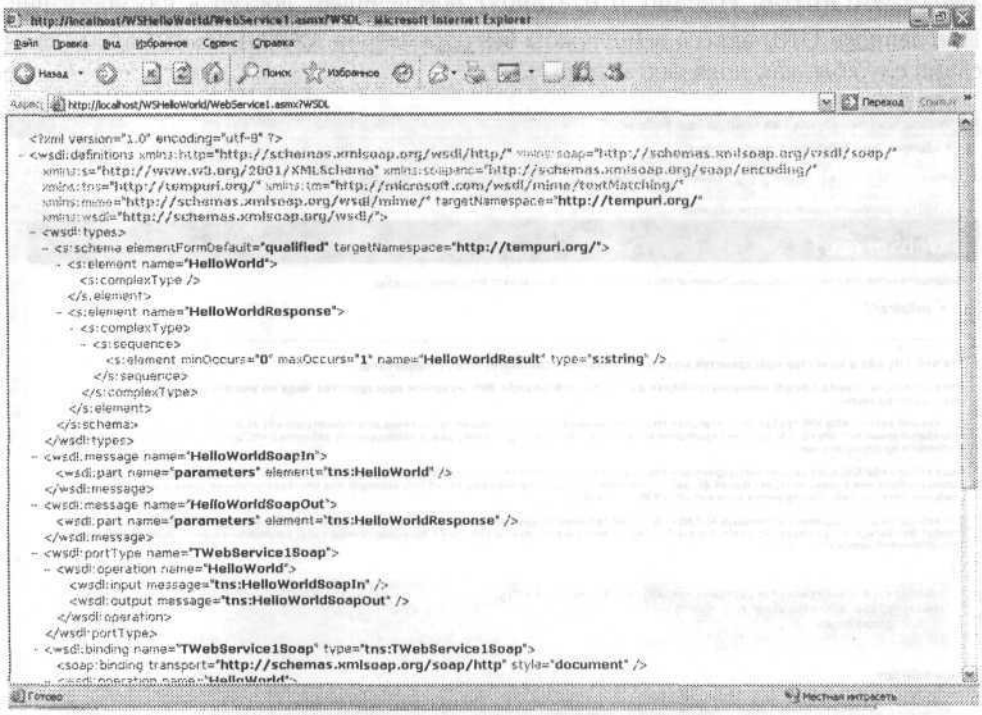


Рис. 13.3. Описание службы на языке WSDL

Обратите внимание на URL-адрес этой страницы: в нем присутствуют символы `?WSDL`, предписывающие вывести описание службы на языке WSDL (Web Services Description Language — язык описания веб-служб). Это описание необходимо различным инструментам (в том числе — IDE Delphi), способным автоматически создавать классы и интерфейсы.

Для проверки вновь созданной службы вернитесь на предыдущую страницу (см. рис. 13.2) и щелкните на ссылке HelloWorld. Появится окно, показанное на рис. 13.4.

Щелчок на кнопке Вызвать этого окна приведет к появлению следующей страницы (рис. 13.5).

Обращение к службе, как это явствует из текста на странице, изображенной на рис. 13.4, осуществлялось по протоколу HTTP методом POST. Средства этого протокола позволяют обмениваться с веб-службой данными примитивных типов (строками, числами и т. п.). На этой странице также представлен пример запроса к службе, сделанного с помощью протокола SOAP (Simple Object Access Protocol — простой протокол доступа к объекту). Протокол SOAP использует средства языка XML для обмена с веб-службой данными любой сложности, в том числе данными пользовательских типов, двоичными файлами, наборами данных и т. п.

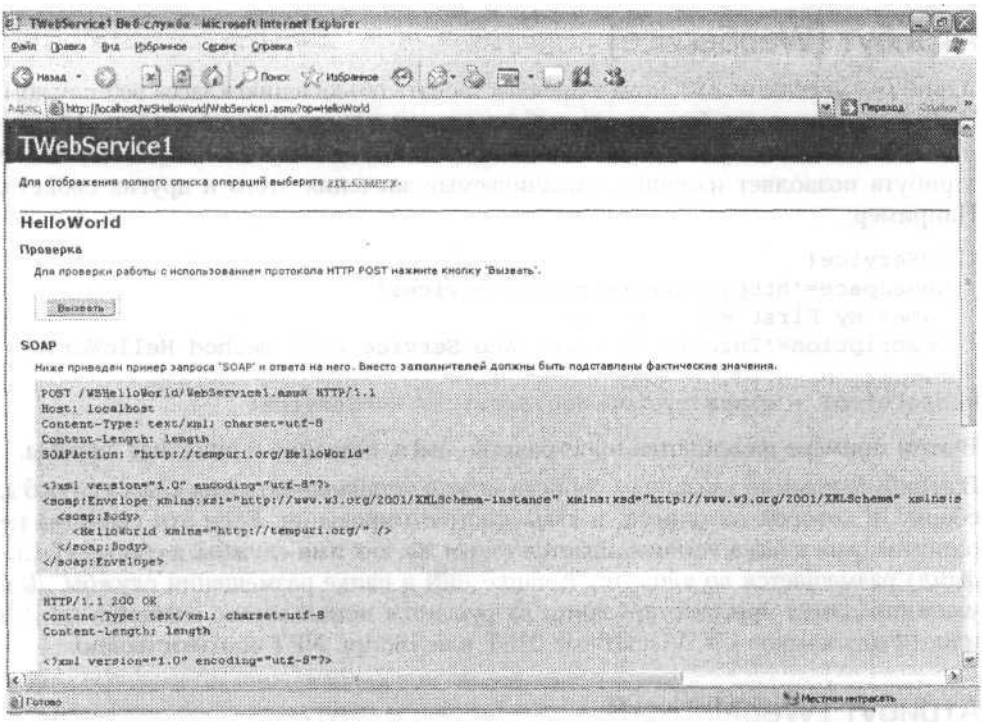


Рис. 13.4. Проверка веб-службы

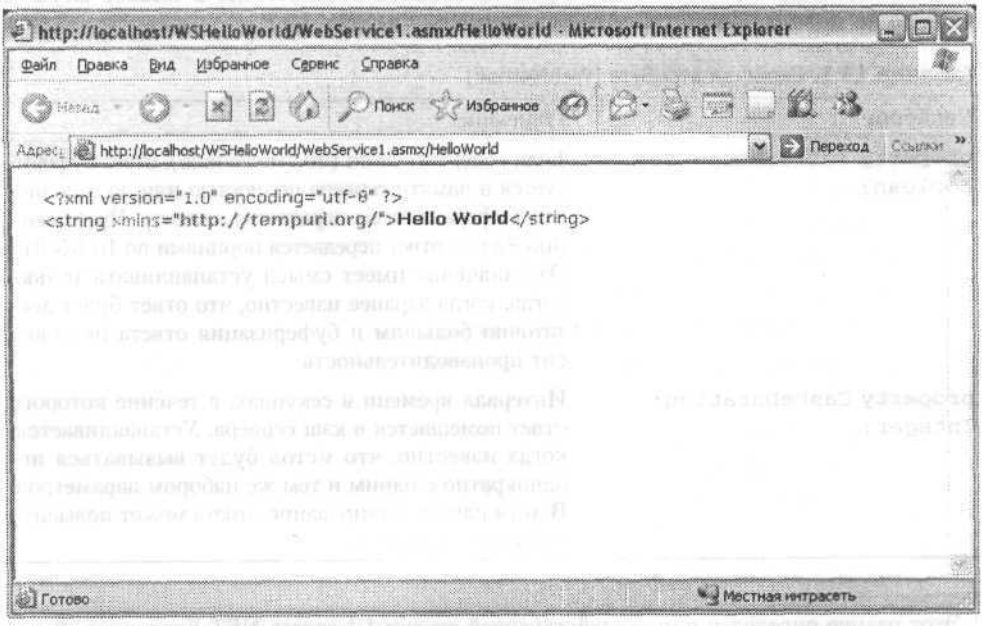


Рис. 13.5. Результат обращения к службе по протоколу HTTP POST

Атрибут [WebService]

Атрибут `[WebService]` может предшествовать объявлению класса веб-службы. Если он опущен, служба получает набор умалчиваемых значений свойств. Например, свойство `Namespace` получает значение `http://tempuri.org`. Объявление атрибута позволяет изменить умалчиваемые значения этого и других свойств. Например:

```
[WebService(
  NameSpace='http://MyFirm.ru/WebServices'
  Name='My First Web Service'
  Description='This is My First Web Service with method HelloWorld'
  )]
WebService1 = class(System.Web.Services.WebService)
```

В этом примере изменяются пространство имен, название и описание службы.

В атрибуте помимо указанных свойств можно устанавливать имя класса службы, сборку, в которой он описан, и язык программирования. Если эти свойства не описаны, имя класса устанавливается таким же, как имя службы, а сборка (DLL-файл) размещается во вложенной папке **/BIN** в папке размещения службы. Для указания языка программирования разрешается использовать значения `C#`, `VB` или `JS` для языков C#, VisualBasic .NET или JScript .NET соответственно.

Атрибут [WebMethod]

Атрибут `[WebMethod]` должен предшествовать объявлению в классе службы любого ее метода. Свойства атрибута описаны в табл. 13.1.

Таблица 13.1. Свойства атрибута [WebMethod]

Свойство	Описание
`property BufferResponse: Boolean;`	Если содержит `True` (по умолчанию), ответ буферизуется в памяти сервера полностью или до заполнения буфера, а затем передается клиенту. При значении `False` ответ передается порциями по 16 Кбайт[1]. Это значение имеет смысл устанавливать только тогда, когда заранее известно, что ответ будет достаточно большим и буферизация ответа не повысит производительность
`property CasheDuration: Integer;`	Интервал времени в секундах, в течение которого ответ помещается в кэш сервера. Устанавливается, когда известно, что метод будет вызываться неоднократно с одним и тем же набором параметров. В этом случае кэширование ответа может повысить производительность

[1] Этот размер определен в ныне действующей версии 1.1 среды .NET Framework. В следующих версиях он может быть изменен.

Свойство	Описание
`property Description: String;`	Содержит описание метода
`property EnableSession: Boolean;`	Если содержит `True`, метод может использовать объект `Session`
`property MessageName; String;`	Задает псевдоним имени. Широко используется для перегруженных методов, так как протокол SOAP не различает перегруженные методы и всегда обращается к первому из них
`property TransactionOption: TransactionOption;`	Это свойство позволяет методу работать подобно транзакциям базы данных

Пример использования свойства `MessageName`:

```
[WebMethod(MessageName='AddInt')]
function Add(A, B: Integer): Integer; overload;
[WebMethod(MessageName='DoubleInt')]
function Add(A, B: Double): Double; overload;
```

Если бы свойство `MessageName` не применялось, всегда вызывался бы только первый метод `Add`, так как SOAP не анализирует типы параметров обращения и определяет метод только по его имени.

Использование веб-служб

Для использования веб-службы нужно выполнить перечисленные ниже действия:

1. Найти службу (то есть узнать ее URL-адрес).
2. Получить описание службы на языке WSDL.
3. Изучить описание и создать так называемый прокси-класс.
4. Обратиться к нужным методам службы с помощью средств прокси-класса.

Напомню, что для получения описания веб-службы нужно с помощью браузера обратиться по URL-адресу службы, добавив в конце строки вызова символы `?WSDL`. Это описание может быть довольно пространным (даже описание созданной в этой главе простейшей службы WSHelloWorld занимает более одной страницы). К счастью, среда Delphi способна анализировать описание и автоматически создавать прокси-класс.

Создание прокси-класса

Прокси-классом называется специальный класс, создаваемый на основе описания веб-службы и играющий роль посредника[1] между клиентской программой

[1] Proxy в буквальном переводе с английского означает *представитель, уполномоченный, заместитель.*

и службой. В рамках этого класса для каждого метода веб-службы создаются по три метода — одноименный метод `Method`, а также методы `BeginMethod` и `EndMethod`. Первый используется для синхронного обращения к службе, два других — для асинхронного обращения. При синхронном вызове управление передается методу веб-службы, и работа клиентской программы приостанавливается до завершения этого метода. При асинхронном вызове метод веб-службы стартует при вызове `BeginMethod`, но выполнение клиентской программы продолжается. Работа метода завершается вызовом `EndMethod`.

Как уже говорилось, анализ описания и ручное кодирование прокси-класса могут быть довольно трудоемким процессом, поэтому далее описываются соответствующие инструментальные средства Delphi.

К веб-службе можно обратиться из клиентской программы, реализованной в виде приложения ASP.NET или приложения WinForms. Для конкретности используем приложение WinForms:

1. Создайте новый проект WinForms (File ▸ New ▸ Windows Forms Application — Delphi for .NET). На пустой форме расположите кнопку и надпись. Сохраните проект на диске.
2. Выберите команду Project ▸ Add Web Reference. В результате появится диалоговое окно, показанное на рис. 13.6.

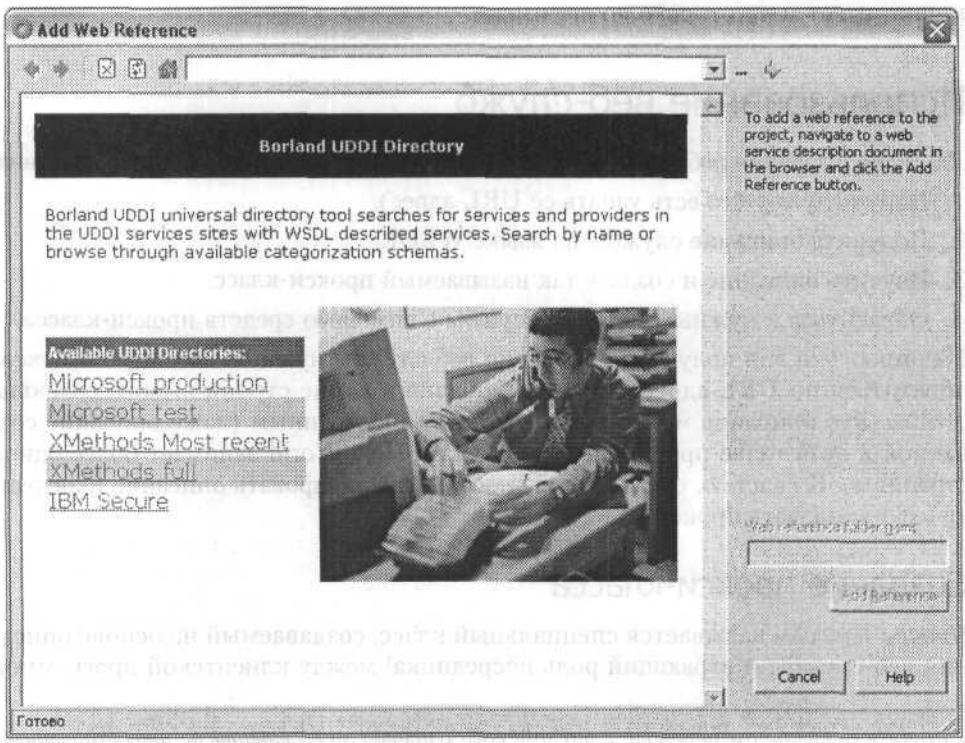


Рис. 13.6. Диалоговое окно UDDI

В верхней строке этого окна следует указать URL-адрес веб-службы или выбрать описание службы в одной из доступных папок UDDI (Universal Description, Discovery and Integration – универсальное описание, обнаружение и интеграция). UDDI представляет собой открытый системный реестр, предназначенный для хранения информации о веб-службах.

3. Введите в верхней строке такой URL-адрес:

```
http://localhost/WSHelloWorld/WebService1.asmx?WSDL
```

После щелчка на расположенной справа синей стрелке появится окно, показанное на рис. 13.7.

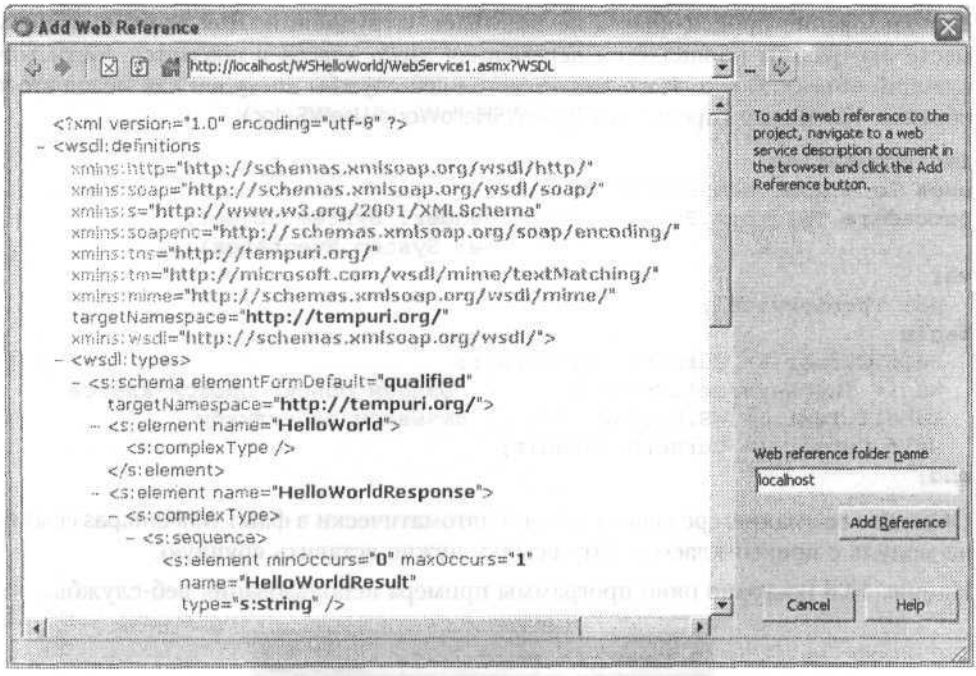


Рис. 13.7. Окно с описанием веб-службы

4. Щелкните на кнопке Add Reference, чтобы среда Delphi создала прокси-класс.

ПРИМЕЧАНИЕ

Если вы работаете с локализованной (русифицированной) версией Windows, щелчок на кнопке может вызвать сообщение об ошибке. Связано это с тем, что в процессе создания прокси-класса среда Delphi создает временный файл в папке, указанной в переменной среды `Temp`. Маршрут доступа к этому файлу по умолчанию включает имя пользователя и папку Мои документы, то есть содержит кириллицу. В результате среда не сможет создать временный файл и прокси-класс. В этом случае измените переменную `Temp`, вызвав командой Пуск ▸ Панель управления ▸ Система окно Свойства системы и на вкладке Дополнительно этого окна щелкнув на кнопке Переменные среды. Для переменной `Temp` укажите такой маршрут: `C:\Temp`.

Просматривая проект в окне менеджера проектов, можно увидеть, что среда создала и добавила к проекту несколько новых файлов:

- WebService1.map — справочная информация о веб-службе;
- WebService1.wsdl — описание службы на языке WSDL;
- localhost.WebService1.pas — код прокси-класса.

Эти файлы располагаются в папке Web References\localhost, вложенной в папку размещения проекта.

Использование прокси-класса

Использование прокси-класса не вызывает затруднений. Для этого в нужном месте программы объявляется переменная этого класса и создается соответствующий объект. После этого любой метод веб-службы доступен как метод этого объекта. Например (проект Ch13\UseWSHelloWorld\UseWS.dpr):

```
implementation
uses localhost.WebServise1;
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  WS: TWebService1;
begin
  Self.Cursor := Cursors.WaitCursor;
  WS := TWebService1.Create;     // Создаем объект прокси-класса
  Label1.Text := WS.HelloWorld; // Вызываем веб-службу
  Self.Cursor := Cursors.Default;
end;
```

Обратите внимание: среда не вставляет автоматически в файл WinForm.pas ссылку на модуль с прокси-классом. Эту ссылку нужно вставить вручную.

На рис. 13.8 показано окно программы примера использования веб-службы.

Рис. 13.8. Использование веб-службы

Создание пользовательских элементов управления

14

В этой главе описывается процесс создания и использования пользовательских элементов управления. *Пользовательский элемент управления* — это страница ASP.NET, преобразованная в элемент управления. Основное назначение таких элементов — многократное использования их содержимого и логики на нескольких страницах. Например, в большинстве веб-узлов на страницах отображаются одинаковые верхние и нижние колонтитулы, левые и правые панели. Технология пользовательских элементов управления позволяет один раз создать такие колонтитулы и панели, а затем использовать их на любой странице узла.

Пример простого пользовательского элемента управления

В качестве примера создадим простой пользовательский элемент управления, помещающий на веб-страницу текстовую строку (проект Ch14\UserControl\UserControl.dpr):

1. Начните новое приложение ASP.NET и сохраните его на диске. Затем выберите команду File ▸ New ▸ Other и в открывшемся окне на странице, связанной с узлом New ASP.NET Files, выберите значок ASP.NET User Control, как показано на рис. 14.1.
2. После щелчка на кнопке OK среда Delphi создаст заготовку нового пользовательского элемента управления: в проекте появятся файл WebUserControl1.ascx и связанный с ним файл WebUserControl1.pas. Сохраните файл пользовательского элемента под именем SimpleUserControl. Имена ascx- и pas-файлов соответствующим образом изменятся.

   Для наполнения нового пользовательского элемента управления используются те же средства редактора форм, что и при создании обычной страницы ASP.NET: на нем можно размещать элементы категорий HTML Elements и Web Controls, а также редактировать саму страницу (ascx-файл).

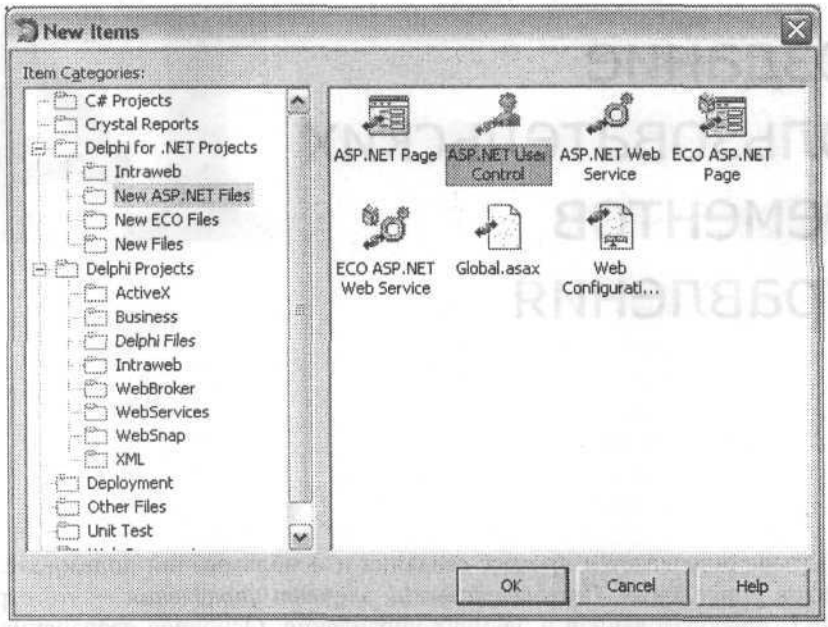


Рис. 14.1. Активизация эксперта создания пользовательского элемента

3. Добавим в наш элемент текст: щелкните в окне редактора форм и прямо на поверхности элемента введите строку

```
Это - пример простого пользовательского элемента управления.
```

4. Теперь вернитесь к странице WebForm1 и добавьте такую строку:

```
Эта страница демонстрирует простой пользовательский элемент
управления.
```

5. Завершите ввод нажатием клавиши Enter (для бóльшей наглядности можно нажать эту клавишу дважды) и выберите команду Insert ▸ Insert User Control. Появится окно, показанное на рис. 14.2.

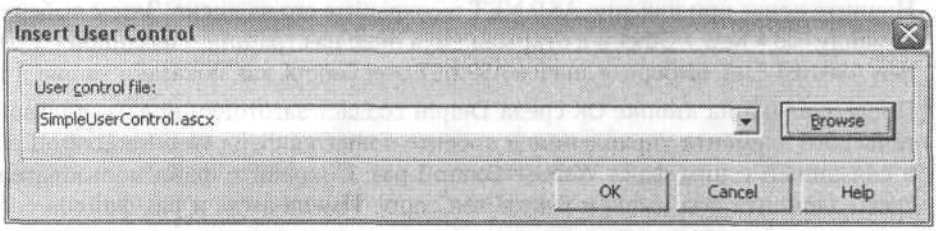


Рис. 14.2. Диалоговое окно выбора пользовательского элемента управления

6. Убедитесь, что в поле ввода содержится ссылка на файл SimpleUserControl.ascx, и щелкните на кнопке OK. В результате редактор формы должен выглядеть так, как показано на рис. 14.3.

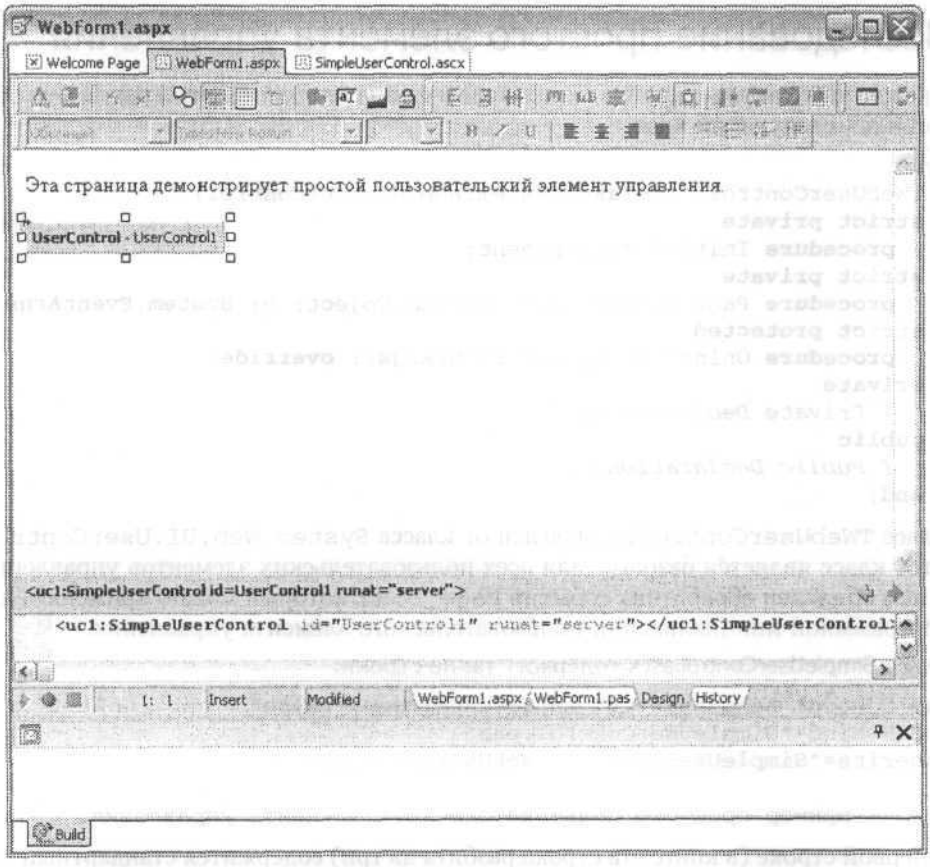


Рис. 14.3. Окно редактора форм с пользовательским элементом управления

После запуска приложения появится окно, показанное на рис. 14.4.

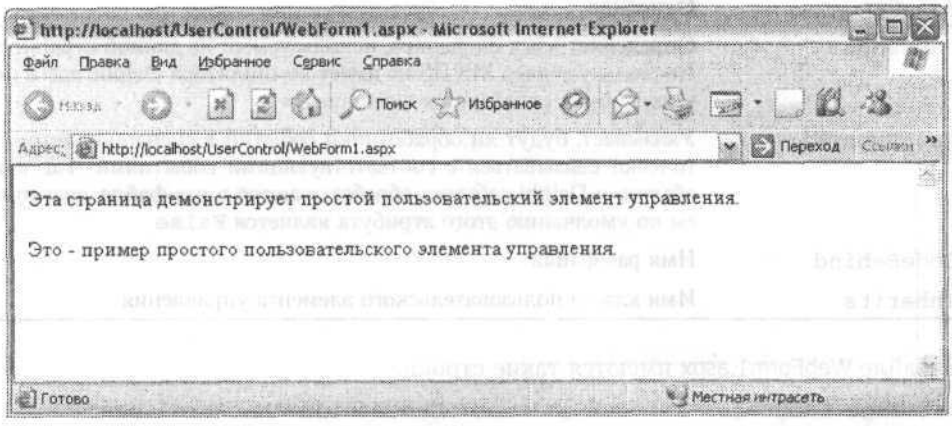


Рис. 14.4. Пользовательский элемент управления на странице в окне браузера

Исследование простого элемента управления

Прежде всего активизируйте окно с кодом файла SimpleUserControl.pas. В нем объявляется следующий класс:

```
type
  TWebUserControl1 = class(System.Web.UI.UserControl)
  strict private
    procedure InitializeComponent;
  strict private
    procedure Page_Load(sender: System.Object; e: System.EventArgs);
  strict protected
    procedure OnInit(e: System.EventArgs); override;
  private
    { Private Declarations }
  public
    { Public Declarations }
  end;
```

Класс `TWebUserControl1` порожден от класса `System.Web.UI.UserControl`. Этот класс является базовым для всех пользовательских элементов управления. В нем объявлен обработчик события `Page_Load`, который можно применять для программной инициализации пользовательского элемента управления.

Файл SimpleUserControl.ascx содержит такие строки:

```
<%@ Control Language="c#" AutoEventWireup="false"
CodeBehind="SimpleUserControl.pas"
Inherits="SimpleUserControl.TWebUserControl1"%>

 Это - пример простого пользовательского элемента управления.
```

В первой строке (в книге эта строка разбита на три) содержится стандартный для ASP.NET дескриптор `Control`. Атрибуты дескриптора описаны в табл. 14.1.

Таблица 14.1. Атрибуты дескриптора Control

Атрибут	Описание
`Language`	Определяет язык сценариев, используемый на данной странице. Поскольку сервер MS IIS не имеет компилятора Delphi, здесь по умолчанию используется значение `C#`
`AutoEventWireup`	Указывает, будут ли обработчики событий в сценариях автоматически связываться с соответствующими событиями. Так как обычно в Delphi события обрабатываются в pas-файле, значением по умолчанию этого атрибута является `False`
`CodeBehind`	Имя pas-файла
`Inherits`	Имя класса пользовательского элемента управления

В файле WebForm1.aspx имеются такие строки:

```
<%@ Page Language="c#" Debug="true" Codebehind="WebForm1.pas"
AutoEventWireup="false" Inherits="WebForm1.TWebForm1"%>
```

```
<%@ Register tagprefix="uc1" tagname="SimpleUserControl"
Src="SimpleUserControl.ascx"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>
  <body bgcolor="#ffffff" ms_positioning="GridLayout">
    <form runat="server">
      <p>Эта страница демонстрирует простой пользовательский элемент
      управления.</p>
      <p>
        <uc1:SimpleUserControl id="UserControl1" runat="server">
        </uc1:SimpleUserControl>
      </p>
    </form>
  </body>
</html>
```

В его начале располагается дескриптор `Register`, с помощью которого осуществляется регистрация пользовательского элемента управления. Атрибуты дескриптора описаны в табл. 14.2.

Таблица 14.2. Атрибуты дескриптора Register

Атрибут	Описание
`TagPrefix`	Определяет префикс, предназначенный для маркировки дескрипторов, описывающих пользовательский элемент
`TagName`	Задает имя пользовательского элемента
`Src`	Указывает файл с кодом пользовательского элемента

Сам пользовательский элемент управления добавляется на веб-страницу следующим образом:

```
<uc1:SimpleUserControl id="UserControl1" runat="server">
</uc1:SimpleUserControl>
```

Этот дескриптор начинается значением атрибута `TagPrefix` (в нашем случае — `uc1`), затем следует имя элемента (`SimpleUserControl`).

Элемент управления для регистрации пользователя

В этом разделе описывается относительно сложный элемент, содержащий встроенные элементы и логику (проект Ch14\UCLogin\UCLogin.dpr). Он предназначен для регистрации пользователя и, таким образом, затрагивает проблему безопасности приложений ASP.NET. В главе 15 эта проблема рассматривается более подробно. Страница, на которой используется этот элемент, показана на рис. 14.5.

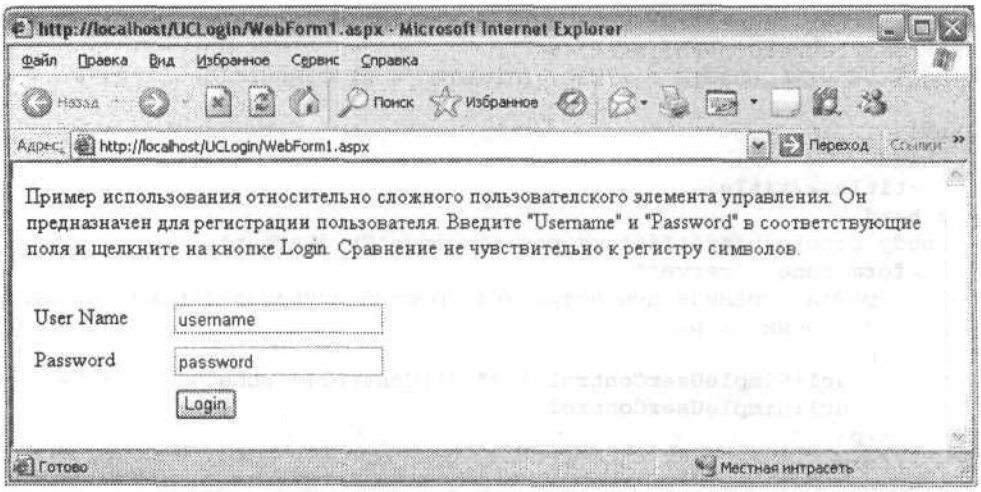


Рис. 14.5. Использование элемента UserControlLogin

1. Начните новое приложение ASP.NET и сохраните его на диске под именем `UCLogin`.
2. *С помощью окна, открываемого командой* File ▸ New ▸ Other, *запустите эксперт* создания пользовательских компонентов (значок ASP.NET User Control). Сохраните файл элемента под именем UserControlLogin.ascx.
3. Поместите на поверхность элемента панель `Panel` (категория Web Controls). Панель послужит контейнером для вставляемых в наш элемент других элементов. В свойстве `ID` панели укажите значение `LoginPanel`.
4. На панель поместите две надписи `Label`, два поля `TextBox`, два элемента `RequiredFieldValidator` и кнопку `Button` так, как показано на рис. 14.6 (все элементы из категории Web Controls).
5. Измените умалчиваемые значения свойств элементов следующим образом:
   - `Label1.Text = User Name;`
   - `Label2.Text = Password;`
   - `TextBox1.ID = tbUserName;`
   - `TextBox2.ID = tbPassword;`
   - `RequiredFielValidator1.ControlToValidate = tbUserName;`
   - `RequiredFielValidator2.ControlToValidate = tbPassword;`
   - `RequiredFielValidator1.ErrorMassage = Поле User Name не может быть пустым!;`
   - `RequiredFielValidator2.ErrorMessage = Поле Password не может быть пустым!;`
   - `Button1.ID = LoginButon;`
   - `Button1.Text = Login.`

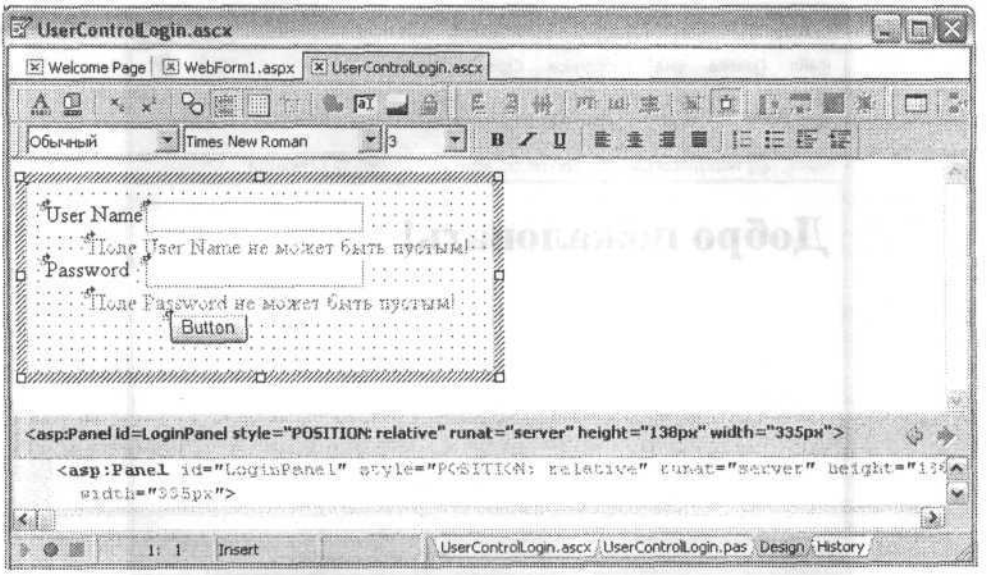


Рис. 14.6. Вид элемента в окне редактора форм

6. Для кнопки LoginButton напишите такой обработчик:

```
procedure TWebUserControl1.LoginButton_Click(sender: System.Object;
                                             e: System.EventArgs);
begin
      DoLogin(tbUserName.Text, tbPassword.Text);
end;
```

Процедура DoLogin определена следующим образом:

```
procedure TWebUserControl1.DoLogin(UserName, Password: String);
begin
  if (System.String.Compare(UserName, 'Username', True) = 0) and
     (System.String.Compare(Password, 'Password', True) = 0) then
       Response.Redirect('Welcom.aspx') else
       Response.Redirect('ErrorPage.aspx');
end;
```

В процедуре DoLogin используется метод Compare класса System.String. Он принимает две сравниваемые строки и логический признак, разрешающий игнорировать разницу в регистре букв сравниваемых строк. Метод возвращает 0, если строки совпадают.

Если пользователь введет правильные данные, он методом Response.Redirect() будет перенаправлен на страницу Welcom.aspx, в противном случае — на страницу ErrorPage.aspx. Первая страница содержит приветствие (рис. 14.7), а вот вторая — сообщение об ошибке и новый экземпляр элемента UserControlLogin.ascx (рис. 14.8).

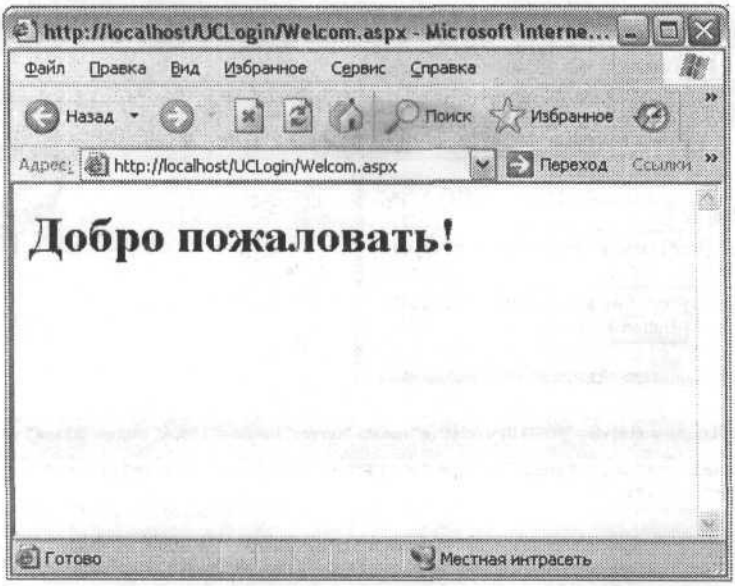


Рис. 14.7. Реакция на ввод пользователя — страница Welcom.aspx

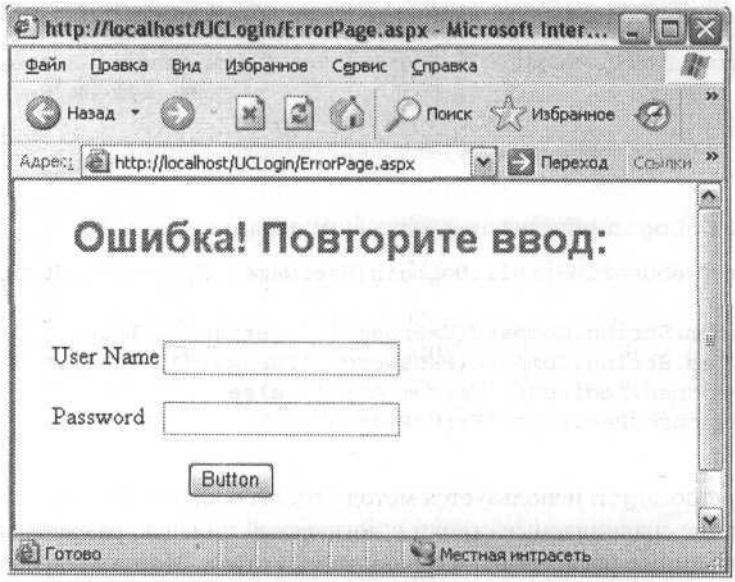


Рис. 14.8. Реакция на ввод пользователя — страница ErrorPage.aspx

Обе страницы должны быть добавлены в проект с помощью команды File ▸ New ▸ Other ▸ ASP.NET Page.

15 Защита приложений ASP.NET

Безопасность веб-приложений является их важнейшим аспектом. Незащищенные веб-приложения уязвимы для сетевых атак, могущих привести к необратимым нарушениям целостности данных. В этой главе обсуждается проблема защиты приложений ASP.NET от несанкционированного доступа.

Способы защиты приложений ASP.NET

Существует несколько способов защиты приложений ASP.NET, но все они в той или иной форме основаны на *аутентификации* пользователя и его *авторизации*. Под аутентификацией понимают процесс идентификации пользователя и проверки его *роли*. Определить роль пользователя — означает отнести его к той или иной группе пользователей и наделить его правами, присущими этой группе. Например, в бизнес-приложениях клерки имеют ограниченные права относительно размеров заключаемых сделок, руководители отделов имеют права проведения крупных сделок, а высшее руководство компании — неограниченные права.

Результат аутентификации пользователя позволяет выяснить его идентификатор, иногда называемый *учетной записью пользователя*, и его роль. Результат авторизации позволяет на основе идентификатора пользователя и его роли определить, к каким ресурсам и на каком уровне он имеет право доступа.

В ASP.NET существует три формы аутентификации: аутентификация Windows, аутентификация на основе форм и аутентификация по паспорту. Вид аутентификации определяется дескриптором **authentication** и его параметром `mode`, который может принимать значения `Windows`, `Forms`, `Passport` и `None`. Этот

дескриптор помещается в файл Web.config, которым снабжается каждое приложение ASP.NET.

Далее рассматриваются указанные формы аутентификации.

Аутентификация Windows

Аутентификация Windows может использоваться только при работе машины пользователя под управлением этой операционной системы, а приложения — под управлением MS IIS.

ПРИМЕЧАНИЕ

По умолчанию каждое приложение ASP.NET в файле Web.config имеет дескриптор вида `<authentication mode="Windows"/>`. Этот дескриптор устанавливает аутентификацию Windows, но не защищает приложение ASP.NET.

Для использования аутентификации Windows нужно запустить утилиту Internet Information Services (Пуск ▸ Панель управления ▸ Администрирование), раскрыть узел с именем машины, затем узел Веб-узлы и узел Веб-узлы по умолчанию, разыскать папку с защищаемым веб-приложением и в ее локальном меню выбрать команду Свойства. В новом окне на вкладке Безопасность каталога щелкните на кнопке Изменить. В результате появится окно Методы проверки подлинности (рис. 15.1).

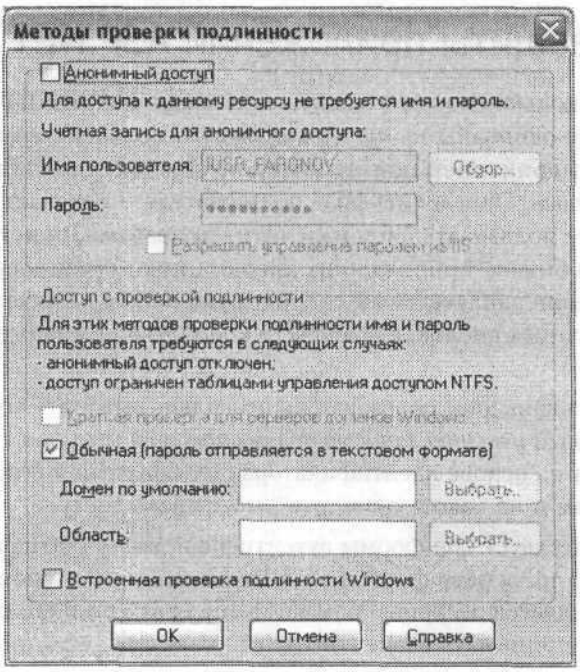


Рис. 15.1. Диалоговое окно Методы проверки подлинности

В этом окне следует сбросить флажок Анонимный доступ и установить флажок Обычная (пароль отправляется в текстовом формате). После этого при попытке обращения к защищенному ресурсу появится окно, показанное на рис. 15.2.

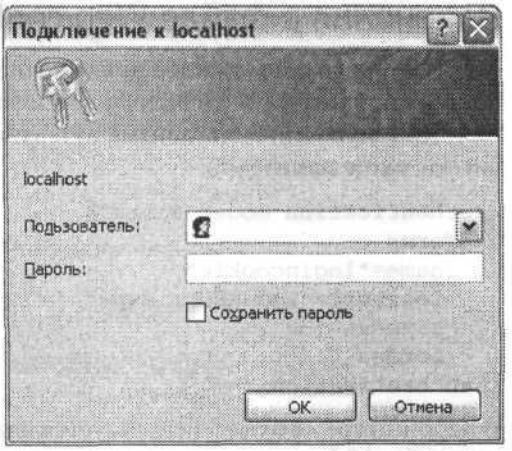


Рис. 15.2. Диалоговое окно для регистрации пользователя

Аутентификация Windows применяется только в локальных сетях, построенных по технологии интранет, так как предъявляет жесткие требования к клиентским машинам (они должны работать под управлением Windows и использовать браузер Internet Explorer). При создании интернет-сайта следует помнить, что далеко не все клиенты могут удовлетворять этим требованиям. Кроме того, аутентификация пользователя на основе учетной записи Windows не всегда приемлема с точки зрения безопасности.

Аутентификация на основе форм

Аутентификация на основе форм не предъявляет никаких требований к клиентам и позволяет разработчику приложения ASP.NET использовать любой механизм для хранения имен пользователей и их паролей, будь то XML-файл (в том числе файл Web.config), таблица базы данных, реестр Windows и т. д.

Для реализации этого вида аутентификации разработчик приложения ASP.NET должен создать страницу для регистрации пользователя и предъявлять ее пользователю при первом обращении к сайту. Если регистрация прошла успешно, браузеру пользователя отсылается так называемый «регистрационный билет» в виде файла *cookie*[1]. При обращении к любой другой странице сайта сервер получает из этого файла регистрационный билет пользователя и блокирует его повторную регистрацию.

Для использования аутентификации на основе форм необходимо проделать следующее:

1. Внести изменения в раздел <**authentication**> конфигурационного файла Web.config.
2. Создать в проекте страницу для регистрации пользователя.
3. Защитить остальные страницы приложения от несанкционированного доступа.

[1] Cookie (читается *куки*) в буквальном переводе с английского означает «печенье». В терминологии Интернета этим термином обозначается некоторая информация, предназначенная для временного размещения на машине пользователя.

Изменения в файле Web.config

Как уже говорилось, каждое приложение ASP.NET имеет индивидуальный конфигурационный файл Web.config, расположенный в папке размещения приложения. Для включения механизма аутентификации на основе форм в него нужно внести такие изменения:

```
<authentication mode="Forms">
  <forms
    name="logincookie"
    loginUrl="LoginForm.aspx"
    timeout="30">
  </forms>
</authentication>
```

В дескрипторе <**forms**> используется ряд атрибутов, описание которых приведено в табл. 15.1.

Таблица 15.1. Атрибуты дескриптора <forms>

Атрибут	Описание
`name`	Имя используемого файла cookie. По умолчанию принимается значение .ASPXAUTH
`loginUrl`	URL-адрес страницы, на которую переадресуется запрос, если файл cookie не найден
`timeout`	Период времени в минутах, по истечении которого файл cookie удаляется
`protection`	Тип шифрования файла cookie. Допустимые значения: `All` (все); `None` (отсутствует); `Encryption` (шифрование); `Validation` (проверка)
`path`	Маршрут доступа к файлу cookie
`requireSSL`	Если содержит `True`, для передачи cookie используется протокол повышенной секретности SSL (Socket Security Layer — секретный слой сокета)
`slidingExpiration`	Если содержит `True`, атрибут `timeout` обновляется после каждого запроса

Индивидуальная защита страниц

Обычно сайт содержит множество страниц, каждая из которых имеет свой URL-адрес. Ничто не мешает злоумышленнику обратиться непосредственно к нужной странице, не проходя процедуру регистрации. Чтобы исключить возможность доступа к странице клиента, который не прошел процедуру регистрации, можно написать такой обработчик события `Page_Load`:

```
procedure TWebForm.Page_load(Sender: Object;
                             e: System.EventArgs);
```

```
begin
  if not User.Identity.IsAuthenticated then // Клиент
                                            // зарегистрирован?
    Response.Redirect('LoginForm.aspx');    //-Нет.
                                            // Регистрируем его
end;
```

Подобный обработчик следует написать для каждой защищаемой страницы сайта. Если защищаются все страницы, вместо указанных обработчиков можно дополнить конфигурационный файл следующим дескриптором:

```
<authorization>
  <deny users="?" />
</authorization>
```

В этом случае знак ? указывает, что страницы защищаются от любых незарегистрированных клиентов. Заметьте, что при такой защите в составе вашего веб-узла должна быть регистрационная страница в файле, определяемом атрибутом `loginUrl` дескриптора **<forms>**, а если этот атрибут не задан — в файле Login.aspx.

В обработчике используется свойство `User` объекта `Page`, содержащее объект, реализующий интерфейс `IPrincipal`. Подсвойство `Identity` этого свойства исполняет интерфейс `IIdentity`, с помощью которого программа может определить не только факт регистрации пользователя (свойство `IsAutheticated`), но и некоторые другие параметры — его имя, регистрационное удостоверение и т. п.

Создание страницы регистрации

На странице регистрации пользователя необходимо предусмотреть элементы управления для ввода необходимой регистрационной информации. Чаще всего пользователь должен указать свое регистрационное имя и пароль, поэтому непременными атрибутами окон регистрации являются два текстовых поля. Кроме того, в некоторых случаях на страницу помещается флажок, с помощью которого пользователь может указать на необходимость создания постоянного файла cookie. Постоянный файл не уничтожается браузером и позволяет получить доступ к защищаемым ресурсам без повторной регистрации. Таким образом, типичная страница регистрации обычно выглядит так, как показано на рис. 15.3.

Создание регистрационного удостоверения пользователя

После щелчка на кнопке на странице регистрации осуществляется аутентификация пользователя и, если она проходит успешно, создается регистрационное удостоверение пользователя. В регистрационное удостоверение переносятся вся регистрационная информация и другие сведения, необходимые для регистрации. Регистрационное удостоверение размещается на браузере клиента в виде файла cookie. Оставив пока в стороне процедуру аутентификации (далее рассматриваются ее различные реализации), рассмотрим создание регистрационного удостоверения и его размещение в файле cookie.

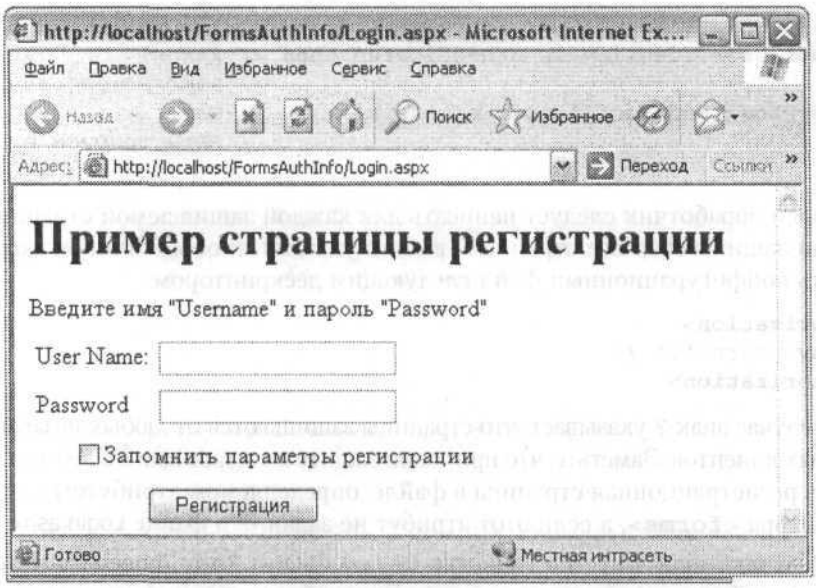


Рис. 15.3. Пример страницы для регистрации пользователя

Эти действия реализуются в такой процедуре:

```
class procedure RedirectFromLoginPage(UserName: String;
                    PersistentCookie: Boolean); overload;
```

Эта процедура объявлена в классе `FormsAuthentication` как статическая, то есть к ней можно обращаться без создания экземпляра класса. При обращении к ней передаются регистрационное имя пользователя и логический признак, указывающий на необходимость создания постоянного файла cookie. В перегруженном варианте процедуре в качестве третьего параметра передается строковое выражение, указывающее маршрут размещения файла cookie. Процедура создает стандартный файл cookie и возвращает его браузеру пользователя.

Вот типичный пример использования процедуры:

```
procedure TLoginForm.btnLogin_Click(Sender: &Object;
                                    e: EventArgs);
begin
  if UserLoginValid then // Регистрационные параметры верные?
    FormsAuthentication.RedirectFromLoginPage( // -Да
          tbUsername.Text, cbRemember.Checked);
end;
```

Предполагается, что в логической функции `UserLoginValid` осуществляется проверка правильности введенной регистрационной информации, поле `tbUsername` содержит регистрационное имя пользователя, а флажок `cbRemember` управляет созданием постоянного файла cookie.

После успешного выполнения метода RedirectFromLoginPage управление передается на страницу с именем default.aspx. Обработчик ее события Page_Load имеет такой вид (проект Ch15\FormsAuthInfo.dpr):

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
var
  FormsId: FormsIdentity;
begin
  if not User.Identity.IsAuthenticated then // Нужна регистрация?
    Response.Redirect('Login.aspx') // -Да. На странице регистрации
    else  //-Нет. Выводим информацию о пользователе:
  begin
    Response.Write('<h1>Параметры клиента:</h1>');
    Response.Write('Имя пользователя = ' +
                   User.Identity.Name + '<br>');
    Response.Write('Тип аутентификации = ' +
      User.Identity.AuthenticationType + '<br>');
      FormsId := FormsIdentity(User.Identity);
    Response.Write('Маршрут файла Cookie = ' +
      FormsId.Ticket.CookiePath + '<br>');
    Response.Write('Время уничтожения Cookie = ' +
      Convert.ToString(FormsId.Ticket.Expiration) + '<br>');
  end;
end;
```

После регистрации пользователь возвращается на страницу default.aspx, на которой выводится некоторая информация о нем (рис. 15.4).

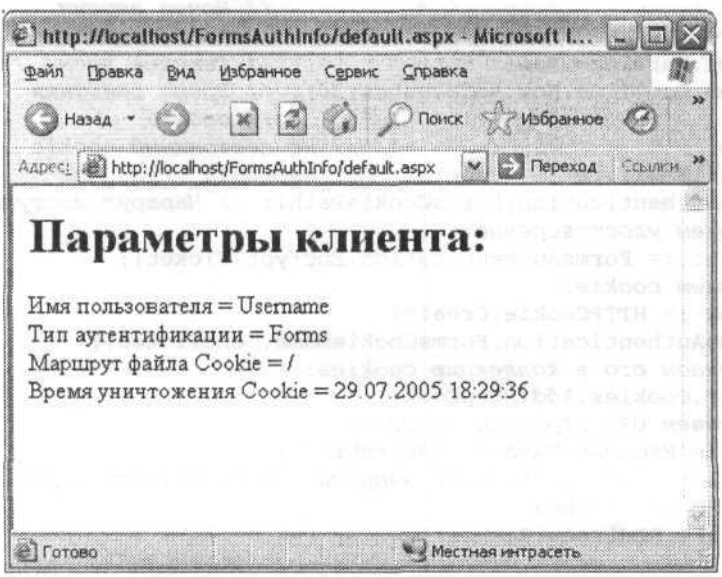


Рис. 15.4. Страница с информацией о пользователе

В классе FormsAuthentication определена следующая функция:

```
class function GetRedirectURL(UserName: String;
                              IsPersistentCookie: Boolean): String;
```

Как сказано в документации, эта функция создает стандартный файл cookie и возвращает URL страницы, которая перенаправила запрос на страницу регистрации. Она используется так:

```
Response.Redirect(GetRedirectURL('UserName', False));
```

Однако мне не удалось добиться от нее возврата на страницу запроса, если эта страница содержится в произвольном файле (не в файле default.aspx).

Возврат на страницу запроса можно реализовать при программном создании файла cookie. Например (проект Ch15\FormsAuthCreateCookie.dpr):

```
implementation
uses System.Web.Security;

procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  Ticket: FormsAuthenticationTicket;
  encTicket: String;
  HTTPCook: HTTPCookie;
  RetUrl: String;
begin
  // Создаем регистрационное удостоверение:
  Ticket := FormsAuthenticationTicket.Create(
    1,                                    // Номер версии
    'user',                               // Регистрационное имя
    System.DateTime.Now,                  // Текущее время
    System.DateTime.Now.AddMinutes(30),   // Время действия
                                          // cookie
    false,                          // Не постоянный cookie
    'mymail@mail.ru',               // UserData
    FormsAuthentication.FormsCookiePath); // Маршрут доступа
  // Шифруем удостоверение:
  encTicket := FormsAuthentication.Encrypt(Ticket);
  // Создаем cookie:
  HTTPCook := HTTPCookie.Create(
     FormsAuthentication.FormsCookieName, encTicket);
  // Помещаем его в коллекцию cookies:
  Response.Cookies.Add(HttpCook);
  // Получаем URL страницы запроса:
  RetUrl := Request.Params['ReturnUrl'];
  // Переадресуем на страницу запроса или на Welcome.aspx:
  if RetURL <> '' then
    Response.Redirect(RetUrl)
  else
    Response.Redirect('Welcome.aspx');
end;
```

При программном создании в cookie можно поместить произвольные данные (параметр `UserData`). Эти данные будут доступны на любой странице. Например (файл Welsome.aspx):

```
implementation
uses System.Web.Security;

procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
var
  FormsId: FormsIdentity;
begin
  // Получаем регистрационное удостоверение:
  FormsId := User.Identity as FormsIdentity;
  // Регистрационное имя:
  lbName.Text := FormsId.Name;
  // EMail-адрес:
  lbUserData.Text := FormsId.Ticket.UserData;
end;
```

Свойство `Page.User.Identity` определено как объект, исполняющий интерфейс `IIdentity`. Класс `FormsIdentity` также исполняет этот интерфейс, но в нем определено свойство `Ticket`, открывающее доступ к полям регистрационной записи. В первом операторе осуществляется явное приведение объекта `User.Identity` к классу `FormsIdentity`.

Обратите внимание: в приведенном обработчике события `Page_Load` нет переадресации на страницу регистрации. Тем не менее такая переадресация осуществляется автоматически, так как в файл Web.config вставлены следующие дескрипторы:

```
<authentication mode="Forms" />
<authorization>
  <deny users="?" />
</authorization>
```

Хранение аутентификационной информации в файле Web.config

До сих пор мы фактически не рассматривали процесс аутентификации пользователя. В этом и двух следующих разделах этот процесс рассматривается применительно к месту хранения аутентификационной информации.

Аутентификационную информацию можно хранить в конфигурационном файле Web.config. Для этого предусмотрены специальные дескрипторы **<creditionals>** и **<user>**. Например:

```
<authentication mode="Forms">
  <forms
    name="AuthCred" loginUrl="LoginPage.aspx" timeout="30">
    <creditionals passwordFormat="Clear">
      <user name="Sasha" password="Secret"/>
```

```
      <user name="Olga" password="secret"/>
    </creditionals>
  </forms>
</authentication>
```

Хранение паролей в незашифрованном виде опасно, так как файл Web.config может быть доступен любому, кто работает с веб-сервером. Для шифрования паролей используется метод `HashPasswordForStoringInConfigFile` класса `FormsAuthetication`, который принимает два строковых параметра (пароль и название алгоритма шифрования) и возвращает хеш-код пароля.

ПРИМЕЧАНИЕ

Хеш-кодом называется последовательность шестнадцатеричных цифр, которая однозначно определяет зашифрованный текст, но не позволяет узнать этот текст.

Существует несколько алгоритмов получения хеш-кода. Наиболее устойчивыми к дешифрированию считаются алгоритмы MD5 и SHA1. Именно эти алгоритмы можно указывать при обращении к методу `HashPasswordForStoringInConfigFile` и в атрибуте `passwordFormat` дескриптора `<creditionals>`. Вот как выглядит файл Web.config при шифровании паролей алгоритмом MD5:

```
<creditionals passwordFormat="MD5">
  <user name="Sasha"
        password="1E6947AC7FB3A9529A9726EB692C8CC5"/>
  <user name="Olga"
        password="5EBE2294ECD0E0F08EAB7690D2A6EE69"/>
</creditionals>
```

Обратите внимание на любопытную особенность хеширования: в данном случае пароли обоих пользователей различаются лишь регистром первой буквы, тем не менее они порождают совершенно разные хеш-коды.

Оба алгоритма дают код одной и той же длины, независимо от длины кодируемого пароля: 20 байт для SHA1 и 16 байт для MD5. С помощью проекта Ch15\Hash.dpr вы сможете получить хеш-код для любых пароля и алгоритма хеширования.

После создания списка пользователей и их паролей в файле Web.config их аутентификация сводится к вызову логической функции `Authenticate` класса `FormsAuthentication`, которая принимает в качестве двух строковых параметров регистрационное имя пользователя и его пароль и возвращает `True`, если данные пользователя обнаружены в файле Web.config (проект Ch15\FormsAuth\AuthCredit.dpr).

Несмотря на простоту реализации, размещение регистрационной информации в файле Web.config в серьезных проектах практически не используется. Связано это с двумя обстоятельствами. С одной стороны, файл Web.config не защищен от несанкционированного доступа и может быть случайно или намеренно поврежден. Вторая и главная причина — трудности администрирования: администратор должен вручную изменять файл при добавлении очередного пользователя или его удалении.

Хранение регистрационной информации в XML-файле

Размещение регистрационной информации в XML-файле сопряжено с теми же недостатками, что и ее хранение в файле Web.config. Тем не менее далее кратко описываются методика создания XML-файла и его использование для аутентификации.

Для создания и редактирования XML-файла подойдет любой текстовый редактор, например Блокнот Windows или текстовый редактор среды Delphi. Создайте с его помощью такой файл:

```
<users>
  <user>
    <name>Sasha</name>
    <password>Secret</password>
  </user>
  <user>
    <name>Olga</name>
    <password>secret</password>
  </user>
</users>
```

В отличие от файла Web.config, хранить такой файл можно в любой папке. В следующем примере (проект Ch15\FormsAuthXML.dpr) показано использование файла:

```
implementation
uses System.Web.Security, System.IO;

procedure TWebForm2.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  dsUsers: DataSet;
  fsUsers: FileStream;
  srUsers: StreamReader;
  drArray: array of DataRow;
  S: String;
begin
  // Проверяем заполнение полей:
  if (tbUserName.Text <> '') and
     (tbPassword.Text <> '') then
  begin
    // Создаем поток ввода:
    fsUsers := FileStream.Create(Server.MapPath(
          'users.xml'), FileMode.Open, FileAccess.Read);
    try
      // Создаем набор данных:
      dsUsers := DataSet.Create;
      // Создаем объект StreamReader:
      srUsers := StreamReader.Create(fsUsers);
      //  Читаем файл в набор данных:
      dsUsers.ReadXML(srUsers);
```

```
    // Выбираем нужную запись:
    drArray := dsUsers.Tables[0].Select(
      System.&String.Format('name=''{0}''',
                                    tbUsername.Text));
  finally
    fsUsers.Close;
  end;
  // Сравниваем пароли:
  if System.&Array(drArray).Length > 0 then
  begin
    S := String(drArray[0]['password']);
    if S = tbPassword.Text then
      FormsAuthentication.RedirectFromLoginPage(
                                tbUserName.Text, false);
  end;
 end;
end;
```

Даже беглый анализ показывает, что более сложная, чем в предыдущем случае, аутентификация не дает никаких преимуществ.

Хранение регистрационной информации в БД

Наиболее часто регистрационная информация хранится в таблицах серверной базы данных. Главными достоинствами такого подхода являются высокая защищенность конфиденциальной информации и довольно простое ее администрирование. Например, не составляет особых проблем создание страниц для регистрации новых пользователей и автоматического занесения новой регистрационной информации в БД.

В следующем примере (проект Ch15\FormsAuthBD.dpr) для хранения регистрационной информации используется небольшая таблица `users` БД IB_BIBL.gdb. В этой таблице определены два строковых поля — `user_name` и `passwrd` (слово `password` является зарезервированным для сервера InterBase и не может использоваться для именования полей). К проекту присоединены компонент `BdpConnection1`, настроенный на связь с БД, компонент `BdpCommand1`, ссылающийся в своем свойстве `Connection` на компонент `BdpConnection1`, и компонент `DataSet1`. Работа с регистрационной информацией осуществляется следующим образом:

```
procedure TWebForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
  S: String;
begin
  BdpConnection1.Close;  // Закрываем соединение
  // Формируем команду выборки нужного имени:
  BdpCommand1.CommandText :=
    'select passwrd from users where user_name= ''' +
                                  tbUserName.Text + '''';
```

```
  BdpConnection1.Open;
  // Выполняем команду:
  S := System.&String(BdpCommand1.ExecuteScalar);
  BdpConnection1.Close;
  // Сравниваем пароли:
  if S = tbPassword.Text then
    FormsAuthentication.RedirectFromLoginPage(
                                    tbUserName.Text, false);
end;
```

Аутентификация по паспорту

Аутентификация по паспорту представляет собой централизованную услугу аутентификации в Веб, предоставляемую некоторыми крупными производителями всем желающим на бесплатной основе. Наибольшую известность получила служба Microsoft Passport (далее — просто Passport), которой на сегодня пользуются уже около 200 млн пользователей во всем мире.

ПРИМЕЧАНИЕ

Чтобы создать веб-узел, поддерживающий аутентификацию по паспорту, надо стать сертифицированным партнером Microsoft и получить пакет Passport SDK. Годовая стоимость участия в программе Microsoft Certified Partner едина для всех компаний, не подлежит дополнительным скидкам или согласованиям и составляет для российских компаний 1610 долларов США (без учета НДС). Дополнительную информацию можно получить в ЗАО «Экоинвент» по тел. (095) 916-71-14.

Безусловным достоинством аутентификации по паспорту является возможность свободного перемещения между веб-узлами без повторной регистрации (если, разумеется, узлы поддерживают технологию Passport). Платформа .NET обеспечивает полную поддержку паспортов при помощи модуля `PassportAutenticationModule`. Чтобы разрешить аутентификацию по паспорту, в файл Web.config необходимо поместить следующую информацию:

```
<authentication mode="Passport">
  <passport redirectUrl="Login.aspx" />
</authentication>
```

Программирование приложений ASP.NET

16

В этой главе обсуждаются некоторые общие вопросы создания приложений ASP.NET. В частности, рассматриваются популярные объекты и классы, а также механизм управления состоянием вида приложения.

Объекты и классы приложений ASP.NET

Далее в этом разделе рассматриваются наиболее важные для программиста классы и их свойства.

Класс Page

Одним из важнейших объектов приложения ASP.NET является объект `Page`, представляющий собой экземпляр класса `TWebForm`, объявленного в pas-файле. Этот класс является наследником класса `System.Web.UI.Page` и инкапсулирует все особенности конкретной веб-формы.

Свойства класса `System.Web.UI.Page` представлены в табл. 16.1.

Таблица 16.1. Свойства класса System.Web.UI.Page

Свойство	Описание
`property Application: HttpApplicationState;`	Содержит объект-приложение `Application`
`property Cache: Cache;`	Содержит объект `Cache` приложения
`property ClientTarge: String;`	Указывает условное имя браузера клиента
`property EnableViewState: Boolean;`	Возвращает `True`, если страница сохраняет свое состояние вида

Свойство	Описание
`property ErrorPage: Strting;`	Содержит имя страницы, которую будет использовать браузер для отображения ошибок
`property IsPostBack: Boolean;`	Содержит `False`, если страница загружена впервые
`property IsValid: Boolean;`	Содержит `True`, если все компоненты коллекции `Validators` успешно завершили свою работу
`property Request: HttpRequest;`	Содержит запрос клиента
`property Response: HttpResponse;`	Содержит ответ на запрос клиента
`property Server: HttpServerUtility;`	Содержит объект `Server`
`property Session: HttpSessionState;`	Содержит объект `Session`
`property SmartNavigation: Boolean;`	Разрешает/запрещает «умную» навигацию между страницами (см. далее)
`property Trace: TraceContext;`	Содержит объект `TraceContext`, с помощью которого можно прослеживать процесс создания страницы ответа
`property User: IPrincipal;`	Содержит информацию о пользователе, запросившем данную страницу
`property Validators: ValidatorCollection;`	Содержит ссылку на коллекцию контролирующих элементов, расположенных на данной странице
`property ViewStateUserKey: String;`	Устанавливает ID пользователя в переменную состояния вида, чтобы блокировать попытку однократного щелчка для доступа к ресурсу

Как видно из таблицы, многие свойства класса являются объектами других классов, что требует пояснения.

Свойство `Application` объединяет совокупность файлов, сеансов, запросов, обработчиков и модулей в виртуальном каталоге приложения и всех его подкаталогов. Этот объект создается в ответ на каждый запрос конкретного URL-адреса в данном виртуальном каталоге и не разделяется с возможными другими запросами.

Свойство `Cache` содержит буферную память (кэш) объекта `Application`. Размещение страниц или их частей в буферной памяти в ряде случаев помогает сэкономить время ответа на запрос пользователя.

Свойство `ClientTarge` может содержать одно из следующих значений:

- `ie5` — страницу можно обрабатывать браузером Internet Explorer 5.5 и выше;
- `ie4` — страницу можно обрабатывать браузером Internet Explorer 4.0;

- `uplevel` — страницу можно обрабатывать браузером Internet Explorer 4.0 и более ранним;
- `downlevel` — страницу можно обрабатывать браузером более ранним, чем Internet Explorer 4.0.

Кроме того, в свойство можно помещать значение, описанное в файле Web.config. Например:

```
<configuration>
  <system.web>
   <clientTarget>
    <add alias="ie302" useragent="Mozilla/2.0
          (compatible; MSIE 3.02; Windows NT 3.5)" />
   </clientTarget>
  </system.web>
</configuration>
```

В этом фрагменте описывается Internet Explorer 3.02.

Свойства `Request`, `Response`, `Session` очень часто используются в программном коде и поэтому описываются в этой главе отдельно (см., соответственно, разделы «Класс HttpRequest», «Класс HttpResponse» и «Управление состоянием на уровне сеанса»).

Свойство `Server` класса `Page` определено как объект класса `HttpServerUtility`. Этот класс содержит два свойства и несколько методов, которые дают вспомогательные средства для обработки запроса пользователя.

Если в свойство `SmartNavigation` поместить значение `True`, навигация по страницам будет происходить со следующими особенностями:

- устраняется излишнее мелькание изображения;
- поддерживается постоянство позиции прокрутки при возвращении к ранее покинутой странице;
- поддерживается постоянство фокуса ввода при возвращении к ранее покинутой странице;
- в списке посещенных страниц окна просмотра сохраняется только URL-адрес последней страницы.

Свойство `Trace` облегчает отладку приложения. Если `Trace.IsEnabled` имеет значение `True`, программист может использовать метод `Trace.Write`, чтобы вывести в окне трассировки информацию о текущем состоянии приложения, как показано на рис. 16.1 (проект Ch16\Trace\Trace.dpr).

Это окно создано таким обработчиком:

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
begin
  Trace.IsEnabled := True;
  Trace.Write('Hellow');
end;
```

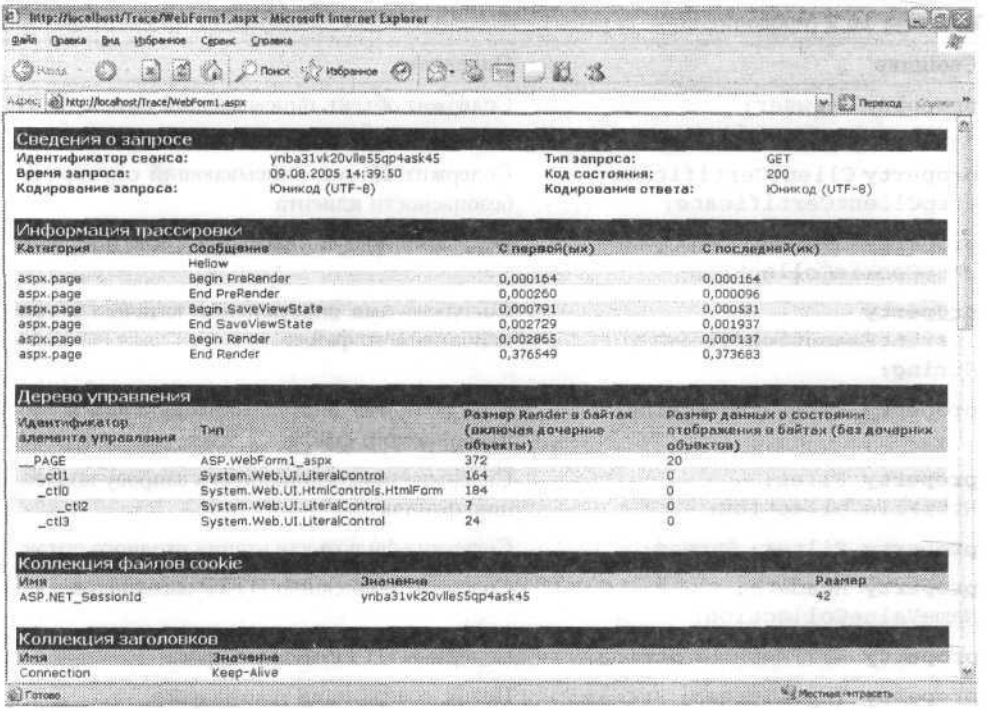


Рис. 16.1. Окно трассировки

Заметьте: по умолчанию трассировка отключена, поэтому первый оператор включает ее. Если этого не сделать, окно останется пустым.

Свойство `User` описано как объект, исполняющий интерфейс `IPrincipal`. В интерфейс входит метод `IsInRole`, который возвращает `True`, если для пользователя определена указанная роль. Кроме того, интерфейс имеет свойство `Identity`, которое представляет собой регистрационный билет пользователя (особенности использования свойства `User` описаны в главе 15).

Класс HttpRequest

Свойство `Request` объекта `Page` относится к классу `HttpRequest`. Этот класс предназначен для описания параметров запроса. Его свойства представлены в табл. 16.2.

Таблица 16.2. Свойства класса HttpRequest

Свойство	Описание
`property AcceptTypes: array of Strng;`	Определяет, какого типа сообщения может принимать клиент
`property ApplicationPath: String;`	Содержит виртуальный каталог приложения

продолжение

Таблица 16.2 (продолжение)

Свойство	Описание
`property Browser: HttpBrowserCapabilites;`	Содержит объект, описывающий возможности клиентского браузера
`property ClientCertificate: HttpClientCertificate;`	Содержит объект, описывающий сертификат безопасности клиента
`property Cookies: HttpCookieCollection;`	Открывает доступ к объектам `Cookie` клиента
`property CurrentExecutionFilePath: String;`	Содержит имя виртуального каталога и имя исполняемого файла
`property FilePath: String;`	Содержит имя виртуального каталога и имя исполняемого файла
`property Files: HttpFileCollection;`	Содержит присоединенную к запросу коллекцию файлов
`property Filter: Stream;`	Содержит фильтр для чтения входного потока
`property Headers: NameValueCollection;`	Содержит коллекцию HTTP-заголовков
`property HTTPMethod: String;`	Содержит HTTP-метод запроса
`property InputStream: Stream;`	Поток, содержащий тело запроса
`property IsAuthenticated: Boolean;`	Содержит `True`, если клиент прошел аутентификацию
`property IsSecureConnection: Boolean;`	Содержит `True`, если клиент использует протокол повышенной секретности
`property Item: String;`	Возвращает указанный объект в коллекциях `Cookies`, `Form`, `QueryString`, `ServerVariables`
`property Params: NameValueCollection;`	Содержит все члены коллекций `Cookies`, `Form`, `QueryString`, `ServerVariables`
`property Path: String;`	Содержит виртуальный каталог и исполняемый файл
`property PhisicalApplicationPath: String;`	Содержит полный путь к исполняемому файлу
`property PhisicalPath: String;`	Содержит полный путь к запросу
`property QueryString: NameValueCollection;`	Возвращает коллекцию переменных запроса
`property RawURL: String;`	Возвращает виртуальный каталог и исполняемый файл запроса
`property RequestType: String;`	Возвращает тип запроса

Свойство	Описание
`property ServerVariables: NameValueCollection;`	Возвращает все серверные переменные запроса
`property TotalBytes: Integer;`	Общее количество байтов в запросе
`property URL: Uri;`	Возвращает информацию о текущем URL-адресе
`property URLReferrer: Uri;`	Возвращает информацию об URL-адресе, с которого клиент перешел на текущий URL-адрес
`property UserAgent: String;`	Возвращает информацию об агенте браузера
`property UserHostAddress: String;`	Возвращает IP-адрес клиента
`property UserHostName: Strimg;`	Возвращает DNS-имя клиента
`property UserLanguages: array of String;`	Возвращает отсортированный набор строк с указанием предпочитаемых языков

Обработчик, представленный в листинге 16.1 (проект Ch16\RequestPropDemo\RequestPropDemo.dpr), иллюстрирует некоторые свойства объекта `Request` (рис. 16.2).

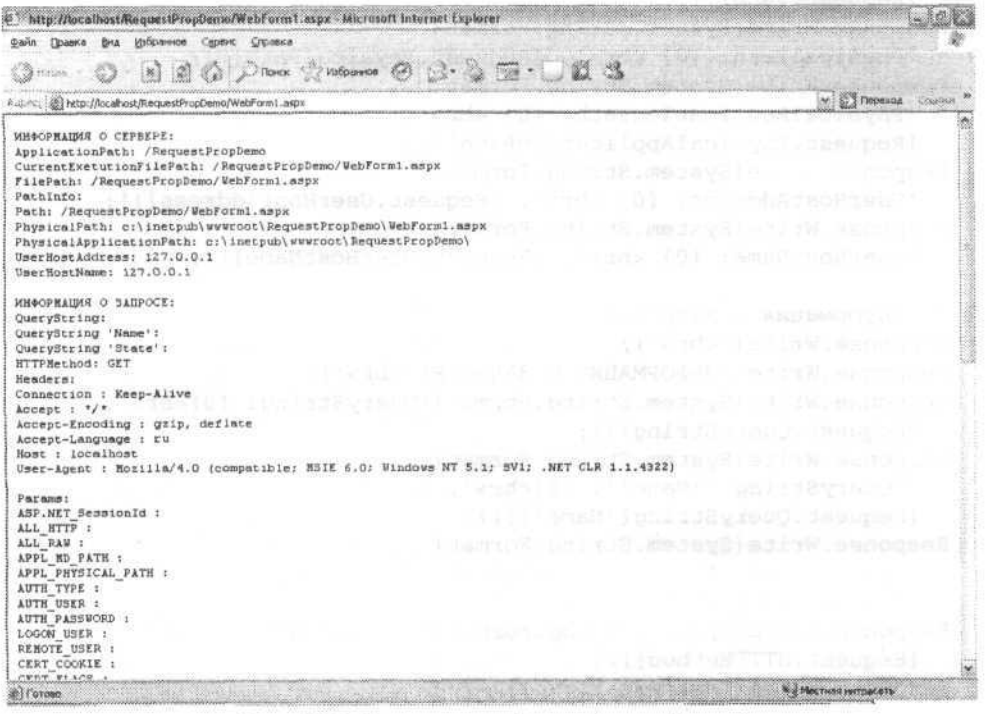


Рис. 16.2. Некоторые свойства объекта Request

Листинг 16.1. Отображение некоторых свойств объекта Request

```
procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
var
  NameValCol: NameValueCollection;
  StrArray: array of String;
  i: integer;
begin
  Response.Write('<code>');
  // Клиент может запросить информацию о сервере:
  Response.Write('ИНФОРМАЦИЯ О СЕРВЕРЕ: <br>');
  Response.Write(System.String.Format(
    'ApplicationPath: {0} <br>', [Request.ApplicationPath]));
  Response.Write(System.String.Format(
    'CurrentExetutionFilePath: {0} <br>',
    [Request.CurrentExecutionFilePath]));
  Response.Write(System.String.Format('FilePath: {0} <br>',
    [Request.FilePath]));
  Response.Write(System.String.Format('PathInfo: {0} <br>',
    [Request.PathInfo]));
  Response.Write(System.String.Format('Path: {0} <br>',
    [Request.Path]));
  Response.Write(System.String.Format(
    'PhysicalPath: {0} <br>', [Request.PhysicalPath]));
  Response.Write(System.String.Format(
    'PhysicalApplicationPath: {0} <br>',
    [Request.PhysicalApplicationPath]));
  Response.Write(System.String.Format(
    'UserHostAddress: {0} <br>', [Request.UserHostAddress]));
  Response.Write(System.String.Format(
    'UserHostName: {0} <br>', [Request.UserHostName]));

  // Информация о запросе:
  Response.Write('<br>');
  Response.Write('ИНФОРМАЦИЯ О ЗАПРОСЕ: <br>');
  Response.Write(System.String.Format('QueryString: {0}<br>',
    [Request.QueryString]));
  Response.Write(System.String.Format(
     'QueryString ''Name'': {0}<br>',
    [Request.QueryString['Name']]));
  Response.Write(System.String.Format(
     'QueryString ''State'': {0}<br>',
    [Request.QueryString['State']]));
  Response.Write(System.String.Format('HTTPMethod: {0} <br>',
    [Request.HTTPMethod]));
  Response.Write('Headers:<br>');

  NameValCol := Request.Headers;
  StrArray := NameValCol.AllKeys;
```

```
  for i := Low(StrArray) to High(StrArray) do
    Response.Write(System.String.Format('    {0} : {1} <br>',
      [StrArray[i], Request.Headers.Item[StrArray[i]]]));

  Response.Write('<br>');
  Response.Write('Params:<br>');
  NameValCol := Request.Params;
  StrArray := NameValCol.AllKeys;
  for i := Low(StrArray) to High(StrArray) do
    Response.Write(System.String.Format('    {0} : {1} <br>',
      [StrArray[i], Request.Headers.Item[StrArray[i]]]));


  // Информация о браузере клиента:
  Response.Write('<br>');
  Response.Write('ИНФОРМАЦИЯ О БРАУЗЕРЕ:<br>');
  Response.Write(System.String.Format('Type: {0}<br>',
    [Request.Browser.&Type.ToString]));
  Response.Write(System.String.Format('Version: {0}<br>',
    [Request.Browser.Version]));
  Response.Write(System.String.Format('ClrVersion: {0}<br>',
    [Request.Browser.ClrVersion.ToString]));
  Response.Write(System.String.Format('Platform: {0}<br>',
    [Request.Browser.Platform]));
  Response.Write(System.String.Format('Frames: {0}<br>',
    [Request.Browser.Frames.ToString]));
  Response.Write('</code>');
end;
```

Класс HttpResponse

Свойство `Response` объекта `Page` относится к классу `HttpResponse`. Этот класс предназначен для описания параметров ответа на запрос пользователя. Его свойства представлены в табл. 16.3.

Таблица 16.3. Свойства класса HttpResponse

Свойство	Описание
`property Buffer: Boolean;`	Содержит `True`, если ответ кэшируется
`property Cache: HttpCachePolisy;`	Содержит объект `HttpCachePolisy` с информацией о кэшированной части ответа
`property CacheControl: String;`	Определяет способ использования кэша и может принимать значение `Public` или `Private`
`property Charset: String;`	Содержит название набора символов ответа
`property ContentEncodind: Encoding;`	Содержит объект `Encoding` с информацией о наборе символов ответа

продолжение

Таблица 16.3 (продолжение)

Свойство	Описание
`property ContentType: String;`	Содержит тип ответа. По умолчанию — `"Text/HTML"`
`property Cookies: HttpCookieCollection;`	Содержит связанную с ответом коллекцию объектов `HttpCookie`
`property Expires: Integer;`	Содержит время хранения ответа в буфере в минутах
`property ExpiresAbsolute: DateTime;`	Содержит время и дату удаления из кэша кэшированной информации ответа
`property Filter: Stream;`	Содержит поток для фильтрации ответа
`property IsClientConnected: Boolean;`	Содержит `True`, если клиент соединен с сервером
`property Output: TextWriter;`	Содержит объект `TextWriter` с ответом клиенту
`property OutputStream: Stream;`	Содержит ссылку на поток с ответом клиенту
`property RedirectLocation: String;`	Содержит URL-адрес, передаваемый клиенту в строке заголовка
`property Status: String;`	Содержит строку статуса, возвращаемую в ответ на запрос клиента. По умолчанию — `"200OK"`
`property StatusCode: Integer;`	Содержит код статуса, возвращаемый в ответ на запрос клиента. По умолчанию — `200 ("OK")`
`property StatusDescription: String;`	Содержит строку статуса, возвращаемую в ответ на запрос клиента. По умолчанию — `"OK"`
`property SuppressContent: Boolean;`	Содержит `True`, если ответ будет посылаться клиенту

Свойство `Response` объекта `Page` используется в основном в трех случаях: для программной передачи HTML-информации пользователю, для фильтрации ответа и для его переадресации.

Для передачи HTML-информации используется метод `Write` класса `HttpResponse`. Рассмотренный ранее пример отображения некоторых свойств объекта `Request` наглядно демонстрирует эту возможность (см. листинг 16.1).

Фильтрация данных в общем случае затруднена, так как символьный тип `Char` хранит символ Unicode и не может преобразовываться в целочисленные типы[1]. Фильтрация же обычно сводится к некоторым манипуляциям над символами, требующими преобразования символьного типа в число и наоборот. Для строк, содержащих только символы с кодами от 0 до 127 включительно, символ Unicode

[1] В языке Visual Basic .NET определена функция `AscW`, которая преобразует символ Unicode в значение типа `Word`, но в общей системе типов (и Delphi) такой функции нет.

совпадает с однобайтным символом ASCII, который может преобразовываться в значение типа Byte функцией Delphi ord.

В листинге 16.2 (проект Ch16\ResponseFilter\ResponseFilter.dpr) все символы текста ответа преобразуются в прописные.

Листинг 16.2. Преобразование в верхний регистр всех символов текста ответа

```
unit WebForm1;

interface

uses
  System.Collections, System.ComponentModel,
  System.Data, System.Drawing, System.Web,
  System.Web.SessionState, System.Web.UI,
  System.Web.UI.WebControls, System.Web.UI.HtmlControls,
  System.IO; // В этом пространстве имен объявлен класс Stream

type
  TWebForm1 = class(System.Web.UI.Page)
  strict private
    procedure Page_Load(sender: System.Object;
                        e: System.EventArgs);
  strict protected
    procedure OnInit(e: EventArgs); override;
  private
    { Private Declarations }
  public
    { Public Declarations }
  end;

  TFilter = class(MemoryStream) // Класс для фильтрации
  private
    MemStream: Stream;
  public
    constructor Create(aStream: Stream); override;
    procedure Write(Bfr: array of Byte;
                Offset, Count: Integer); override;
  end;

implementation

procedure TWebForm1.InitializeComponent;
begin
  Include(Self.Load, Self.Page_Load);
end;

procedure TWebForm1.OnInit(e: EventArgs);
begin
  InitializeComponent;
```

продолжение

Листинг 16.2 (продолжение)

```
  inherited OnInit(e);
end;

procedure TWebForm1.Page_Load(sender: System.Object;
                              e: System.EventArgs);
begin
  // Заменяем свойство Filter новым потоком:
  Response.Filter := TFilter.Create(Response.Filter);
  // Выводим строку ответа (только латиница!):
  Response.Write('Delphi 2005 for .NET Framework<br>');
end;

{ TFilter }

procedure TFilter.Write(Bfr: array of Byte;
                        Offset, Count: Integer);
// Используемое в процедуре преобразование регистров
// возможно только для латиницы!
var
  Data: array of Byte;
  k: Integer;
  Delta: Integer;
begin
  inherited;
  Borland.Delphi.System.SetLength(Data, Count);
  Buffer.BlockCopy(Bfr, Offset, Data, 0, Count);
  Delta := ord('A') - ord('a');
  for k := 0 to Count - 1 do
    if (Data[k] >= ord('a')) and (Data[k] <= ord('z')) then
      Data[k] := Data[k] + Delta;;
  MemStream.Write(Data, 0, Count);
end;

constructor TFilter.Create(aStream: Stream);
begin
  inherited Create;
  MemStream := aStream;
end;

end.
```

Для переадресации запроса используется метод `Redirect`. Например:

```
Response.Redirect('http://DelphiKingdom.com');
```

Состояние вида

Состояние вида (*view state*) — это механизм ASP.NET, с помощью которого отслеживаются текущие значения свойств серверных элементов управления. Благодаря

этому механизму можно восстановить вид страницы, которая уже посещалась ранее, причем не обязательно в том же сеансе связи с сервером. Обычно для сохранения информации состояния вида используется скрытое поле _ViewState, которое автоматически создается сервером и размещается в странице ответа на запрос пользователя. Но состояние вида можно также сохранять в наборе состояний страницы или отдельного элемента, в объектах Session и Application, на специальном сервере состояний и даже на SQL-сервере. В этом разделе обсуждаются различные способы хранения состояния вида.

Поле _ViewState

Стандартная веб-страница, создаваемая в Delphi 2005, включает директиву Page следующего вида:

```
<%@ Page language="c#" Debug="true" Codebehind="WebForm1.pas"
  AutoEventWireup="false" Inherits="WebForm1.TWebForm1" %>
```

В этой директиве отсутствует атрибут EnableViewState, который, таким образом, получает умалчиваемое значение True. В результате стандартная веб-страница автоматически настраивается на создание и использование поля _ViewState.

Поле _ViewState работает следующим образом. Данные, введенные пользователем в веб-форму, передаются на сервер как часть команды POST или GET протокола HTTP. Сервер упаковывает эту информацию в скрытое поле _ViewState и возвращает браузеру как часть ответа. Например:

```
<input type="hidden" name="__VIEWSTATE"
  value="dDw2MDM5MTc5OTA7Oz76vVjs6Z1VGNd8+CxSPPVuouWK5A==" />
```

ПРИМЕЧАНИЕ

Чтобы увидеть скрытое поле, нужно в браузере выбрать команду Вид ▸ Просмотр HTML-кода.

Клиентский браузер игнорирует эту информацию, но отсылает ее серверу при последующих запросах.

Поле _ViewState может хранить довольно большой объем информации, в особенности для сложных страниц, насыщенных серверными элементами управления. Чтобы уменьшить объем информации, которой обмениваются браузер клиента и сервер, это поле можно отключить. Для этого в директиву Page следует добавить такой атрибут:

```
EnableViewState="False"
```

Эта директива отменяет необходимость хранить состояние вида для страницы в целом. Но можно отключать этот механизм и для отдельного элемента управления. Для этого в окне инспектора объектов нужно свойству EnableViewState элемента присвоить значение False.

Управление состоянием на уровне сеанса

Управление состоянием на уровне сеанса начинается при входе пользователя на сайт и завершается при его выходе.

Когда пользователь посещает сайт, ASP.NET создает для него специальный объект `Session` (*сеанс*) класса `HttpSessionState`. Кроме того, пользователю присваивается уникальный идентификатор, который передается его браузеру и возвращается серверу при каждом запросе в течение сеанса. По умолчанию для этого используется механизм cookie.

Сеанс остается в памяти до тех пор, пока пользователь не покинет сайт или пока не закончится отведенное для него время (по умолчанию — 20 минут).

Информацию сеанса поддерживает так называемый *провайдер состояний*, который может работать в одном из следующих режимов:

- `InProc` — данные сеанса хранятся в том же самом домене, что и приложение ASP.NET, то есть внутри контекста процесса aspnet_wp.exe (этот режим принят по умолчанию);
- `StateServer` — данные сеанса хранятся в контексте службы aspnet_state.exe Windows, которая может выполняться на той же машине, что и сервер, или в ее сетевом окружении;
- `SQLServer` — данные сохраняются в таблице базы данных;
- `Off` — данные сеанса не хранятся.

Для добавления данных в объект `Session` используется его метод `Add`:

```
Session.Add('UserName', TextBox1.Text);
```

Чтобы получить данные, нужно обратиться к ним по имени, например:

```
Response.Write('Welcom ' + Session['UserName'].ToString);
```

Или:

```
TextBox1.Text := Session['UserName'].ToString;
```

В объекте `Session` можно хранить данные любого типа:

```
Session.Add('MyDataset', DataSet1);
```

Для удаления данных используется метод `Remove`:

```
Session.Remove('MyDataSet');
```

Изменение умалчиваемого времени сеанса

По умолчанию на сеанс отводится 20 минут, после чего объект `Session` (и вся хранимая в нем информация) уничтожается. С помощью дескриптора **<sessionState>** конфигурационного файла Web.config можно изменить это время:

```
<configuration>
  <system.web>
```

```
    <sessionState timeout="60"/>
  </system.web>
</configuration>
```

Сеансы без cookie

Как правило, идентификатор сеанса сохраняется в файле cookie браузера клиента. Однако пользователь может запретить использование cookie (для браузера IE с помощью команды Сервис ▸ Свойства обозревателя нужно открыть окно свойств и на вкладке Конфиденциальность передвинуть ползунок доступности cookie в крайнее верхнее положение). В этом случае идентификатор сеанса можно передавать в составе URL. Для этого в файл Web.config вставляются такие строки:

```
<configuration>
  <system.web>
    <sessionState cookieless="true"/>
  </system.web>
</configuration>
```

Хранение данных сеанса на сервере MS SQL Server

В ASP.NET имеется возможность хранить данные сеанса на сервере MS SQL Server. Этот вариант — наименее производительный, но и наиболее отказоустойчивый.

Для организации хранения информации о состоянии на сервере MS SQL Server нужно, во-первых, создать на сервере базу данных с нужными таблицами, во-вторых, настроить конфигурационный файл Web.config.

Для создания БД следует использовать один из следующих файлов сценариев:

- InstallSqlState.sql — создает базы данных `ASPState` и `TempDB`, информация о состоянии сохраняется в таблицах БД `TempDB` и теряется при отключении сервера;
- InstallPersistSqlState.sql — создает и использует БД `ASPState`, информация в этой БД не теряется при отключении сервера.

Оба файла располагаются в папке Windows\Microsoft.NET\Framework\v1.1.4322. В этой же папке расположены и файлы UninstallSqlState.sql и UninstallPersistSqlState.sql, служащие для отмены регистрации баз данных.

Для использования файлов сценариев запустите утилиту SQL Server Enterprise Manager и выберите команду Tools ▸ SQL Query Analyzer. В новом окне загрузите нужный файл и нажмите клавиши Ctrl+F5 или выберите команду Query ▸ Parse (рис. 16.3).

После создания БД нужно в файл Web.config поместить такие строки:

```
<configuration>
  <system.web>
    <sessionState mode="SQLServer"
        sqlConnectionString="data source=localhost;
        user id=sa; pwd=password"/>
  </system.web>
</configuration>
```

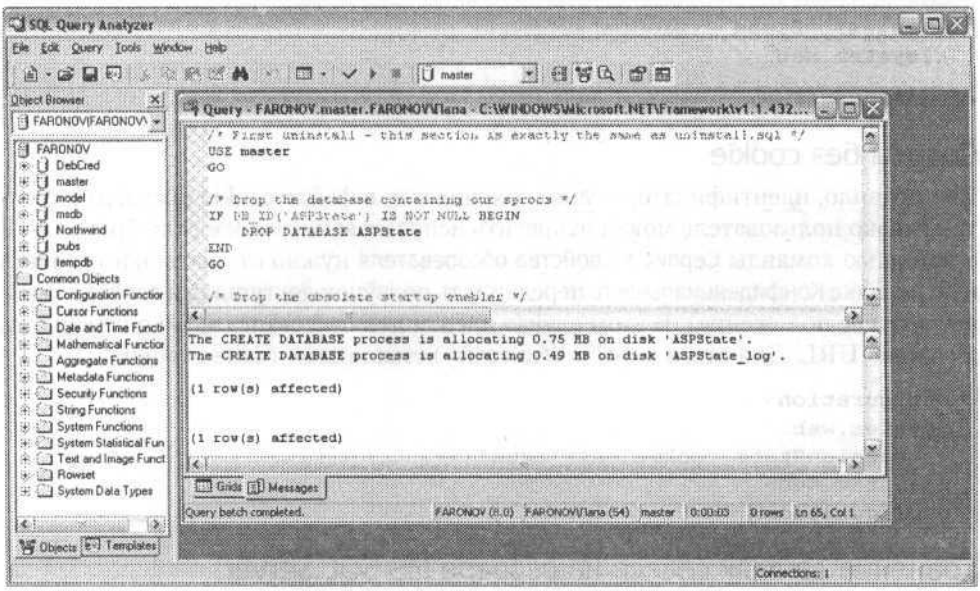


Рис. 16.3. Выполнение файла сценария

Управление состоянием на уровне приложения

Управление состоянием на уровне приложения во всем подобно управлению на уровне сеанса, за исключением того, что данные о состоянии оказываются доступны всем клиентам. Связано это с тем, что хранение состояния на уровне приложения реализуется экземпляром класса `HttpApplicationState`, который создается при первом обращении к приложению (сеанс создается при обращении к приложению каждого нового клиента).

Приложения

Сервер InterBase

Назначение и возможности

SQL-сервер InterBase предназначен для хранения и обработки больших объемов информации в условиях одновременной работы с БД множества клиентских приложений. Масштаб информационной системы при этом произволен — от уровня рабочей группы (под управлением Novell Netware, Linux или 32-разрядной версии Windows на базе IBM-совместимых персональных компьютеров) до уровня большого предприятия (на базе компьютеров IBM, Hewlett-Packard, SUN).

В этом приложении рассматриваются те технологии InterBase, использование которых обеспечивает максимальную вычислительную разгрузку клиентского приложения и гарантирует высокую степень безопасности и целостность информации. Перечислим эти технологии:

- *Первичные и внешние ключи.* Отношения подчиненности между таблицами БД определяются с помощью первичных ключей у родительских и внешних ключей — у дочерних таблиц.
- *Ограничения на значения отдельных столбцов.* Условия ограничений могут быть разнообразными — от требования, чтобы вводимые значения не выходили из определенного диапазона или соответствовали некоторой маске, до требуемого отношения с одной или несколькими записями из другой таблицы (или многих таблиц) БД.
- *Триггеры.* Триггеры — это подпрограммы, автоматически выполняемые сервером до или/и после события изменения записи в таблице БД.
- *Генераторы.* Генераторы служат для создания и использования уникальных значений нужных полей (только для InterBase).
- *Хранимые процедуры.* Для ускорения работы клиентских приложений с удаленной БД могут быть определены хранимые процедуры, которые представ-

ляют собой подпрограммы, принимающие и возвращающие параметры и способные выполнять запросы к БД, условные ветвления и циклическую обработку. Хранимые процедуры пишутся на специальном алгоритмическом языке. В них программируются часто повторяющиеся последовательности запросов к БД. Текст процедур хранится на сервере в откомпилированном виде.

- *BLOB-поля* и *BLOB-фильтры*. В составе записи БД могут определяться BLOB-поля (Binary Large OBject — большой двоичный объект), предназначенные для хранения больших объемов данных в виде последовательности байтов. Таким образом могут храниться текстовые и графические документы, файлы мультимедиа, звуковые файлы и т. д. Интерпретация BLOB-поля выполняется в приложении, однако разработчик может определить так называемые BLOB-фильтры, предназначенные для автоматического преобразования содержимого BLOB-поля к другому виду.
- *Пользовательские функции*. InterBase дает возможность создавать пользовательские функции (User Defined Function, UDF), в которых может реализовываться функциональность, отсутствующая в стандартных встроенных функциях InterBase (стандартные функции применяются для вычисления максимума, минимума, среднего значения, преобразования типов и смены регистра букв). Например, в UDF можно реализовать извлечение из значения даты номера дня, года, определение длины символьного значения, усечение пробелов, разные математические алгоритмы и т. п. Функция пишется на любом алгоритмическом языке, позволяющем разрабатывать библиотеки динамической компоновки (DLL), например на языке Delphi.
- *Уведомления*. InterBase может посылать уведомления клиентским приложениям о наступлении какого-либо события. Одновременно работающие приложения могут обмениваться сообщениями через сервер БД, вызывая хранимые процедуры, инициирующие нужные события.
- *Виртуальные таблицы*, или *представления*. Чтобы ускорить выполнение запросов и освободить клиентское приложение от необходимости такие запросы выдавать, в БД можно определить виртуальные таблицы, или представления (views), в которых объединяются записи из одной или более таблиц, соответствующих некоторому условию. Работа с представлением из клиентского приложения ничем не отличается от работы с обычной таблицей. Поддерживает представления сервер, реагируя на изменение данных в БД. Представления могут быть изменяемыми или не изменяемыми (не допускающими внесения в них изменений).

Некоторые технические характеристики

Сервер InterBase был разработан в начале 80-х годов в американской корпорации DEC. В дальнейшем разработка сервера велась независимой компанией InterBase Software и впоследствии слившейся с ней компанией Ashton-Tate. Корпорация Borland приобрела права на InterBase у компании Ashton-Tate после слияния с нею.

Сервер InterBase активно используется в государственном и военном секторах США, что, видимо, и стало преградой для его продвижения в Россию. Интерес к этому серверу возрос только в последнее время в связи с включением его локальной (а начиная с Delphi 3 — и пользовательской) версии в состав Delphi Client/Server Suite и Delphi Enterprise. Внимание разработчиков БД сервер InterBase привлек, во-первых, потому, что это «родной» продукт Borland (а средства разработки приложений этой компании давно зарекомендовали себя с положительной стороны), во-вторых, потому, что сервер InterBase весьма прост в установке, настройке и администрировании по сравнению с другими SQL-серверами, и, в-третьих, потому, что он обладает прекрасными функциональными возможностями.

В табл. А.1 приводятся некоторые технические характеристики сервера.

Таблица А.1. Технические характеристики сервера InterBase

Характеристика	Значение
Максимальный размер одной БД	Рекомендуется не выше 10 Гбайт, однако известны случаи объема одной БД в 10–20 Гбайт
Максимальное количество таблиц в одной БД	65 536
Максимальное количество полей (столбцов) в одной таблице	1000
Максимальное количество записей (строк) в одной таблице	Не ограничено
Максимальная длина записи	64 Кбайт (кроме BLOB-полей)
Максимальная длина поля	32 Кбайт (кроме BLOB-полей)
Максимальная длина BLOB-поля	Не ограничена
Максимальное количество индексов в БД	65 536
Максимальное количество полей в индексе	16
Максимальный уровень вложенности SQL-запроса	16
Максимальный размер хранимой процедуры или триггера	48 Кбайт
Максимальное количество пользовательских функций (UDF) в базе данных	Длина имени UDF — не более 31 символа, каждая пользовательская функция должна иметь уникальное имя, поэтому максимальное количество UDF ограничивается только требованием уникальности имен

Физическая организация базы данных InterBase

База данных InterBase состоит из последовательно, начиная с 0, пронумерованных страниц. Нулевая страница является служебной и содержит информацию,

необходимую для соединения с БД. Размер страницы — 1 (по умолчанию), 2, 4 или 8 Кбайт. Размер страницы задается при создании БД, но может быть изменен при сохранении и восстановлении базы. Одна страница читается сервером за один логический доступ к БД.

Объем буфера ввода-вывода для операций чтения-записи определяется в количестве страниц (по умолчанию — 75). Если БД будет чаще читаться, объем буфера следует увеличить. Если в нее будет чаще осуществляться запись, размер буфера можно уменьшить.

В InterBase для записей поддерживается режим множества версий. При изменении записи какой-либо транзакцией создается новая версия записи, куда помимо данных записываются номер транзакции и указатель на предыдущую версию записи. Старая версия помечается как измененная, а ее указатель на следующую версию записи содержит ссылку на вновь созданную версию. Каждая стартующая транзакция работает с последней версией записи, изменения для которой подтверждены. Таким образом, параллельно работающие с БД транзакции всегда используют разные версии записей, что позволяет снимать блокировки для клиентских приложений, одновременно работающих с одними и теми же данными в БД. При удалении записи она физически не удаляется с диска, а помечается как удаленная до тех пор, пока не завершатся все активные транзакции, использующие эту запись.

InterBase размещает на одной своей странице все версии одной записи таблицы БД. После удаления записей на странице образуются «дырки». При добавлении новой записи анализируется размер максимальной «дырки» и, если он меньше длины добавляемой записи, происходит сжатие страницы, в процессе которого «дырки» объединяются. Если освободившегося пространства не хватает для размещения новой записи, та записывается с новой страницы. Загрузка страницы считается нормальной в случае, если «дырки» занимают не более 20 % объема страницы.

Порядок выделения страниц для хранения данных никак не оптимизирован. На отдельной служебной странице БД хранятся номера всех свободных страниц. Для сохранения записей одной таблицы БД не предпринимается никаких действий по выделению смежных страниц, а выделяется первая страница в списке свободных. Если свободной страницы нет, добавляется новая в конец БД. Только в этом случае размер БД возрастает.

Структура записей, поддерживающая режим множества версий, и неоптимальное выделение страниц ведут к высокой степени фрагментации БД и, как следствие, к замедлению работы с ней. Поэтому необходимо периодически производить дефрагментацию БД. Дефрагментированная БД характеризуется расположением записей таблиц БД на смежных страницах и отсутствием «мусора». Под «мусором» понимаются версии записей, с которыми не работает никакая активная транзакция.

Для удаления «мусора» БД сохраняется на дисковом носителе и затем восстанавливается из сделанной резервной копии с помощью утилиты IBConsole (для предыдущих версий InterBase — с помощью утилиты InterBase Server Manager). Этот процесс гарантирует удаление всего «мусора», так как в момент сохранения

и восстановления БД обеспечивается отсутствие активных подключений к БД со стороны других пользователей, а значит, и отсутствие активных транзакций. Кроме того, в InterBase предусмотрено автоматическое удаление «мусора» на фоне активных транзакций. Интервал, через который происходит автоматическое удаление «мусора», по умолчанию составляет 20 000 транзакций. Это значение может быть изменено с помощью утилиты IBConsole. При автоматической очистке удаляются только те версии записей, для которых нет активных транзакций. В результате могут быть удалены не все старые версии.

ВНИМАНИЕ

Если довольно долго не осуществляется принудительная очистка страниц БД InterBase, это может привести к существенному (в десятки раз) снижению производительности сервера. При активной работе с БД рекомендуется производить принудительную очистку БД InterBase не реже 2–3 раз в неделю (или даже ежедневно).

Типы данных InterBase

Как уже говорилось, сервер InterBase выбран в качестве базового сервера прежде всего из-за того, что он входит в комплект поставки Delphi. Описанные в этом приложении типы данных относятся именно к InterBase, хотя многие другие SQL-серверы имеют схожие типы данных. В отличие от других серверов, сервер InterBase:

- не имеет автоинкрементного типа данных и компенсирует его отсутствие особым механизмом генераторов;
- не имеет логического типа данных и вместо него предлагает использовать символьный тип `Char(1)` или `VarChar(1)`;
- не имеет денежного типа данных;
- использует столбцы-массивы.

Несмотря на это InterBase как, возможно, ни один другой SQL-сервер следует промышленному стандарту SQL92. Как бы там ни было, большинство его типов данных совпадают с аналогичными типами Delphi.

Обзор типов данных InterBase

Типы столбцов, которые могут использоваться в таблицах сервера InterBase, перечислены в табл. А.2.

Таблица А.2. Типы данных сервера InterBase

Тип столбца	Размер, байт	Описание
`SMALLINT`	2	Целочисленные значения от –32 768 до +32 767
`INTEGER`	4	Целочисленные значения от –2 147 483 648 до +2 147 483 647
`FLOAT`	4	Вещественные числа до 7 значащих цифр в диапазоне от $3{,}4 \cdot 10^{-38}$ до $3{,}4 \cdot 10^{+38}$

Тип столбца	Размер, байт	Описание
DOUBLE PRECISION	8	Вещественные числа до 15 значащих цифр в диапазоне от $1,7 \cdot 10^{-308}$ до $1,7 \cdot 10^{+308}$
NUMERIC или DECIMAL	Переменный	Вещественные числа с фиксированной запятой. При определении этого типа дополнительно указывается общее количество значащих цифр числа и количество цифр в дробной части
CHAR(n) или CHARACTER	0–32 767	Текстовый столбец длиной до *n* символов
VARCHAR(n) или CHAR[ACTER] VARYING	0–32 767	Текстовый столбец переменной длины, содержащий до *n* символов
DATE	8	Дата в пределах от 01.01.0100 до 11.12.5941. Также может хранить сведения о времени
BLOB	Переменный	Любой тип двоичных данных

Столбцы могут определяться в SQL-операторах CREATE TABLE — создать таблицу БД, CREATE DOMAIN — создать домен и ALTER TABLE — изменить структуру таблицы.

Пример описания столбцов разного типа при определении таблицы:

```
CREATE TABLE MixTable(
/* столбец типа INTEGER; вместо INT можно использовать INTEGER */
  Col1 INT,
/* столбец-массив из трех значений SMALLINT */
  Col2 SMALLINT[3],
/* символьный столбец длиной до 70 символов с возможностью
использовать кириллицу */
  Col3 VARCHAR(70) CHARACTER SET Win1251,
/* символьный столбец из одного символа со значением "Y",
заданным по умолчанию; такие столбцы заменяют логический
тип, если значениями являются Y, N, T, F */
  Col4 CHAR(1) DEFAULT 'Y',
/* вещественный столбец, который не может быть пустым */
  Col5 FLOAT NOT NULL,
/* вещественный столбец; его значения содержат до 12 значащих
цифр, из которых 2 представляют дробную часть: 1234567890,12 */
  Col6 NUMERIC(12,2))
```

При определении столбца с помощью спецификатора DEFAULT можно указать значение, заданное по умолчанию. В этом случае сервер автоматически поместит в столбец указанное значение, если при вводе новой записи этот столбец остался пустым:

```
CREATE TABLE MoveBook(..., IsDelNak1 CHAR(1) DEFAULT 'F')
```

При определении значения по умолчанию можно использовать три стандартных значения:

- USER — помещает в столбец новой записи регистрационное имя пользователя БД;
- "NOW" — помещает в столбец текущие дату и время;
- "TODAY" — помещает в столбец текущую дату.

Обратите внимание: значение USER указывается без кавычек, в то время как два других — в кавычках.

Как видно из табл. А.2, в InterBase не определен логический тип. Вместо него можно использовать тип CHAR(1) с указанием значений типа T, F, Y, N, Д, Н, +, - и т. п. При переносе таблиц Paradox утилита Datapump (поставлялась с предыдущими версиями Delphi) заменяет значения True и False логического поля, соответственно, символами T и F столбца CHAR(1). Однако компонент TDBCheckBox (флажок), который обычно визуализирует логическое значение, при связи его со столбцом CHAR(1) показывает значение True независимо от содержимого столбца (даже если он содержит NULL). Происходит это по той причине, что его свойства ValueChecked и ValueUnchecked содержат значения True и False, заданные по умолчанию. Чтобы компонент правильно отображал логические значения, в эти свойства следует поместить нужные значения. Например:

```
DBCheckBox1.ValueChecked := 'T;t;Д;д'
DBCheckBox1.ValueUnchecked := 'F;f;H;н'
```

Столбцы-массивы

Столбец любого типа, кроме типа BLOB, может быть объявлен как столбец-массив. Такие столбцы вместо единичного значения содержат массив значений одинакового типа, доступ к каждому из которых возможен с помощью индекса. Для создания столбца-массива в конце его определения указывается в квадратных скобках целое число, определяющее количество элементов массива. Например:

```
CREATE TABLE MYTABLE (A INTEGER [3])
```

В этом примере столбец A будет содержать по 3 целочисленных значения в каждой записи таблицы MYTABLE. Однако следует учесть, что стандарт SQL92, которому в целом следуют InterBase и BDE, не поддерживает столбцы-массивы, поэтому, например, нельзя выполнить следующие операторы:

```
INSERT INTO MyTable (A[1]) Values(1)  /* Недопустимый символ [ */
INSERT INTO MyTable (A) Values(1,2,3) /* Количество значений не
равно количеству столбцов */
```

Доступ к отдельным значениям столбца-массива возможен только с помощью низкоуровневых API-функций InterBase, что затрудняет практическое использование массивов (подробное описание работы со столбцами-массивами см. в документации к InterBase).

Типы DECIMAL и NUMERIC

Типы DECIMAL и NUMERIC задают вещественные значения и определяют в них фиксированное количество знаков после запятой. Формат определения этих типов:

имя_столбца [=] {DECIMAL | NUMERIC} [(*точность* [, *масштаб*])]

Здесь *точность* — общее количество знаков в хранимом числе (максимум 15), *масштаб* — количество знаков в дробной части (может быть равен нулю). Во всех случаях масштаб должен быть меньше точности. Например:

```
CREATE TABLE MYTABLE (DECIMAL_COLUMN DECIMAL(9,2))
```

Замечу, что специальных типов DECIMAL и NUMERIC не существует, а вместо них используется тип INTEGER или DOUBLE PRECISION: если точность (количество знаков в числе) меньше 10, то реальный тип столбца — INTEGER, если больше или равно 10, реальный тип столбца — DOUBLE PRECISION. Поскольку в столбцах типа INTEGER хранить дробные значения нельзя, то независимо от того, указано количество знаков после десятичной точки или нет, объявление DECIMAL и NUMERIC с общим числом разрядов меньшим 10 приведет к тому, что фактически в столбце будут храниться *только целочисленные значения*. Например, если в столбец DECIMAL_COLUMN таблицы MYTABLE (см. выше) поместить значение 123.456, фактически в таблице будет храниться число 123, а все цифры дробной части будут безвозвратно потеряны.

Тип DATE

Столбцы типа DATE позволяют хранить значения даты в пределах от 01.01.0100 до 11.12.5941, а также значения времени (тип DATE InterBase полностью совместим с типом TDateTime Delphi).

Если ввод данных в столбец типа DATE производится из утилиты IBConsole, значения даты должны указываться в формате InterBase. Согласно этому формату, значения даты состоят из номера дня (01-31), месяца (JAN-DEC) и года. Эти значения отделяются друг от друга разделителями. Стандартным разделителем является символ дефиса (-), но принимаются и пробел, правый слеш (/) и точка (.).

Значения дат в InterBase должны лежать в диапазоне от 1-JAN-100 до 11-DEC-5941. Интерпретация формата представления значений типа DATE зависит от настройки — программы или операционной системы компьютера. Полезно всякий раз при старте приложения программно переустанавливать формат даты и времени к привычному нам российскому формату:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DateSeparator := '.';
  ShortDateFormat := 'dd.mm.yyyy';
  ShortTimeFormat := 'hh:mm:ss';
end;
```

Таким образом можно игнорировать неопределенность текущей настройки операционной системы на конкретном компьютере.

В InterBase значения типа DATE совместимы со строковыми типами. Поэтому, если в SQL-операторах InterBase требуется интерпретировать значения типа DATE как строку, нет необходимости в приведении типов. Например, можно так записать в символьный столбец S значение типа DATE (символы || означают операцию конкатенации, или сцепления, строк, функция NOW возвращает текущие дату и время):

```
UPDATE SOMETABLE
  SET S = "Дата отгрузки " || NOW;
```

Кроме того, присваивание значения полям типа DATE также производится в символьном формате:

```
UPDATE Nakls
SET NDate = "01.01.2000"
WHERE NaklID<100
```

Типы CHAR и VARCHAR

Столбцы типа CHAR и VARCHAR позволяют хранить текстовые значения длиной до *n* символов. В любом случае *n* не может превышать 32 Кбайт.

Как сказано в документации, тип CHAR(n) определяет текстовые столбцы фиксированной длины. Определение «фиксированная длина» означает, что даже если в столбец записано меньше *n* символов, незаполненное пространство заполняется пробелами. При хранении завершающие пробелы удаляются, что позволяет хранить действительно значащее содержимое столбцов типа CHAR(n) и сближает их со строками переменной длины. При чтении значения столбца завершающие пробелы вставляются в столбец снова.

ПРИМЕЧАНИЕ

Опытным путем нетрудно выяснить, что Delphi при работе с InterBase интерпретирует столбцы типа CHAR и VARCHAR как TStringField. При этом, во-первых, столбцы типа CHAR всегда читаются без завершающих пробелов, во-вторых, при занесении завершающих пробелов в столбец (а часто это бывает необходимо) типа CHAR они всегда удаляются, в то время как в столбце типа VARCHAR, наоборот, они всегда хранятся.

Тип VARCHAR(n) определяет текстовые столбцы переменной длины. Определение «переменная длина» означает, что в записи хранится ровно столько символов, сколько их имеется в значении столбца. Например, если для VARCHAR(100) значения столбцов в одной записи — «Петров», а в другой — «Барабанов», в первом случае хранится 6 символов, а во втором — 9. Столбцы VARCHAR(n) позволяют экономить дисковое пространство, давая возможность серверу располагать больше записей на странице БД, однако они читаются медленнее, чем CHAR(n).

Попытка записать в столбец CHAR (VARCHAR) более чем *n* символов приведет к удалению лишних символов.

Формат определения текстовых столбцов:

```
имя_столбца [=] {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
                [(n)] [CHARACTER SET набор_символов]
```

Здесь n определяет длину текстового столбца. Если аргумент n опущен, по умолчанию подразумевается 1.

Спецификатор CHARACTER SET позволяет установить кодировку путем указания имени применяемого набора символов. Если спецификатор CHARACTER SET опущен, используется кодировка, принятая по умолчанию для БД (устанавливается в предложение DEFAULT CHARACTER SET оператора CREATE DATABASE). Если определен набор символов, то в столбец невозможно ввести символы, не входящие в данную таблицу кодировки. Для представления столбцов, которые будут содержать русскоязычные тексты, рекомендуется использовать набор символов WIN1251 или CYRL, например:

```
CREATE TABLE MYTABLE (S CHAR(20) CHARACTER SET CYRL)
```

Если при создании БД и описании столбца набор символов не указан, принимается по умолчанию набор NONE, который позволяет вводить в текстовые столбцы символы любых кодировок. Однако для столбцов с этим набором символов могут возникнуть проблемы при записи их значения в столбцы, которым набор символов назначен явно.

Порядок сортировки символов определяет принцип, по которому символьные значения будут сравниваться и сортироваться в операторах SELECT (если в нем присутствуют разделы WHERE, ORDER BY), при обновлении индексов и т. д. При объявлении символьного поля порядок сортировки может явно указываться следующим спецификатором:

```
COLLATE порядок_сортировки
```

Для кириллического набора символов WIN1251 можно использовать порядок сортировки WIN1251 или PXW_CYRL, для набора CYRL — CYRL или DB_RUS. Например:

```
CREATE TABLE MYTABLE (S CHAR(20) CHARACTER
                      SET WIN1251 COLLATE PXW_CYRL);
```

Если задан набор символов, но не указан порядок сортировки, принимается одноименный с набором символов порядок (для CYRL — CYRL, для WIN1251 — WIN1251). Два порядка сортировки для кириллических наборов задают две разные сортировки: первая (заданная по умолчанию) — сортировка в стиле MS-DOS:

```
A B C D ... Z a b c d ... z А Б В Г ... Я а б в г ... я
```

Вторая — сортировка в стиле Windows:

```
a A b B c C d D ... z Z а А б Б в В г Г ... я Я
```

При указании строковых значений в SQL-операторах эти значения должны заключаться либо в апострофы, либо в двойные кавычки. Если значение указано в апострофах, в строке значения можно использовать двойные кавычки как обычный символ и наоборот. Если значение должно одновременно содержать и кавычки, и апострофы, используется операция конкатенации || для сцепления двух строк с разными обрамляющими символами:

```
INSERT INTO MyTable (S)
VALUES("Апостроф ' "||'и кавычки " в одной строке')
```

Помимо операции конкатенации для работы со строковыми значениями может использоваться функция UPPER, преобразующая буквы в прописные. При работе с кириллицей эта функция дает верный результат только для набора WIN1251 и порядка PXW_CYRL. Для остальных сочетаний она не преобразует кириллицу, а для набора CYRL и порядка DB_RUS — искажает русскоязычный текст.

Тип BLOB

В столбцах типа BLOB хранят данные, которые имеют переменную длину и интерпретируются как последовательность байтов (графические образы, тексты большой длины, звуковые файлы, видеоизображения и т. п.). При этом интерпретация содержимого BLOB-столбца может ложиться не только на клиентскую часть, но и на сервер. В последнем случае в серверной БД могут быть созданы так называемые BLOB-фильтры, определяющие способы преобразования BLOB-значений из одного вида в другой. Например, преобразование изображений из одного графического формата в другой.

InterBase хранит значения BLOB-столбцов в самой БД в виде сегментов. Одна операция ввода-вывода при доступе к BLOB-информации оперирует одним сегментом. В таблице БД, если в ней объявлен столбец типа BLOB, хранится указатель на начальный сегмент столбца в области хранения BLOB-информации этой БД. Длина сегмента типа BLOB не влияет на производительность InterBase. Тем не менее ее можно определять явно при создании BLOB-столбца:

```
B BLOB SEGMENT SIZE 1024
```

По умолчанию длина сегмента составляет 80 байт, максимальная длина сегмента — 32 Кбайт (32 768 байт).

Для типа BLOB может быть указан *подтип*, определяющий, какой тип данных будет записан в этот столбец. Подтип есть условное число. Положительные значения зарезервированы InterBase: 0 (по умолчанию) для указания двоичных данных и 1 для указания ASCII-текстов. Отрицательные значения могут использоваться программистом для определения собственных подтипов. Указание подтипа производится с помощью спецификатора SUB_TYPE:

```
имя_столбца BLOB SUB_TYPE -1
```

Интерпретация подтипа не отслеживается сервером и целиком возлагается на программиста.

Если для BLOB-столбца НД создан объект-поле, значение типа BLOB для текущей записи можно сохранить в файле и загрузить из файла:

```
procedure TForm1.FormActivate(Sender:TObject);
begin
  MyTable.Open;
  MyTable.Append;
  MyTableImage.LoadFromFile('ATHENA.BMP');
  MyTableName.Value := 'Голова Афины';
  MyTable.Post;
end;
```

Денежные столбцы

В InterBase нет столбцов денежного типа (Currency). Вместо них обычно используются столбцы FLOAT или DOUBLE PRECISION. При выводе их значений в визуализирующих компонентах они представляют денежные суммы как значения с плавающей запятой, что в большинстве случаев мешает их восприятию. Вернуть этим столбцам нормальный денежный вид можно либо в обработчике события OnGetText объекта-поля, либо с помощью его (объекта-поля) свойства DisplayEdit. В обработчике можно использовать оператор такого вида:

```
procedure TForm1.QUSUMMAGetText(Sender: TField;
                                var Text: String;
                                DisplayText: Boolean);
begin
  Text := FloatToStrF(quSumma.AsFloat,
                      ffCurrency, 10, 2)
end;
```

В свойстве DisplayEdit можно указать следующую маску:

```
###,###.00'р.'
```

Конечный результат будет одинаков, но указать маску, как мне кажется, проще.

Генераторы

Как уже говорилось, в InterBase нет автоинкрементного типа. Поля автоинкрементного типа автоматически заполняются уникальными числовыми значениями при вводе очередной записи. Такие поля обычно используются для создания первичных ключей.

Вместо автоинкрементных полей InterBase предоставляет механизм *генераторов* — особых программ, которые хранят некоторое значение вплоть до момента, когда оно будет использовано, после чего изменяют хранимое значение на заданную величину.

Генератор создается следующим оператором:

```
CREATE GENERATOR Имя_генератора
```

После создания генератора ему необходимо присвоить начальное значение таким оператором:

```
SET GENERATOR Имя_генератора TO Начальное_значение
```

Для получения очередного значения генератора используется такая функция:

```
GEN_ID(Имя_генератора, Шаг)
```

Здесь *Шаг* — целое число, на которое изменяется текущее значение генератора.

Проиллюстрируем использование генератора на примере поля NaklID таблицы NAKLS. Создаем генератор:

```
CREATE GENERATOR NAKLS_NAKLID
SET GENERATOR NAKLS_NAKLID TO 1
```

Однажды созданный генератор хранится в БД наряду с другими ее сущностями — таблицами, индексами, хранимыми процедурами и т. п. Используем генератор при вводе очередной записи:

```
INSERT INTO NAKLS VALUES(GEN_ID(MAKLS_NAKLID, 1), ...)
```

Для удаления генератора используется оператор DROP:

```
DROP GENERATOR NAKLS_NAKLID
```

Совместимость типов

При выполнении операций над столбцами разного типа InterBase пытается автоматически привести типы таким образом, чтобы значения, участвующие в операции, принадлежали совместимым типам. Кроме того, для явного приведения типов можно использовать функцию CAST, которая приводит типы внутри оператора SELECT, обычно в секции WHERE:

```
CAST (<значение> | NULL AS ТипДанных)
```

Совместимыми считаются только типы DATE, CHAR и NUMERIC.

Вот как можно привести значение столбца Date_Data (тип DATE) к типу CHAR и сравнить его со значением столбца Char_Data (тип CHAR):

```
SELECT ...
  WHERE CHAR_DATA <= CAST(DATE_DATA AS CHAR);
```

Домены

Доменами называются заранее созданные описания столбцов. Наряду с другими сущностями БД, домены должны иметь уникальные имена. Однажды созданный домен хранится в БД и может использоваться вместо типа столбца. С помощью доменов достигается унификация типов данных, хранящихся в различных столбцах, возможно, разных таблиц.

Пусть, например, в столбцах Name1 и Name2 таблицы Firms будет содержаться информация о контактных лицах партнера. Оба столбца должны иметь одинаковую длину, и в них может использоваться кириллица. Объявление таблицы может выглядеть следующим образом:

```
CREATE TABLE Firms(..., Name1 VARCHAR(40) CHARACTER SET WIN1251
COLLATE PXW_CYRL, Name2 VARCHAR(40) CHARACTER SET WIN1251 COLLATE
PXW_CYRL, ...)
```

Вместо этого объявления можно сначала создать домен:

```
CREATE DOMAIN Firm_Names AS VARCHAR(40) CHARACTER SET WIN1251
COLLATE PXW_CYRL
```

Далее этот домен используется при объявлении таблицы:

```
CREATE TABLE Firms(..., Name1 Firm_Names, Name2 Firm_Names, ...)
```

Ограничения на значения столбцов

При определении (изменении) таблицы или домена на значения столбца могут налагаться ограничения. В этом случае InterBase будет автоматически отслеживать вводимые в столбец значения, отвергая те из них, которые не удовлетворяют наложенным на столбец ограничениям. Ограничение накладывается спецификатором CHECK и имеет такой формат:

```
CHECK (<ограничение>)
<ограничение> =
 {<значение> <знак операции> {<значение1> | (<выбор_одного>)}
 | <значение> [NOT] BETWEEN <значение1> AND <значение2>
 | <значение> [NOT] LIKE <маска_подобия> [ESCAPE <символ>]
 | <значение> [NOT] IN (<значение1>
  [, <значение2> ...] | <выбор_многих>)
 | <значение> IS [NOT] NULL
 | <значение> {[NOT] {= | < | >} | >= | <=}
  {ALL | SOME | ANY} (<выбор_многих>)
 | EXISTS (<выражение_выбора>)
 | SINGULAR (<выражение_выбора>)
 | <значение> [NOT] CONTAINING <значение1>
 | <значение> [NOT] STARTING [WITH] <строка>
 | NOT <ограничение>
 | <ограничение> OR <ограничение>
 | <ограничение> AND <ограничение>}

<значение> = {столбец | <константа> | <выражение> | <функция>
  | NULL | USER | RDB$DB_KEY
 } [COLLATE collation]

<константа> = число | "строка"

<функция> = {
COUNT (* | [ALL] <значение> | DISTINCT <значение>)
 | SUM ([ALL] <значение> | DISTINCT <значение>)
 | AVG ([ALL] <значение> | DISTINCT <значение>)
 | MAX ([ALL] <значение> | DISTINCT <значение>)
 | MIN ([ALL] <значение> | DISTINCT <значение>)
 | CAST (<значение> AS <тип_данных>)
 | UPPER (<значение>)
 | GEN_ID (генератор, <значение>)
 }
```

Ниже перечислены используемые обозначения:

- <знак операции> — один из следующих символов или их комбинация:

  = < > <= >= !< !> <> !=

- <выбор_одного> — оператор SELECT, возвращающий одно значение или не возвращающий ничего;
- <маска_подобия> — произвольная строка, возможно, содержащая символы-заменители % (символ процента) и/или _ (символ подчеркивания):
  - символ процента означает произвольное (в том числе и нулевое) количество любых символов, которые могут стоять на его месте;
  - символ подчеркивания означает, что на его месте должен стоять любой символ;
- <символ> — любой символ, отличный от символа-заменителя, который в маске подобия играет роль символа-заменителя;
- <выбор_многих> — оператор SELECT, который может возвращать более одного значения (список значений) или ни одного;
- <выражение_выбора> — оператор SELECT, который может возвращать более одного значения (список значений) или ни одного.

Примеры наложения ограничений:

```
CREATE TABLE MixTable(
/* Столбец не должен содержать отрицательных значений */
  Col1 INT CHECK(Col1>=0),
/* Значения столбца >= 100 и <= 200 */
  Col2 INT CHECK(Col2 BETWEEN 100 AND 200),
/* Значения столбца должны заканчиваться символами "у.е." */
  Col3 CHAR(40) CHECK(Col3 LIKE '% у.е. '),
/* Значения столбца должны заканчиваться символом "%" */
  Col4 CHAR(20) CHECK(Col4 LIKE '%!%' ESCAPE '!'),
/* Значения столбца должны совпадать с одним из значений столбца
NaklID таблицы Nakls */
  Col5 INT CHECK(Col5 IN (SELECT BookID FROM Books)),
/* Столбец должен содержать значения большие, чем значение,
возвращаемое генератором GEN_BOOKS */
  Col6 INT CHECK(Col6 > Gen_ID(GEN_BOOKS,0)),
/* Строка в значении столбца должна начинаться пробелом */
  Col7 CHAR(5) CHECK(Col7 STARTING WITH " "))
```

Ручное администрирование сервера

Под ручным администрированием понимается воздействие на сервер со стороны администратора с целью поддержания работоспособности этого сервера. Администратор также обычно назначает права доступа, создает базы данных,

осуществляет периодическое сохранение баз данных на внешних носителях информации.

Для ручного администрирования используются утилиты InterBase Service Manager и IBConsole. Их вы найдете в программной группе Пуск ▸ Все программы ▸ InterBase 7.5 Developer Edition.

В окне утилиты InterBase Server Manager отображается состояние сервера (рис. А.1).

Рис. А.1. Окно утилиты InterBase Server Manager

Если сервер запущен, в строке Status имеется слово Running, в противном случае — Stopped. По умолчанию при установке сервера он становится *службой* Windows и автоматически стартует при запуске ОС. Об этом свидетельствует установленный переключатель Automatic в группе Startup Mode. Если щелкнуть на кнопке Server Properties, вы увидите окно, показанное на рис. А.2.

В строке License указывается максимальное количество одновременных подключений к серверу (20), а в строке Number of attachments — текущее количество подключений. В строке Capabilities перечисляются обслуживаемые сервером протоколы (протоколы TCP/IP и NetBEUI означают, что сервер может использоваться в Интернете и в локальной сети), а также указывается возможность использования локального клиента (то есть клиента, размещенного на машине сервера).

За текущим состоянием сервера следит специальная утилита InterBase Guardian for Windows. Ее задача — перезапустить сервер, если он по каким-либо причинам «рухнет». Количество перезапусков, осуществленных этой утилитой, можно увидеть после щелчка в окне утилиты InterBase Server Manager (см. рис. А.1) на кнопке Guardian Properties.

Утилита IBConsole (рис. А.3) используется главным образом для создания файла БД.

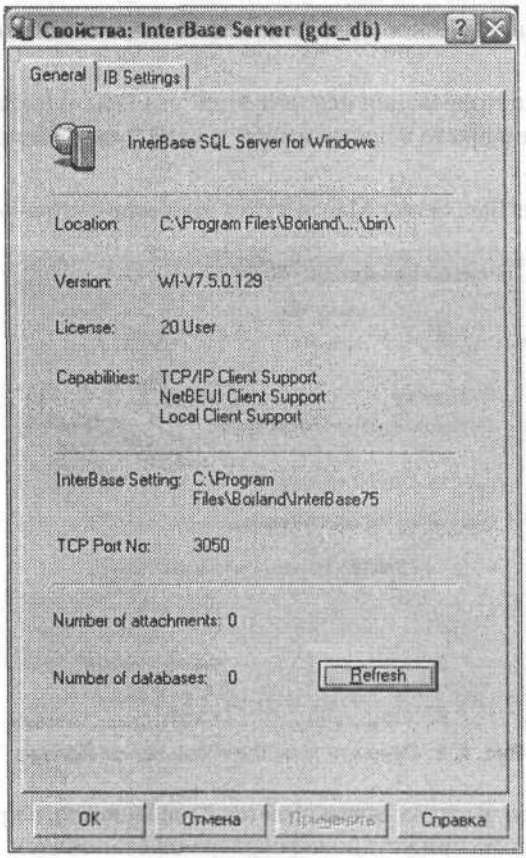


Рис. А.2. Окно Server Properties

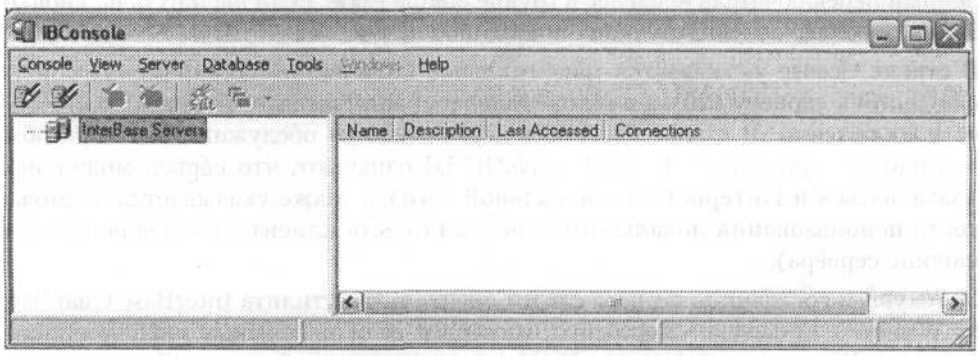


Рис. А.3. Окно утилиты IBConsole

Для работы с ней нужно прежде всего зарегистрироваться на сервере (команда Server ▸ Register). В окне регистрации (рис. А.4) следует выбрать сервер (локальный или удаленный) и ввести имя и пароль.

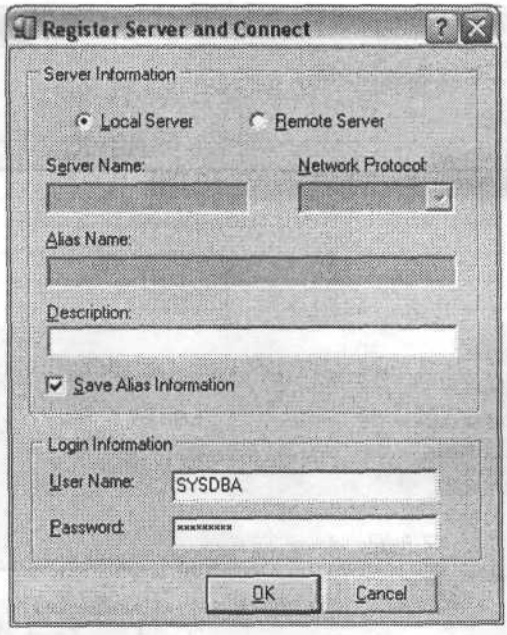


Рис. А.4. Регистрация администратора на сервере

ПРИМЕЧАНИЕ

Серверы InterBase всегда имеют хотя бы одного зарегистрированного пользователя (администратора) с именем SYSDBA и паролем masterkey (пароль чувствителен к регистру букв).

Для создания новой базы данных выберите команду Database ▸ Create Database, открыв окно, показанное на рис. А.5.

В списке File(s) следует указать маршрут доступа к файлу БД[1] и, при необходимости, ее размер в страницах. Длина страницы в байтах задается выбором в списке Page Size и по умолчанию равна 4096. Если таблицу предполагается использовать на платформе .NET, в списке Default Character Set следует выбрать пункт none — это гарантирует правильное отображение кириллицы в программах любого типа. Наконец, в строке Alias нужно ввести имя БД, под которым она будет доступна утилите IBConsole.

Для заполнения вновь созданной БД таблицами и данными щелкните на инструментальной кнопке Interactive SQL или выберите команду Tools ▸ Interactive SQL (рис. А.6).

После подготовки запроса нажмите клавиши Ctrl+E, щелкните на инструментальной кнопке Execute или, наконец, выберите в меню команду Query ▸ Execute.

Краткое справочное руководство по языку SQL вы найдете в приложении Б.

[1] Умалчиваемое расширение для файла БД — gdb. Однако вы можете использовать любое расширение или даже не использовать его вовсе.

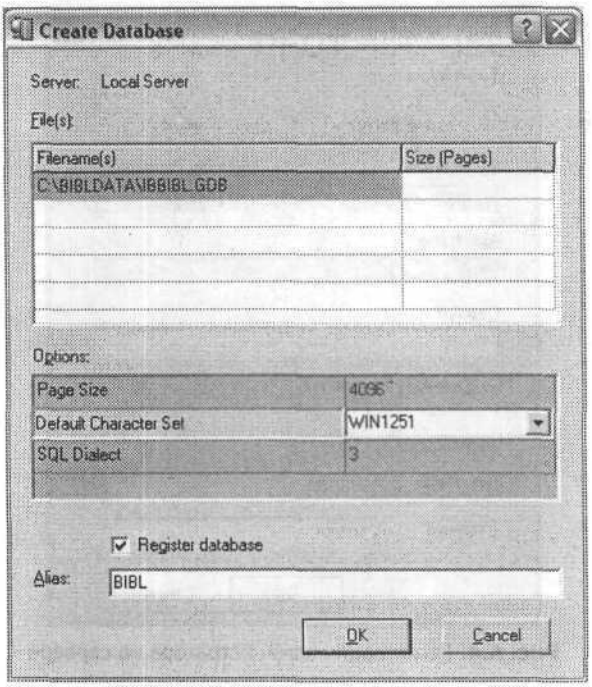


Рис. А.5. Создание новой БД

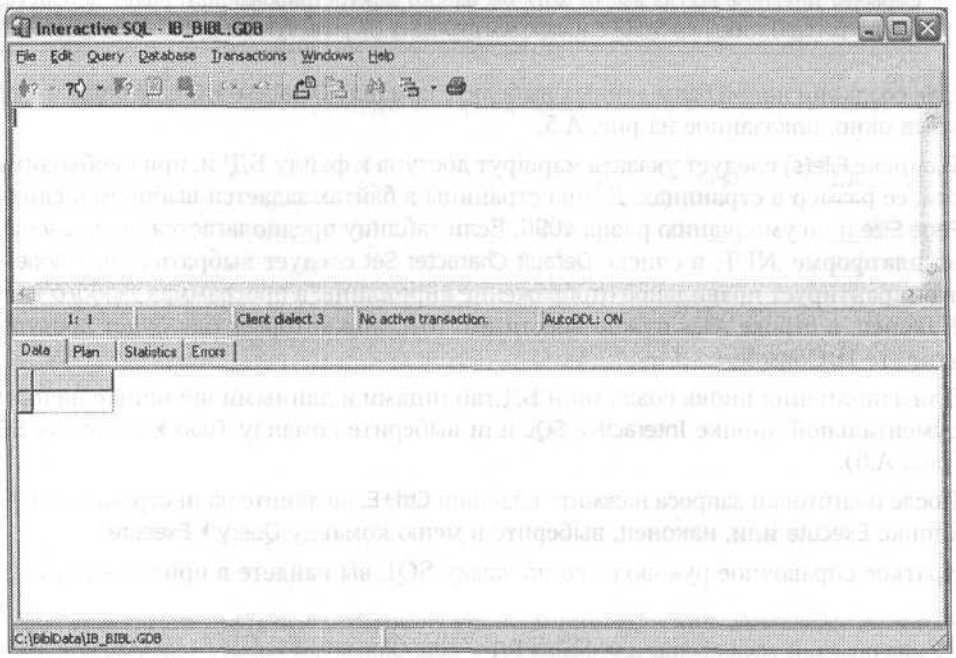


Рис. А.6. Окно для ввода SQL-запросов

Программное администрирование сервера

Базовые классы

Все компоненты группы `Service` категории InterBase Admin[1] имеют общий родительский класс `TIBCustomService`. Его основные свойства, методы и события представлены далее в табл. А.3, А.4 и А.5.

Таблица А.3. Свойства класса TIBCustomService

Свойство	Описание
`property Active: Boolean;`	Подключает/отключает БД
`property LoginPrompt: Boolean;`	Если содержит `True`, в момент подключения к БД появится диалоговое окно регистрации
`property Params: TStrings;`	Содержит набор параметров, уточняющих подключение
`type TProtocol = set of (TCP, SPX, NamedPipe, Local);`	Определяет сетевой протокол для связи с удаленным сервером
`property Protocol: TProtocol;`	
`property ServerName: String;`	Имя исполняемого файла сервера с возможным маршрутом доступа

Таблица А.4. Методы класса TIBCustomService

Метод	Описание
`procedure Attach;`	Присоединяет компонент к БД
`procedure Detach;`	Отсоединяет компонент от БД

Таблица А.5. События класса TIBCustomService

Событие	Описание
`property OnAttach: TNotifyEvent;`	Возникает при присоединении к БД
`property OnLogin: TNotifyEvent;`	Возникает в момент регистрации на сервере

Прямым потомком класса `TIBCustomService` является класс `TIBControlService`, который расширяет функциональность своего родителя за счет свойства `IsServiceRunning` и метода `ServiceStart`:

```
property IsServiceRunning: Boolean;
procedure ServiceStart;
```

Свойство содержит `True` во время выполнения службы и получает значение `False` в момент ее остановки. Метод стартует службу.

[1] Эта категория в палитре компонентов доступна только в режимах создания VCL- или Win32-приложений.

Наследником класса TIBControlService является класс TIBControlAndQueryService. В нем определены два свойства и два метода, представленные в табл. А.6 и А.7.

Таблица А.6. Свойства класса TIBControlAndQueryService

Свойство	Описание
`property BufferSize: Integer;`	Задает/указывает размер буфера
`property Eof: Boolean;`	Содержит True, если встретился конец файла

Таблица А.7. Методы класса TIBControlAndQueryService

Метод	Описание
`function GetNextChunk: String;`	Получает следующий блок данных
`function GetNextLine: String;`	Получает следующую строку данных

Класс TIBBackupRestoreService является непосредственным родителем компонентов TIBBackupService и TIBRestoreService. Его свойства перечислены в табл. А.8. Все свои методы и события он наследует от родителей.

Таблица А.8. Свойства класса TIBBackupRestoreService

Свойство	Описание
`property SQLRole: String;`	Задает/указывает SQL-роли службы
`property Verbose: Boolean;`	Если содержит True, служба выполняется в фоновом режиме

Компоненты TIBInstall и TIBUninstall наследуют от базового класса TIBSetup, свойства и события которого перечислены в табл. А.9 и А.10.

Таблица А.9. Свойства класса TIBSetup

Свойство	Описание
`property ErrorContext: Pointer;`	Устанавливает пользовательские данные для функции ошибок
`property MsgFilePath: String;`	Указывает полный маршрут доступа к файлу с сообщениями
`property Progress: Integer;`	Возвращает текущий процент выполнения задачи в диапазоне от 0 до 100
`property RebootToComplete: Boolean;`	Если содержит True, по окончании процесса компьютер перезагружается
`property StatusContext: Pointer;`	Устанавливает пользовательские данные для функции статуса

Таблица А.10. События класса TIBSetup

Событие	Описание
`type TIBSetupOnError = function (Sender: TObject; IscCode: MSG_NO; ErrorMessage: String): Integer of object;`	Возникает в случае ошибки: `IscCode` — номер ошибки; `ErrorMessage` — сообщение об ошибке
`property OnError: TIBSetupOnError;`	
`type TIBSetupOnStatus = function (Sender: TObject): Integer of object;`	Возникает периодически для информирования пользователя о состоянии процесса установки
`property OnStatusChange: TIBSetupOnStatus;`	
`type TIBSetupOnWarning = function (Sender: TObject; WarningCode: MSG_NO; WarningMessage: String): Integer of object;`	Возникает при необходимости дать предупреждение в ходе установки: `WarningCode` — код предупреждения; `WarningMessage` — текст предупреждения
`property OnWarning: TIBSetupOnWarning;`	

Компонент TIBConfigService

С помощью компонента `TIBConfigService` можно осуществлять конфигурирование некоторых параметров БД. Большинство своих свойств, методов и событий компонент наследует от родительских классов `TIBCustomService` и `TIBControlService`. Он имеет единственное собственное свойство:

```
property DatabaseName: String;
```

В этом свойстве задаются полный маршрут доступа и имя файла БД. Наиболее важные собственные методы компонента перечислены в табл. А.11.

Таблица А.11. Методы компонента TIBConfigService

Метод	Описание
`procedure ActivateShadow;`	Активизирует предварительно созданную резервную копию БД
`procedure BringDatabaseOnline;`	Подключает БД к серверу
`procedure SetAsyncMode(Value: Boolean);`	Если `Value` содержит `True`, после обращения к методу запись в БД идет в асинхронном режиме (через промежуточный буфер)
`procedure SetPageBuffers (Value: Integer);`	Устанавливает количество (`Value`) страниц для буферизации записей

продолжение

Таблица А.11 (продолжение)

Метод	Описание
`procedure SetReadonly(Value: Boolean);`	Если `Value` содержит `True`, после обращения к методу запрещается любое изменение БД
`procedure SetReserveSpace (Value: Boolean);`	Указывает количество резервных страниц для размещения разных версий одной записи
`procedure SetSweepInterval(Value: Integer);`	Указывает количество транзакций, через которое будет происходить автоматическая чистка БД от устаревших версий записей
`type TShutdownMode = set of (Forced, DenyTransaction, DenyAttachment);`	Отключает БД от сервера. Способ отключения задается параметром `Options`: `Forced` — немедленно по истечении `Wait` секунд; `DenyTransaction` — сразу после завершения очередной транзакции; `DenyAttachment` — после отключения всех пользователей
`procedure ShutdownDatabase(Options: TShutdownMode; Wait: Integer);`	

Компонент не имеет собственных событий.

Компонент TIBBackupService

Компонент `TIBBackupService` предназначен для создания копии БД. Собственные свойства компонента представлены в табл. А.12. Компонент не имеет оригинальных методов и событий.

Таблица А.12. Свойства компонента TIBBackupService

Свойство	Описание
`property BackupFile: TStrings;`	Определяет имя и, возможно, полный маршрут доступа к файлу для создания архива
`property BlockingFactor: Integer;`	Определяет количество блоков информации для копирования на ленточный накопитель
`property DatabaseName: String;`	Маршрут доступа и имя файла БД
`type TBackupOptions = set of (IgnoreChecksums, IgnoreLimbo, MetadataOnly, NoGarbageCollection, OldMetadataDesc, NonTransportable, ConvertExtTables);`	Задает параметры архивации: `IgnoreChecksums` — игнорировать контрольные суммы на каждой странице; `IgnoreLimbo` — игнорировать незавершенные транзакции; `MetadataOnly` — архивировать только метаданные: `NoGarbageCollection` — не архивировать устаревшие записи; `OldMetadataDesc` — записывать метаданные в старом стиле (для совместимости с ранними версиями InterBase); `NonTransportable` —сохраненные данные могут быть восстановлены только на машине с ОС Windows; `ConvertExtTables` — преобразовывать внешние таблицы во внутренние
`property Options: TBackupOptions;`	

Компонент TIBRestoreService

Компонент TIBRestoreService предназначен для восстановления БД из сделанной ранее архивной копии. Компонент не имеет собственных методов и событий, а его собственные свойства перечислены в табл. А.13.

Таблица А.13. Свойства компонента TIBRestoreService

Свойство	Описание
`property BackupFile: TStrings;`	Определяет имя и, возможно, полный маршрут доступа к файлу архива
`property DatabaseName: TStrings;`	Маршрут доступа и имя файла БД
`type TRestoreOption = (DeactivateIndexes, NoShadow, NoValidityCheck, OneRelationAtATime, Replace, CreateNew, UseAllSpace);` `TRestoreOptions = set of TRestoreOption;` `property Options: TRestoreOption`	Определяет параметры восстановления: DeactivateIndexes — не активизировать индексы; NoShadow — запрещает создание копии существующей БД; NoValidityCheck — не проверять ссылочные целостности; OneRelationAtATime — сохраняет одновременно вместе с таблицей все относящиеся к ней метаданные; Replace — заменять существующую в БД информацию новой; CreateNew — создать новую БД; UseAllSpace — использовать все пространство страницы для размещения записей
`property PageBuffers: Integer;`	Определяет размер буфера страниц в килобайтах
`property PageSize: Integer;`	Определяет размер страницы в килобайтах

Компонент TIBValidationService

Компонент TIBValidationService проверяет текущее состояние БД. Он является наследником класса TIBControlAndQueryService. Собственные свойства и методы компонента показаны в табл. А.14 и А.15.

Таблица А.14. Свойства компонента TIBValidationService

Свойство	Описание
`property DatabaseName: String;`	Маршрут доступа и имя файла БД
`type TTransactionGlobalAction = (CommitGlobal, RollbackGlobal, RecoverTwoPhaseGlobal, NoGlobalAction);` `property GlobalAction: TTransactionGlobalAction;`	Проверка глобальных транзакций: CommitGlobal — проверка подтверждения; RollbackGlobal — проверка отката; RecoverTwoPhaseGlobal — проверка двухфазного подтверждения; NoGlobalAction — не проверять глобальные транзакции

продолжение

Таблица А.14 (продолжение)

Свойство	Описание
`property LimboTransactionInfo: [Index: Integer] TLimboTransactionInfo;`	Открывает индексированный доступ к незавершенным транзакциям
`property LimboTransactionInfoCount: Integer;`	Количество транзакций в свойстве LimboTransactionInfo

`type TValidateOption = (LimboTransactions, CheckDB, IgnoreChecksum, KillShadows, MendDB, SweepDB, ValidateDB, ValidateFull);`
`TValidateOptions = set of TValidateOption;`
`property Options: TValidateOptions;`
	Определяет параметры проверки: LimboTransactions — получить список незавершенных транзакций; CheckDB — только проверять БД и не корректировать ее; IgnoreChecksum — игнорировать ошибки в контрольных суммах; KillShadows — удалять недействительные резервные файлы; MendDB — корректировать ошибки в БД перед ее сохранением; ValidateDB —проверять структуру БД; ValidateFull — проверить фрагментацию записей

Таблица А.15. Методы компонента TIBValidationService

Метод	Описание
`function FetchLimboTransactionInfo;`	Возвращает информацию об отложенных транзакциях
`function FixLimboTransactionErrors;`	Восстанавливает отложенные транзакции после коррекции ошибок

Компонент TIBStatisticalService

Компонент TIBStatisticalService предназначен для сбора статистических данных о БД. Его свойства указаны в табл. А.16.

Таблица А.16. Свойства компонента TIBStatisticalService

Свойство	Описание
`property DatabaseName: String;`	Имя и маршрут доступа к файлу БД

`type TStatOption = (DataPages, DbLog, HeaderPages, IndexPages, SystemRelations);`
`TStatOptions = set of TStatOption;`
`property Options: TStatOptions;`
	Определяет параметры статистики: DataPages — требуется статистика о данных в БД; DbLog — требуется протокол работы; HeaderPages — требуется информация о заголовках страниц; IndexPages —требуется информация по индексным страницам; SystemRelations — требуется информация по системным таблицам

В табл. А.17 перечислены методы компонента.

Таблица А.17. Методы компонента TIBStatisticalService

Метод	Описание
`function GetNextChunk: String;`	Получает следующий блок данных
`function GetNextLine: String;`	Получает следующую строку данных

Компонент TIBLogService

Компонент `TIBLogService` предназначен для получения от сервера протокола его работы (даты и моменты запуска, кто запустил).

Свойства, методы и события компонента унаследованы им от родительского класса `TIBControlAndQueryService`.

Компонент TIBSecurityService

Компонент `TIBSecurityService` позволяет программно управлять правами пользователей InterBase. В частности, с его помощью можно добавить нового или удалить существующего пользователя, назначить пользователю нужные права и т. д. Компонент является наследником класса `TIBControlAndQueryService`, но имеет собственные свойства, перечисленные в табл. А.18.

Таблица А.18. Свойства компонента TIBSecurityService

Свойство	Описание
`property FirstName: String;`	Имя пользователя
`property GroupID: Integer;`	Индекс группы пользователя
`property LastName: String;`	Фамилия пользователя
`property MiddleName: String;`	Отчество пользователя
`property Password: String;`	Пароль
`type TSecurityAction = (ActionAddUser, ActionDeleteUser, ActionModifyUser, ActionDisplayUser);` `property SecurityAction: TSecurityAction;`	Права пользователя: `ActionAddUser` — разрешается добавлять нового пользователя; `ActionDeleteUser` — разрешается удалять существующего пользователя; `ActionModifyUser` — разрешается изменять права пользователя; `ActionDisplayUser` — разрешается просматривать информацию о правах пользователей
`property UserID: Integer;`	Идентификатор пользователя
`property UserInfo [Index: Integer]: TUserInfo;`	Открывает индексированный доступ к зарегистрированным пользователям
`property UserInfoCount: Integer;`	Количество пользователей в списке `UserInfo`
`property UserName: String;`	Регистрационное имя пользователя

Компонент TIBServerProperties

Компонент `TIBServerProperties` позволяет получить от сервера разнообразную информацию, включая его параметры, версию, лицензионные данные и т. д. Он является прямым наследником класса `TIBCustomService`. В табл. А.19 и А.20 приводятся собственные свойства и методы компонента.

Таблица А.19. Свойства компонента TIBServerProperties

Свойство	Описание
`property ConfigParams: TConfigParams;`	Параметры конфигурации
`property DatabaseInfo: TDatabaseInfo;`	Информация о базах данных и текущее количество подключений
`property LicenseInfo: TLicenseInfo;`	Лицензионные параметры сервера

`TPropertyOption = (Database, License, ConfigParameters, Version);`
`TPropertyOptions = set of TPropertyOption;`
`property Options: TPropertyOptions;`
	Обеспечивает настройку компонента на получение нужной информации: `Database` — информации о БД (`DatabaseInfo`); `License` — информации о лицензии (`LicenseInfo`); `ConfigParameters` — информации о параметрах конфигурации (`ConfigParams`); `Version` — информации о версии сервера (`VersionInfo`)

`property VersionInfo: TVersionInfo;`	Информация о версии сервера

Таблица А.20. Методы компонента TIBServerProperties

Метод	Описание
`procedure Fetch;`	Получает всю информацию, определенную в свойстве `Options`
`procedure FetchConfigParams;`	Получает параметры конфигурации
`procedure FetchDatabaseInfo;`	Получает информацию о БД
`procedure FetchLicenseInfo;`	Получает информацию о лицензии
`procedure FetchVersionInfo;`	Получает информацию о версии

Компонент TIBLicensingService

С помощью компонента `TIBLicensingService` можно управлять лицензионными ограничениями сервера — добавлять количество разрешенных одновременных подключений пользователей (после приобретения дополнительной лицензии). Большинство серверов БД (и в их числе InterBase) поставляются на условиях лицензионного ограничения, определяющего максимальное количество

одновременных подключений к серверу и максимальное количество зарегистрированных на нем пользователей. Например, поставляемый с Delphi сервер InterBase версии 7.5 рассчитан не более чем на 20 подключений одновременно. Такое количество может подойти для некоторых небольших офисов, но в большинстве случаев его явно недостаточно. В этом случае закупается дополнительная лицензия, например, на 50 или 100 подключений.

Компонент является прямым наследником класса `TIBControlService`. Перечислены собственные свойства и методы компонента в табл. А.21 и А.22.

Таблица А.21. Свойства компонента TIBLicensingService

Свойство	Описание
`type TLicensingAction = (LicenseAdd, LicenseRemove);` `property Action: TLicensingAction;`	Определяет вид действия: `LicenseAdd` — добавление лицензии; `LicenseRemove` — удаление лицензии
`property ID: String;`	Идентификатор лицензии
`property Key: String;`	Ключ лицензии

Таблица А.22. Методы компонента TIBLicensingService

Метод	Описание
`procedure AddLicense;`	Добавляет лицензию
`procedure RemoveLicense;`	Удаляет лицензию

Компонент TIBInstall

Компонент `TIBInstall` предназначен для установки компонентов сервера. Он является прямым потомком класса `TIBSetup`. Собственные свойства и методы компонента приводятся в табл. А.23 и А.24.

Таблица А.23. Свойства компонента TIBInstall

Свойство	Описание
`property DestinationDirectory: String;`	Определяет каталог размещения сервера
`property InstallOptions: TInstallOptions;`	Определяет устанавливаемые компоненты программного обеспечения сервера
`property SourceDirectory: String;`	Каталог (устройство) с дистрибутивом сервера
`property SuggestedDestination: String;`	Возвращает каталог размещения сервера, заданный по умолчанию
`property UnInstallFile: String;`	Возвращает имя файла с информацией для процесса удаления сервера

Таблица А.24. Методы компонента TIBInstall

Метод	Описание

`TCmdOption = (cmDBMgmt, cmDBQuery, cmUsrMgmt);`
`TConnectivityOption = (cnODBC, cnOLEDB, cnJDBC);`
`TExamplesOption = (exDB, exAPI);`
`TMainOption = (moServer, moClient, moConServer, moGuiTools, moDocumentation, moDevelopment);`
	Получает описание указанного компонента сервера: `TCmdOption` — команды сервера; `TConnectivityOption` — драйверы сервера; `TExamplesOption` — примеры; `TMainOption` — основные характеристики

function `GetOptionDescription (Option: TCmdOption): String;` **function** `GetOptionDescription (Option: TConnectivityOption): String;` **function** `GetOptionDescription (Option: TExamplesOption): String;` **function** `GetOptionDescription (Option: TMainOption): String;`	
function `GetOptionName (Option: TCmdOption): String;` **function** `GetOptionName (Option: TConnectivityOption): String;` **function** `GetOptionName (Option: TExamplesOption): String;` **function** `GetOptionName (Option: TMainOption): String;`	Возвращает название нужного компонента
function `GetOptionSpaceRequired (Option: TCmdOption): Longword;` **function** `GetOptionSpaceRequired (Option: TConnectivityOption): Longword;` **function** `GetOptionSpaceRequired (Option: TExamplesOption): Longword;` **function** `GetOptionSpaceRequired (Option: TMainOption): Longword;`	Возвращает требуемое пространство диска для размещения компонента
procedure `InstallCheck;`	Проверяет готовность к установке
procedure `InstallExecute;`	Выполняет установку

Компонент TIBUnInstall

Компонент `TIBUnInstall` удаляет установленные компоненты сервера. Он — наследник класса `TIBSetup`. Собственное свойство у него единственное:

```
property UnInstallFile: String;
```

Это свойство содержит имя файла с информацией об установленных компонентах. Два его метода, соответственно, проверяют готовность к удалению и удаляют сервер:

```
procedure UnInstallCheck;
procedure UnInstallExecute;
```

Краткая справка по языку SQL

Как и всякий другой язык программирования, SQL требует определенных усилий для его освоения. Однако программисту баз данных без этого языка не обойтись. В этом приложении я постараюсь по возможности лаконично описать основные приемы создания SQL-запросов.

Для иллюстрации особенностей создания SQL-запросов удобно использовать утилиту SQL Explorer, которая вызывается из среды Delphi выбором команды Tools ▸ Explore (*если вы настроили этот пункт меню так, как описано в разделе* «Создание псевдонима БД» главы 1) или запуском файла dbexplor.exe, который находится в папке BIN каталога размещения Delphi. После вызова утилиты разыщите в окне браузера псевдоним BIBLDATA, щелкните на значке узла слева от него, чтобы получить доступ к БД «Книголюб». Поскольку эта БД — файл-серверная (или, скорее всего, локальная), в ней хранятся только таблицы. Раскройте узел Tables и щелкните на имени любой таблицы, после чего перейдите на вкладку Enter SQL (рис. Б.1).

Советую перед началом работы с запросами изменить заданный по умолчанию шрифт, который используется в окне формирования запроса: вместо пропорционального системного установите более привычный для программиста моноширинный шрифт Courier New (шрифт выбирается через меню утилиты с помощью команды View ▸ Text Font). Выполнить подготовленный запрос можно с помощью кнопки Execute query справа от панели запроса.

Простая выборка данных

Для выборки данных служит оператор SELECT. В простейшем виде этот оператор имеет такой формат:

```
SELECT Список_полей FROM Список_таблиц
```

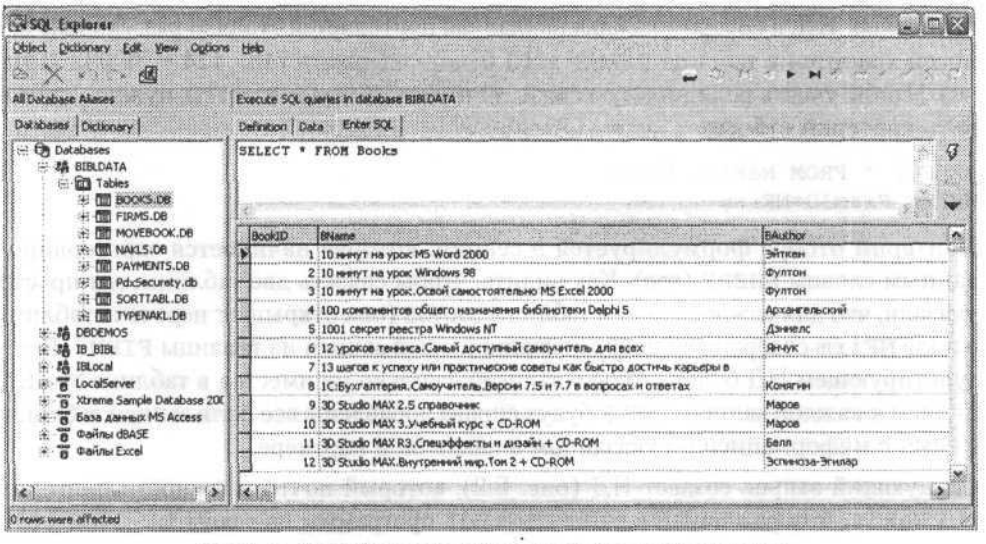


Рис. Б.1. Формирование SQL-запроса и его выполнение

Здесь SELECT (выбор) и FROM (из) — зарезервированные слова. Подобно зарезервированным словам Delphi, они не могут использоваться в качестве имен полей, таблиц или иных сущностей БД. Если из таблиц в списке таблиц выбираются все поля, вместо списка полей можно указать символ звездочки (*). Таким образом, показанный на рис. Б.1 запрос означает выбор всех полей из таблицы BOOKS.

Сразу же замечу, что SQL-интерпретатор BDE игнорирует регистр букв в зарезервированных словах, названиях полей, таблиц и других сущностей БД (индексы, ограничения и т. п.). Однако для наглядности я выделяю зарезервированные слова SQL прописными буквами.

Уже этот простейший формат выборки таит в себе массу возможностей: он позволяет формировать НД из нужных полей нужных таблиц. Например, следующий оператор выберет только три поля из таблицы Books:

```
SELECT BName, BAuthor, BPublish FROM Books
```

Выборка из связанных таблиц

Хотя простейший оператор SELECT допускает одновременную выборку данных из нескольких таблиц, в этом формате он не учитывает возможные связи между таблицами. Например, рассмотрим следующий запрос:

```
SELECT * FROM Nakls, Firms
```

Если выполнить этот запрос (я не рекомендую вам этого делать, если ваш компьютер работает с процессором на частоте менее 400 МГц), будет получен никому

не нужный НД, в котором каждая накладная повторяется столько раз, сколько записей хранится в таблице FIRMS[1] (НД будет содержать 718 · 124 = 89 032 записи). Чтобы учесть реляционную связь по полям NFirm и FirmID, нужно установить критерий отбора:

```
SELECT * FROM Nakls, Firms
WHERE FirmID=NFirm
```

Критерий отбора формулируется в секции, которая начинается зарезервированным словом WHERE (где). Как видите, чтобы связать две таблицы, мы просто указали, что для каждой записи таблицы NAKLS (она открывает перечень таблиц) в поле NFirm содержится шифр связанной с ней записи из таблицы FIRMS. В результирующем НД будет столько записей, сколько их имеется в таблице NAKLS, причем к каждой записи NAKLS будут справа добавлены все поля записи таблицы FIRMS с информацией об указанном в накладной партнере.

Следующий запрос создаст НД (рис. Б.2), который почти в точности повторит НД Nakls, отображаемый в сетке DBGrid1 программы из главы 1:

```
SELECT
  NaklID, NDate, FName, TName, NSum, NPayedSum,
  NRetSum, NCoeff, NRetDate
FROM
  Nakls, Firms, TypeNakl
WHERE
  FirmID=NFirm AND TypeID=NType
```

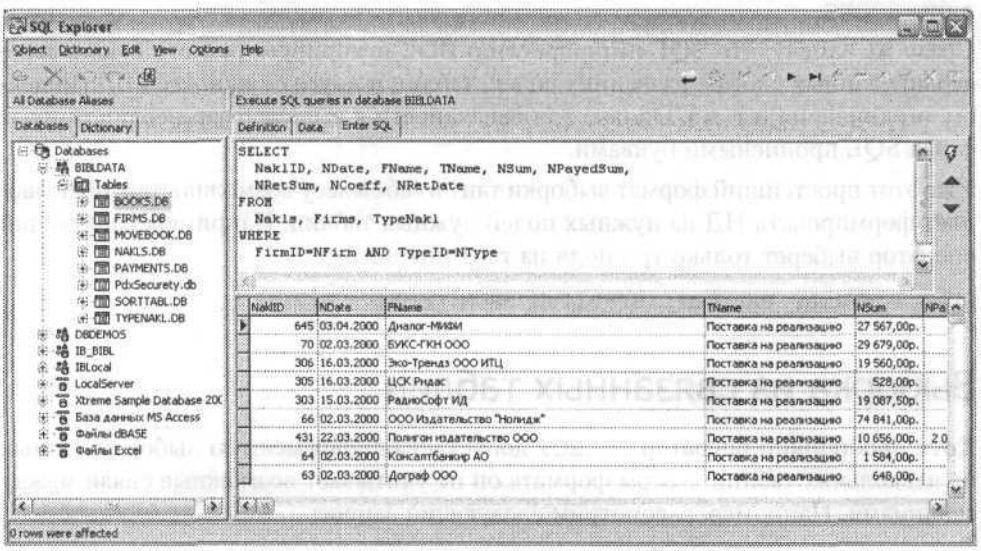


Рис. Б.2. Результат связывания трех таблиц

[1] Такое объединение таблиц иногда называют их декартовым перемножением.

ПРИМЕЧАНИЕ

В отличие от подстановочных полей в наборе Nakls программы из главы 1, связывание таблиц в запросе не требует обязательного наличия индексов по подстановочным полям, но если такие индексы есть (как в нашем случае), BDE будет их использовать.

Сортировка записей

Чтобы предыдущий пример действительно в точности повторял НД Nakls, нам нужно отсортировать записи по полю NaklID. Для сортировки используется секция, начинающаяся зарезервированными словами ORDER BY (сортировать по):

```
SELECT
  NaklID, NDate, FName, TName, NSum, NPayedSum,
  NRetSum, NCoeff, NRetDate
FROM
  Nakls, Firms, TypeNakl
WHERE
  FirmID=NFirm AND TypeID=NType
ORDER BY NaklID
```

Как и при установлении связи между таблицами, сортировать можно по любым полям, не обязательно индексным.

Сложные критерии отбора

С помощью секции WHERE можно не только связывать таблицы, но и создавать довольно сложные критерии отбора данных. Для этого в секции указывается произвольное количество операторов вида ПОЛЕ ~ ЗНАЧЕНИЕ, связываемых логическими операциями NOT, AND, OR, где знак тильды (~) означает операцию отношения (эти операции SQL совпадают с операциями отношения Delphi):

```
SELECT
  NaklID, NDate, FName, TName, NSum, NPayedSum,
  NRetSum, NCoeff, NRetDate
FROM
  Nakls, Firms, TypeNakl
WHERE
  FirmID=NFirm AND TypeID=NType AND (NSum>100000 OR NRetSum>0)
ORDER BY NaklID
```

В этом примере отбираются только те накладные, сумма которых превышает 100 000 рублей или в которых ненулевая сумма возврата (рис. Б.3). Обратите внимание на два обстоятельства. Во-первых, приоритет операций отношения в SQL выше приоритета логических операций — это избавляет от необходимости расстановки многочисленных скобок. Во-вторых, операция OR имеет меньший приоритет, чем операция AND. Если бы в последнем примере мы убрали скобки, результирующий НД содержал бы записи, в которых каждая накладная с суммой

больше 100 000 рублей повторялась бы столько раз, сколько имеется накладных с ненулевой суммой возврата.

NakID	NDate	FName	TName	NSum	NPayedSum	NRetSum
1	01.03.2000	Точка. Социальный университет	Продажа на реализацию	550,00р.	0,00р.	550,00
6	01.03.2000	Точка. МГИЭМ	Продажа на реализацию	2 328,70р.	0,00р.	2 328,70
7	01.03.2000	ООО Издательство "Нолидж"	Поставка на реализацию	155 610,00р.	60 201,65р.	95 408,35
10	01.03.2000	ООО Издательство "Нолидж"	Поставка на реализацию	155 610,00р.	0,00р.	0,00
27	01.03.2000	Точка. МГИЭМ	Продажа на реализацию	1 408,13р.	0,00р.	970,00
28	01.03.2000	Точка. МГИЭМ	Продажа на реализацию	1 500,00р.	0,00р.	261,30
32	01.03.2000	ООО Издательство "Нолидж"	Поставка на реализацию	148 200,00р.	62 371,39р.	0,00
59	02.03.2000	Точка. Институт молодежи	Продажа на реализацию	3 542,00р.	0,00р.	3 542,00
60	02.03.2000	Точка. Социальный университет	Продажа на реализацию	3 407,50р.	0,00р.	2 374,10
67	02.03.2000	Точка. Социальный университет	Продажа на реализацию	283,90р.	0,00р.	283,90
82	03.03.2000	МИДИКС ЗАО	Продажа на реализацию	37 050,00р.	23 743,25р.	631,00
88	03.03.2000	ЦСК Ридас	Поставка на реализацию	1 835,25р.	0,00р.	1 835,25
98	03.03.2000	Склад № 2	Продажа на реализацию	5 698,00р.	852,26р.	4 845,74
100	03.03.2000	Склад № 2	Продажа на реализацию	2 938,88р.	0,00р.	1 535,26
109	06.03.2000	Точка. Институт молодежи	Продажа на реализацию	5 259,10р.	0,00р.	460,00
116	06.03.2000	ООО Издательство "Нолидж"	Поставка на реализацию	395 200,00р.	0,00р.	0,00
126	07.03.2000	ООО Издательство "Нолидж"	Поставка на реализацию	123 500,00р.	0,00р.	0,00

Рис. Б.3. Результат сложного критерия отбора записей

Справа от операции отношения указывается значение. Если это значение — текстовое или типа дата-время, оно должно заключаться в обрамляющие апострофы или двойные кавычки. Кроме того, к значению текстового поля можно применять операцию `LIKE`, когда значение справа от операции `LIKE` представляет не все поле, а лишь его часть:

```
SELECT BName FROM Books
WHERE BName LIKE "Я%"
ORDER BY BName
```

Этот запрос отбирает из таблицы `BOOKS` только те названия книг, которые начинаются с буквы «Я» (рис. Б.4).

BName
Язык SQL в Delphi 5
Язык программирования C++ 3-е издание
Языки программирования.Практический сравнительный анализ

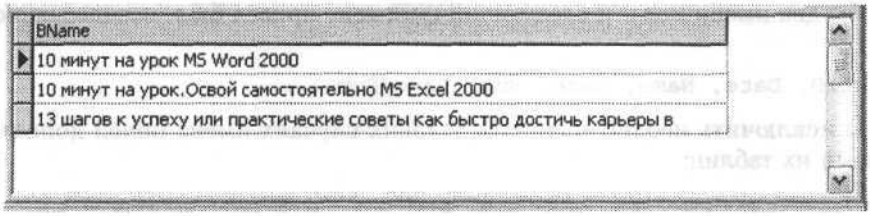
Рис. Б.4. Результат использования операции LIKE

При определении части текстового поля в операции `LIKE` символ процента (`%`) указывает, что на его месте могут стоять любые символы. Следующий запрос отберет все книги, в названиях которых хотя бы раз встречается буква «я»:

```
SELECT BName FROM Books
WHERE BName LIKE "%я%"
ORDER BY BName
```

С помощью зарезервированного слова IN (в [диапазоне]) можно перечислить несколько допустимых значений поля. Следующий запрос отбирает названия 1-й, 3-й и 7-й книг из таблицы BOOKS (рис. Б.5):

```
SELECT BName FROM Books
WHERE BookID IN (1,3,7)
```

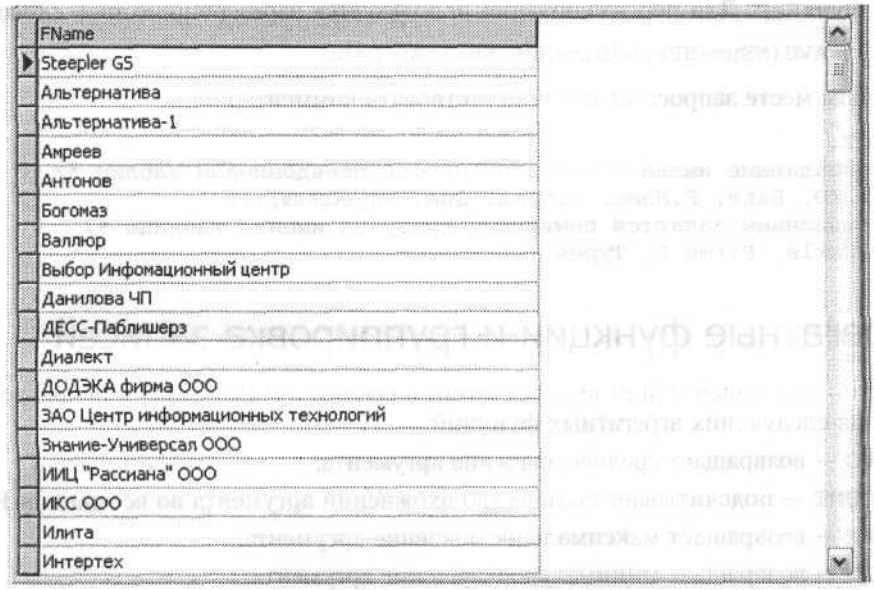


BName
10 минут на урок MS Word 2000
10 минут на урок.Освой самостоятельно MS Excel 2000
13 шагов к успеху или практические советы как быстро достичь карьеры в

Рис. Б.5. Результат использования операции IN

Вместо перечня допустимых значений в круглых скобках после операции IN можно разместить вложенный оператор SELECT. Если требуется посмотреть список всех покупателей, еще не оплативших поставленные им книги, можно сформировать такой запрос (рис. Б.6):

```
SELECT FName FROM Firms
WHERE FirmID IN
  (SELECT NFirm FROM Nakls
   WHERE NPayedSum=0 AND NType IN (1,7))
ORDER BY FName
```



FName
Steepler GS
Альтернатива
Альтернатива-1
Амреев
Антонов
Богомаз
Валлюр
Выбор Инфомационный центр
Данилова ЧП
ДЕСС-Паблишерз
Диалект
ДОДЭКА фирма ООО
ЗАО Центр информационных технологий
Знание-Универсал ООО
ИИЦ "Рассиана" ООО
ИКС ООО
Илита
Интертех

Рис. Б.6. Результат формирования списка значений вложенным оператором SELECT

Псевдонимы полей, таблиц и комментарии

В наших таблицах нет совпадающих имен полей за счет того, что каждое поле в любой таблице начинается с буквы, соответствующей начальной букве имени таблицы. А что делать, когда имена полей из двух и более таблиц совпадают? Если бы мы, например, пренебрегли нашим правилом и не поставили префиксы, появились бы два имени Name и следующий фрагмент привел бы к неоднозначности:

```
SELECT
  NaklID, Date, Name, Name, Sum, PayedSum,...
```

Чтобы исключить неоднозначность, в таких случаях имена полей дополняют именами их таблиц:

```
SELECT
  NaklID, Date, Firms.Name, Types.Name, Sum, PayedSum,...
```

Поскольку совпадающие имена таблиц могут встречаться и в других секциях запроса, для упрощения их именования в SQL разрешено вместо имени таблицы использовать ее псевдоним:

```
SELECT
  NaklID, Date, F.Name, T.Name, Sum, PayedSum,...
FROM Nakls, Firms F, Types T ...
```

Псевдоним указывается сразу за именем таблицы в секции FROM.

Иногда бывает удобно переименовать поле. Особенно часто это необходимо при обращении к агрегатным функциям (см. раздел «Агрегатные функции и группировка записей»), так как возвращаемое ими значение по умолчанию имеет имя этой функции. Для переименования используется зарезервированное слово AS:

```
SELECT AVG(NSum-NPayedSum-NRetSum) AS Debt
```

В любом месте запроса может присутствовать комментарий:

```
SELECT
/* Совпадающие имена полей дополняются псевдонимами таблиц */
  NaklID, Date, F.Name, T.Name, Sum, PayedSum,...
/* Псевдонимы задаются символами сразу за именем таблицы */
FROM Nakls, Firms F, Types T ...
```

Агрегатные функции и группировка записей

Вместо имен полей и/или вместе с ними в секции SELECT можно использовать одну из следующих агрегатных функций:

- AVG — возвращает среднее значение аргумента;
- COUNT — подсчитывает количество вхождений аргумента во все записи НД;
- MAX — возвращает максимальное значение аргумента;
- MIN — возвращает минимальное значение аргумента;
- SUM — суммирует значения аргумента.

В качестве аргумента при обращении к агрегатной функции может стоять произвольное выражение, составленное из полей НД. Например, следующий запрос возвращает среднее значение долга покупателей (рис. Б.7):

```
SELECT AVG(NSum-NPayedSum-NRetSum)
FROM Nakls
WHERE NType in (1,7)
```

AVERAGE OF NSum - NPaye
3 805,64р.

Рис. Б.7. Среднее значение долга покупателей

Список всех поставщиков с указанием суммы стоимости всех поставленных ими книг дает такой запрос (рис. Б.8):

```
SELECT SUM(NSum), FName
FROM Nakls, Firms
WHERE FirmID=NFirm AND NType in (0,6)
GROUP BY FName
```

SUM OF NSum	FName
6 915,00р.	Академия-Центр ЗАО
92 812,41р.	Альфа-книга
42 222,00р.	БУКС-ГКН ООО
27 307,50р.	Вектор О
16 071,00р.	Городец Юридическое бюро
27 567,00р.	Диалог-МИФИ
648,00р.	Дограф ООО
12 727,50р.	ДОДЭКА фирма ООО
3 300,00р.	ДС Экспресс ООО
83 165,00р.	ИКС ООО
17 028,80р.	Институт психотерапии ЗАО
1 584,00р.	Консалтбанкир АО
780,00р.	Кулаичев ПБОЮЛ
11 780,00р.	Ладья+ ООО

Рис. Б.8. Список поставщиков

Разумеется, интересно получить не только суммы поставок, но и имена партнеров, поэтому в приведенном выше запросе в секции SELECT указаны агрегатная функция и поле таблицы FIRMS. Для подобного рода запросов, в которых вместе с агрегатными функциями фигурируют и поля таблиц, SQL требует секции группировки GROUP BY, в которой в обязательном порядке перечисляются все поля, указанные в секции SELECT.

С помощью зарезервированного слова HAVING накладываются ограничения на группировку записей. В следующем запросе возвращаются все покупатели, затраченные ими на покупку книг деньги и количество купленных книг при условии, что закупалось не менее четырех книг (рис. Б.9):

```
SELECT FNAME, SUM(NSUM), COUNT(MOVEID)
FROM NAKLS, FIRMS, MOVEBOOK
WHERE FIRMID=NFIRM AND MOVEID=NAKLID AND NTYPE IN(2, 7)
GROUP BY FNAME
HAVING COUNT(MOVEID)>=4
```

FNAME	SUM	COUNT
Андреев С.В. ПБОЮЛ	56332,214	18
ГУП "ОЦ Московский Дом Книги"	44648	4
Гончаров	29717,456	4
Дементьев ЧП	55656,318	4
Климова ЧП	5635,4	4
Комбук ООО	63366,092	28
Компютер Инвайтс	9903,238	8
Наумов	3224	4
ООО Издательство "Нолидж"	190816,696	20
Ридас ЛТД-1	345694,674	34
Розничная продажа	6079999999	336
ТНДМ ООО	3958,4	4
ЦСК Ридас	7790,74	10
Яновер	985,918	4

Рис. Б.9. Ограничение на группировку

Условие HAVING отличается от WHERE тем, что в нем можно использовать агрегатные функции, а в WHERE — нельзя.

При использовании агрегатных функций иногда требуется исключить из списка повторяющиеся значения полей. Для этого служит зарезервированное слово DISTINCT. Например, представленный ниже запрос вернет количество всех накладных, связанных с поставкой книг (рис. Б.10):

```
SELECT COUNT(NFirm) FROM Nakls
WHERE NType IN (0,6)
```

COUNT OF NFirm
112

Рис. Б.10. Количество накладных поставки

В то же время следующий запрос вернет количество поставщиков (рис. Б.11):

```
SELECT COUNT(DISTINCT NFirm) FROM Nakls
WHERE NType IN (0,6)
```

COUNT OF NFirm
41

Рис. Б.11. Количество поставщиков

Создание/удаление таблиц и индексов

Для создания таблицы используется следующий оператор:

```
CREATE TABLE Имя_таблицы(Определения_полей)
```

Здесь Имя_таблицы — произвольное имя (не должно совпадать ни с одним именем уже существующих в БД таблиц); Определения_полей — список имен полей с обязательным указанием их типов. Например:

```
CREATE TABLE NewTable(Field1 CHAR(20), Field2 INT)
```

При определении текстовых полей полезно явно указывать кодировку символов, иначе в нерусифицированных версиях Windows вы не сможете в такой столбец ввести русскоязычные строки:

```
CREATE TABLE NewTable(Field1 CHAR(20) CHARACTER SET Win1251,
Field2 INT)
```

Допустимые типы полей в общем случае зависят от типа таблиц. Поскольку в файл-серверных БД Delphi создает таблицы типа Paradox, в операторе CREATE TABLE можно использовать следующие типы полей:

- CHAR(N), CHARACTER(N) — символьное поле длиной N символов (до 255);
- INT, INTEGER — соответствует типу Integer языка Delphi;
- SMALLINT — соответствует типу SmallInt языка Delphi;
- BLOB — цепочка байтов неопределенной длины;
- FLOAT — соответствует типу Real языка Delphi;
- DATE — поле для хранения даты;
- BOOLEAN — соответствует типу Boolean языка Delphi.

За типом поля при его описании можно указывать зарезервированные слова NOT NULL (поле не может быть пустым), UNIQUE (значения поля не должны повторяться).

Для удаления существующей таблицы используется оператор

```
DROP TABLE Имя_таблицы
```

Для создания индекса служит оператор

```
CREATE INDEX Имя_индекса ON Имя_таблицы (Список_полей)
```

Например:

```
CREATE INDEX Index1 ON NewTable(Field1, Field2)
```

Первичный ключ — это отсортированный индекс с уникальными значениями:

```
CREATE UNIQUE ASCENDING INDEX Books_ID ON Books(BooksID)
```

Следующий оператор удаляет ранее созданный индекс:

```
DROP INDEX Имя_индекса
```

Изменение таблиц

Под изменением таблицы понимается изменение структуры ее полей или типов ее полей. Для вставки поля используется оператор вида

```
ALTER TABLE Имя_таблицы ADD Имя_и_описание_поля
```

Например:

```
ALTER TABLE BOOKS ADD COVER BLOB SEGMENT SIZE 2048
```

Удаление поля:

```
ALTER TABLE Имя_таблицы DROP Имя_поля
```

Например:

```
ALTER TABLE BOOKS DROP COVER
```

Изменить тип поля без потери данных в общем случае нельзя. Но если новый тип совместим с прежним, то сначала создается временное поле нужного типа, в него копируется содержимое изменяемого поля, старое поле удаляется, создается одноименное старому поле нового типа, в него переносятся данные из временного поля и затем временное поле уничтожается. Например, в таблице NAKLS есть поле NFIRM типа SMALLINT. Оно используется для кода партнера. Емкость этого типа — 32 767 положительных значений. Таким образом, максимальное количество партнеров (поставщиков и покупателей) не может быть больше этого значения. Допустим, нам нужно увеличить емкость этого поля, сделав его тип INTEGER. Вот как это можно сделать:

```
/* Создаем временное поле */
ALTER TABLE NAKLS ADD NFIRM_TEMP INT
/* Переносим данные */
UPDATE NAKLS SET NFIRM_TEMP = NFIRM
/* Удаляем старое поле и переобъявляем его */
ALTER TABLE NAKLS DROP NFIRM
ALTER TABLE NAKLS ADD NFIRM INT
/* Наполняем его */
UPDATE NAKLS SET NFIRM = NFIRM_TEMP
ALTER TABLE NAKLS DROP NFIRM_TEMP
```

Вставка, удаление и редактирование записей

В таблицу БД запись вставляется с помощью оператора

```
INSERT INTO Имя_таблицы (Список_полей) VALUES (Список_значений)
```

Например:

```
INSERT INTO TypeNakl (TName, TypeID) VALUES ("Брак", 8)
```

Каждому полю в списке полей должно соответствовать нужное значение в списке значений. Если заполняются все поля записи, список полей вместе с обрамляющи-

ми скобками можно опускать. В этом случае значения в списке значений перечисляются в строгом соответствии с порядком следования полей в структуре таблицы:

```
INSERT INTO TypeNakl VALUES (8, "Брак")
```

С помощью оператора INSERT можно вставить сразу группу записей. Пусть, например, таблица создана показанным ниже оператором:

```
CREATE TABLE NewTable(MoveID INT, MNakl SMALLINT, MBook SMALLINT,
                      MQuan SMALLINT, MPrice Float)
```

Тогда следующий оператор вставит в нее список книг для накладной с идентификатором 100:

```
INSERT INTO NewTable
SELECT * FROM MoveBook
WHERE MNakl=100
```

Для удаления записей используется оператор

```
DELETE FROM Имя_таблицы WHERE Условие_выборки_записей
```

Например:

```
DELETE FROM TypeNakl WHERE TypeID=8
```

ВНИМАНИЕ!

Если в запросе DELETE опустить секцию WHERE, из таблицы будут удалены все записи.

Изменение отдельных полей таблицы реализуется оператором

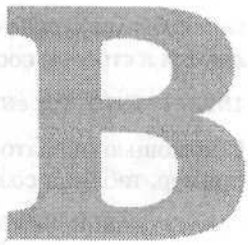
```
UPDATE Имя_таблицы SET Имя_поля=Значение WHERE Условие_выборки
```

Например:

```
UPDATE TypeNakl SET TName="Списание в брак" WHERE TypeID=8
```

За зарезервированным словом SET может стоять произвольное количество полей с указанием их значений; два соседних присваивания Имя_поля=Значение разделяются запятой. Если опустить секцию WHERE, будут изменены значения всех записей таблицы.

Краткая справка по языкам HTML и XML



Знакомство с языком HTML

Язык HTML (Hyper Text Markup Language — язык гипертекстовой разметки) является основным языком общения в гигантской Всемирной Паутине (WWW). В его основе лежит система *тегов*, с помощью которых в обычный текстовый файл вставляется служебная информация, управляющая работой браузера.

Система тегов

Знакомство с HTML начнем с короткого примера. Создайте с помощью Блокнота Windows такой текстовый файл:

```
<HTML>
<HEAD>
<TITLE>Простой пример документа HTML</TITLE>
</HEAD>
<BODY>
<H1>Заголовок первого уровня</H1>
<H2>Заголовок второго уровня</H2>
Текстовая строка первого абзаца.<BR>
Текстовая строка
второго абзаца.
</BODY>
</HTML>
```

Сохраните этот файл под именем Example1.htm. Тогда при загрузке этого файла в браузер он будет выглядеть так, как показано на рис. В.1.

Как видно из примера, теги представляют собой английские слова, обрамленные угловыми скобками. Тег <HTML> извещает о начале документа, написанного на

языке HTML, а парный ему тег </HTML> — о его конце. Эти теги необязательны, и их можно без каких-либо последствий опускать. Сам документ разделен на две неравные части: заголовок (теги <HEAD> и </HEAD>) и тело (<BODY> и </BODY>). Заголовок несет информацию о содержимом страницы. Он отображается в строке заголовка окна браузера. Его можно опускать — в этом случае вместо него в строке заголовка окажется имя файла HTML-страницы. Тело документа опускается значительно реже (в отличие от обрамляющих его тегов <BODY> и </BODY>). В нем размещаются смысловые компоненты документа — заголовки, текст, изображения, перекрестные ссылки и т. п.

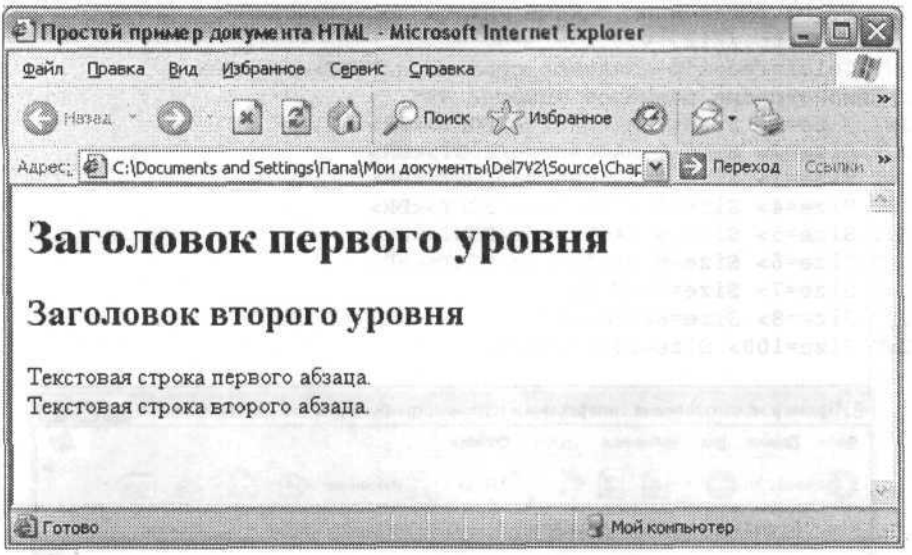


Рис. В.1. Отображение файла Example1.htm в окне браузера

Обратите внимание: почти все теги делятся на открывающие и закрывающие. Закрывающие отличаются наличием косой черты перед условным английским словом. Исключение сделано для тегов
 и <P>. Оба тега завершают текущий абзац и поэтому не имеют парных тегов. Тег <P> не только завершает текущий абзац, но и вставляет перед следующим абзацем небольшой пропуск. Если эти теги опустить, браузер считает весь текст принадлежащим одному абзацу (посмотрите на строку второго абзаца). Регистр букв в теге не имеет значения. Угловые скобки (< и >), амперсант (&) и двойные кавычки (") в рамках HTML считаются специальными символами. Если понадобится вставить в текст эти специальные символы, они заменяются следующими последовательностями:

- открывающая угловая скобка — <
- закрывающая угловая скобка — >
- амперсант — &
- двойные кавычки — &quot.

Гиперссылки

Характерной особенностью гипертекстовой страницы является возможность вставки в нее гиперссылок (гипертекстом является текст справочного hlp-файла, так что понятие гиперссылки вам знакомо). Для вставки гиперссылки используется тег A. В следующем примере в текст вставляется гиперссылка на собственную домашнюю страницу, а также демонстрируется размер шрифта (рис. В.2):

```
<TITLE>Пример использования гиперссылки и демонстрация шрифтов
</TITLE>
<H2>Демонстрация гиперссылки</H2>
Щелкните на выделенном тексте, чтобы посмотреть<BR>
<A HREF="http://127.0.0.1">
<FONT Color=red>мою домашнюю страницу</FONT></A><BR>
<H2>Демонстрация размеров шрифта</H2>
<FONT Size=1> Size=1 AAббввгг</FONT><BR>
<FONT Size=2> Size=2 AAббввгг</FONT><BR>
<FONT Size=3> Size=3 AAббввгг</FONT><BR>
<FONT Size=4> Size=4 AAббввгг</FONT><BR>
<FONT Size=5> Size=5 AAббввгг</FONT><BR>
<FONT Size=6> Size=6 AAббввгг</FONT><BR>
<FONT Size=7> Size=7</FONT>
<FONT Size=8> Size=8</FONT>
<FONT Size=100> Size=100</FONT>
```

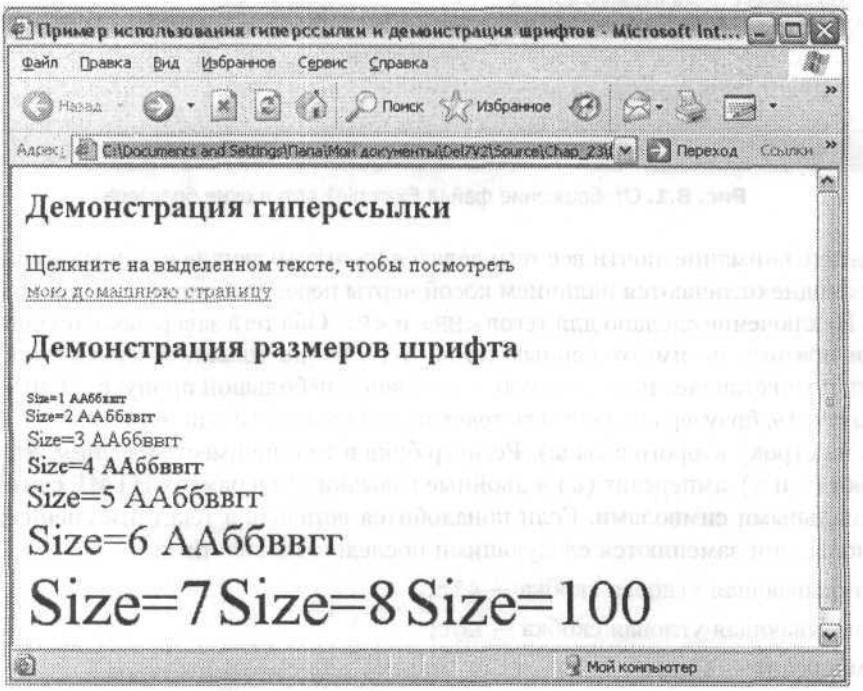


Рис. В.2. Демонстрация гиперссылки и размеров шрифта

Как видим, после тега A следует пробел, имя параметра HREF, знак равенства и в двойных кавычках URL-адрес документа, который будет вызываться после щелчка мышью на гиперссылке. В примере используется URL в виде сетевого имени локального компьютера. Если ваша домашняя страница размещена на сервере поставщика услуг Интернета или на другом сетевом компьютере, нужно соответствующим образом изменить URL.

Шрифты

Любой фрагмент текста может быть выделен шрифтом. Для этого используется тег FONT, за которым через пробел следуют один или несколько параметров; каждый параметр представляет собой пару вида ИМЯ=ЗНАЧЕНИЕ, соседние параметры разделяются точкой с запятой. Например, следующий тег определяет вывод укрупненным шрифтом зеленого цвета:

```
<FONT Size=5; Color=Green>
```

Тег FONT могут обрамлять следующие уточняющие теги:

- <B> (</B>) — полужирное начертание;
- <I> (</I>) — наклонное начертание (курсив);
- <U> (</U>) — подчеркнутое начертание;
- <TT> (</TT>) — моноширинный шрифт.

Замечу, что параметры размера и цвета шрифта могут трактоваться по-разному в браузерах разного типа. Вместо физических параметров шрифта, определяемых тегом FONT, можно использовать следующие логические теги:

- <DFN> — выделяет текст (обычно курсивом);
- <EM> — выделяет текст курсивом;
- <CODE> — выделяет текст моноширинным шрифтом;
- <STRONG> — выделяет текст полужирным начертанием.

Списки

С помощью тегов <LI> можно определять списки. Списки бывают нумерованные и маркированные. Нумерованному списку предшествует тег <OL>, маркированному — тег <UL>. В следующем примере демонстрируются логические шрифты и разного рода списки (рис. В.3):

```
<TITLE>Логические шрифты и списки</TITLE>
<H2>Теги логических шрифтов</H2>
<DFN>Этот текст выделен тегом &ltDFN&gt</DFN><BR>
<EM>Этот текст выделен тегом &ltEM&gt</EM></BR>
<CODE>Этот текст выделен тегом &ltCODE&gt</CODE><BR>
<STRONG>Этот текст выделен тегом &ltSTRONG&gt</STRONG>
<H2>Демонстрация списков</H2>
Нумерованный список:
<OL>
  <LI>Первый уровень
```

```
  <LI>Первый уровень
  <OL>
    <LI>Второй уровень
</OL>
</OL>
Маркированный список:
<UL>
  <LI>Первый уровень
  <LI>Первый уровень
  <UL>
    <LI>Второй уровень
    <UL>
      <LI>Третий уровень
</UL>
</UL>
</UL>
```

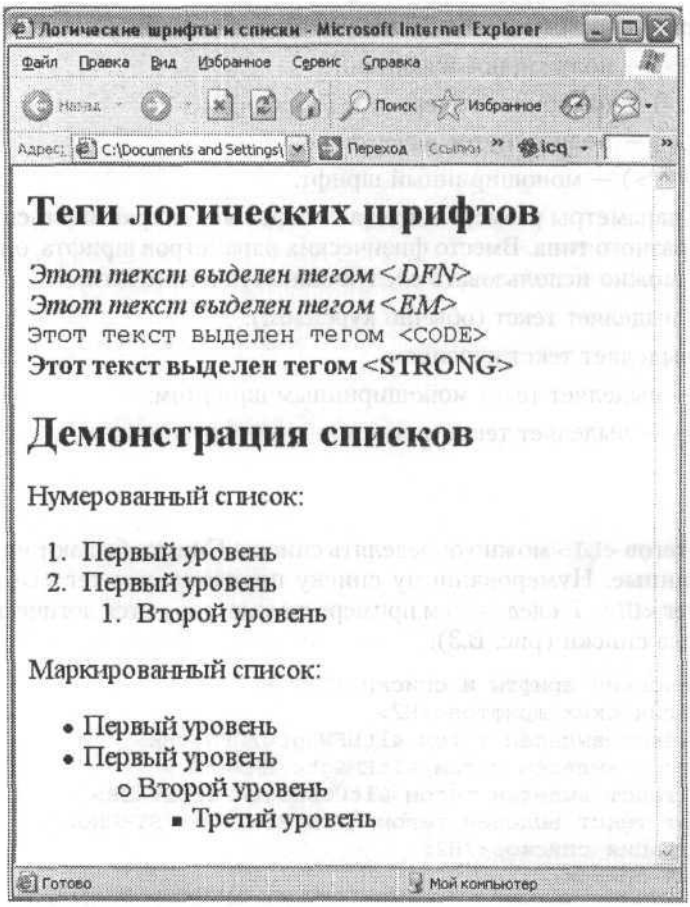


Рис. В.3. Логические шрифты и списки

Изображения

С помощью тега IMG в документ можно вставить изображение (рис. В.4):

```
<TITLE>Пример вставки в HTML-документ изображения</TITLE>
<H2><Center>Изображение вставлено с помощью тега
<CODE>&ltIMG SRC&gt</CODE></Center></H2>
<IMG SRC="pwstitle.gif" BORDER=1 ALIGN=Left HSPACE=20 VSPACE=20>
Этот текст обтекает рисунок справа. Параметр HSPASE определяет
расстояние (в пикселах) от рисунка до текста по горизонтали,
а параметр VSPACE — по вертикали.
```

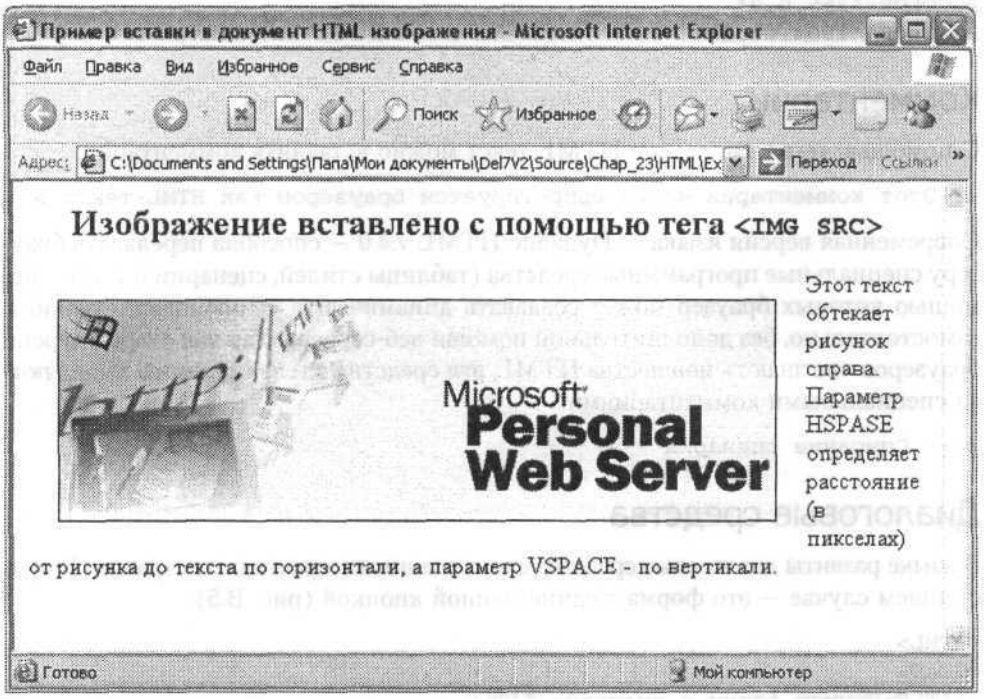


Рис. В.4. Вставка в документ изображения

С помощью параметра ALT="текстовая строка" задается альтернативный текст, заменяющий рисунок, если в браузере запрещен просмотр изображений.

Уточняющие параметры и цвет

В HTML имеется много тегов, которые могут сопровождаться уточняющими параметрами. В предыдущем примере с помощью параметров SRC и BORDER задаются, соответственно, имя файла с изображением и тип рамки, параметр ALIGN определяет выравнивание изображения относительно текста и т. д. В теге BODY с помощью параметров можно указать цвета фона и текста страницы или изображение,

которое будет служить фоном. Следующий тег определяет светло-серый цвет фона и желтый цвет текста:

```
<BODY BGCOLOR="#D3D3D3" TEXT="yellow">
```

Цвет определяется двумя способами: условным названием (`red`, `blue`, `white` и т. п.) или числом в цветовой модели RGB. В последнем случае задается триада байтов, каждый из которых определяет интенсивность красной, зеленой и синей составляющих соответственно. Следующие параметры определяют красный цвет:

```
BGCOLOR="red"
BGCOLOR="#FF0000"
BGCOLOR="255,0,0"
BGCOLOR="100%,0%,0%"
```

Комментарии

С помощью тега `<!- ...>` в HTML-текст можно вставлять комментарии:

```
<!-Этот комментарий не интерпретируется браузером как HTML-текст >
```

Современная версия языка — Dynamic HTML v.4.0 — способна передавать браузеру специальные программные средства (таблицы стилей, сценарии и т. п.), с помощью которых браузер может создавать динамически меняющиеся страницы самостоятельно, без дополнительной помощи веб-сервера. Так как старые версии браузеров «не знают» новшества HTML, эти средства в теле страницы выделяются специальными комментариями:

```
<!-- Описание сценария -->
```

Диалоговые средства

В языке развиты диалоговые средства для создания интерактивных страниц. В простейшем случае — это форма с единственной кнопкой (рис. В.5):

```
<HTML>
<HEAD>
<TITLE>Пример формы с кнопкой</TITLE>
</HEAD>
<BODY BGCOLOR="blanchedalmond">
<CENTRE>
<H2>После щелчка на кнопке "Моя страница" будет показана
умалчиваемая страница сервера</H2>
<!- Тег HR определяет горизонтальную линию:>
<HR>
<FORM ACTION="http://127.0.0.1" METHOD="GET">
<INPUT TYPE="submit" VALUE="Моя страница">
</FORM>
</CENTER>
</BODY>
</HTML>
```

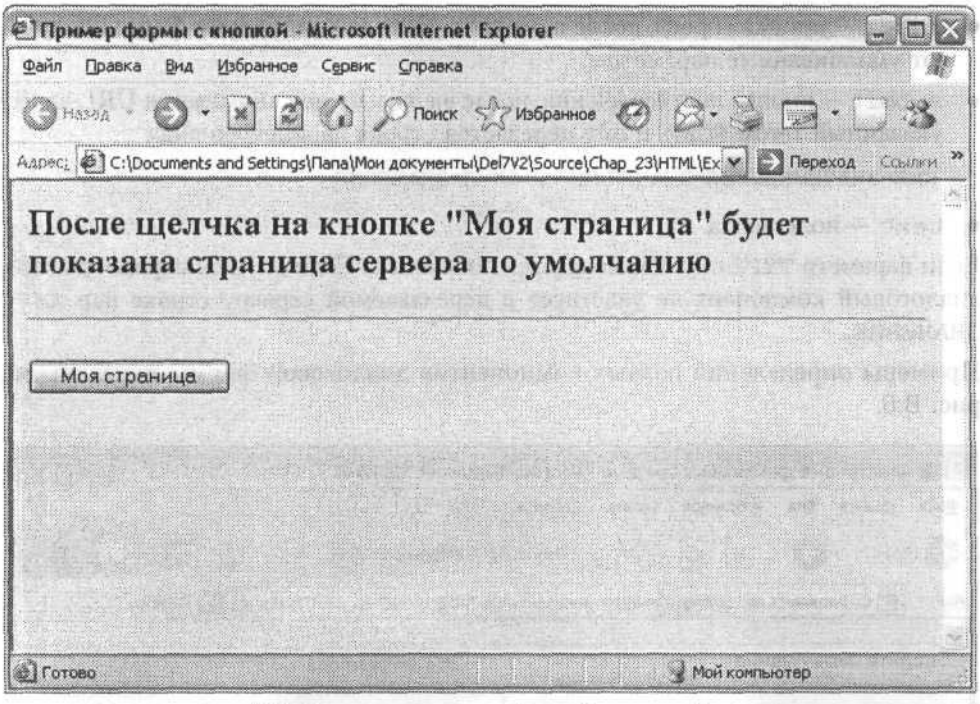


Рис. В.5. Пример страницы с формой и кнопкой на ней

С помощью параметра `ACTION` тега `FORM` определяется URL-адрес, который будет вызван после щелчка пользователя на кнопке типа `submit`. Необязательный параметр `METHOD` определяет метод протокола HTTP (см. далее), который следует использовать для обращения. Кнопки, как и некоторые другие компоненты, предназначенные для диалога с пользователем (поля ввода, флажки и переключатели), определяются тегом `INPUT` с параметрами `NAME` и `TYPE`. Параметр `NAME` задает уникальное для формы имя компонента (переключатели, входящие в одну группу, имеют одинаковые значения этого параметра). Параметр `TYPE` определяет тип вставляемого в форму диалогового элемента и может иметь одно из следующих значений:

- `button` — кнопка, для которой можно написать обработчик события `OnClick`;
- `checkbox` — флажок;
- `file` — внешний файл с описанием компонентов формы;
- `hidden` — скрытый компонент (используется для отправки на сервер данных, полученных из других форм);
- `image` — изображение, при щелчке на нем мышью на сервер отсылаются координаты указателя относительно левого верхнего угла изображения;
- `password` — строка парольного ввода (текст заменяется звездочками);
- `radio` — переключатель;

- `reset` — кнопка сброса, после щелчка на ней все компоненты формы получают умалчиваемые параметры;
- `submit` — кнопка подтверждения; после щелчка по ней вызывается URL-адрес, указанный тегом `FORM`, и ему передается строка параметров вида

  `ИМЯ=ЗНАЧЕНИЕ&ИМЯ=ЗНАЧЕНИЕ&...&ИМЯ=ЗНАЧЕНИЕ`
- `text` — поле ввода.

Если параметр `TYPE` опущен, создается поле ввода. Если опущен параметр `NAME`, диалоговый компонент не участвует в передаваемой серверу строке пар `ИМЯ=ЗНАЧЕНИЕ`.

Примеры определения разных компонентов диалоговой формы показаны на рис. В.6.

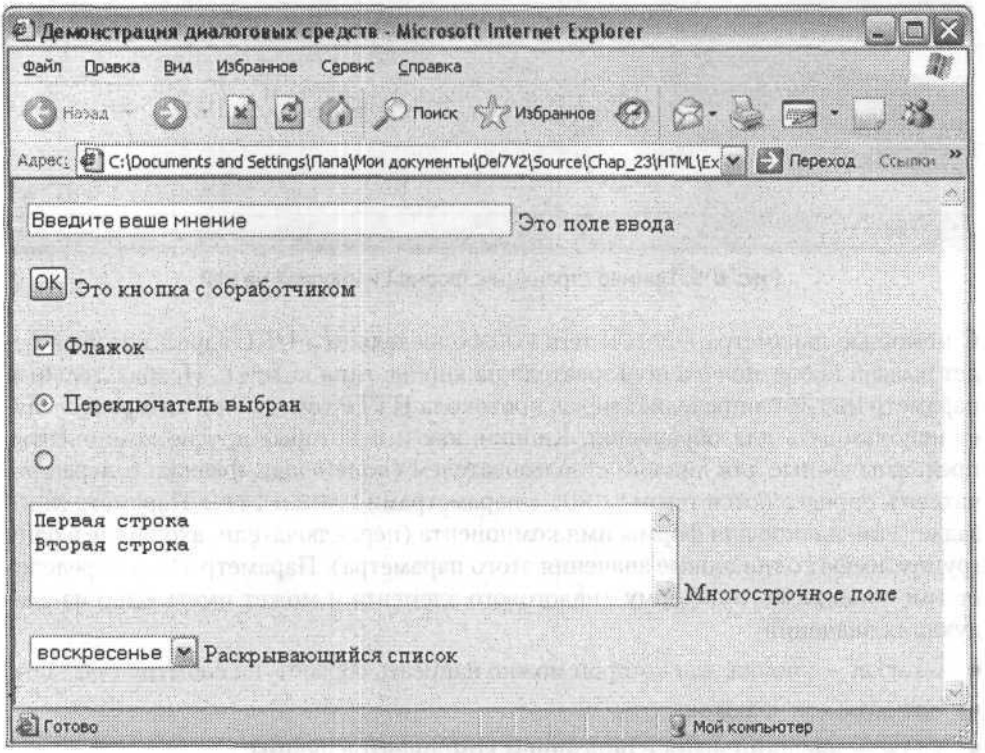


Рис. В.6. Диалоговые средства HTML

Представленные на рисунке интерфейсные элементы созданы с помощью следующего кода:

```
<TITLE>Демонстрация диалоговых средств</TITLE>
<BODY BGCOLOR="blanchedalmond">
```

```
<!-Поле ввода длиной до 50 символов с умалчиваемым значением:>
<INPUT TYPE="text" NAME="txtFavorit" VALUE="Введите ваше мнение"
SIZE=50> Это поле ввода<P>

<!-Кнопка, для которой следует написать обработчик ONCLICK:>
<INPUT TYPE="button" NAME="bthOK" VALUE="OK"
ONCLICK="alert('Нажата кнопка OK')"> Это кнопка с обработчиком <P>

<!-Флажок:>
<INPUT TYPE="checkbox" NAME="chkYes" CHECKED> Флажок<P>

<!-Два переключателя:>
<INPUT TYPE="radio" NAME="rdColor" VALUE="Черный" CHECKED> Группа<P>
<INPUT TYPE="radio" NAME="rdColor" VALUE="Белый"> переключателей<P>

<!-Многострочное текстовое поле, в котором выводятся
4 строки длиной по 52 символа (без отсечения границами компонента):>
<TEXTAREA ROWS=4 COLS=52 NAME="mmInput">
Первая строка
Вторая строка
</TEXTAREA> Многострочное текстовое поле<P>

<!-Раскрывающийся список:>
<SELECT SIZE=1 NAME="selDays">
  <OPTION VALUE="0"> воскресенье
  <OPTION VALUE="1"> понедельник
  <OPTION VALUE="2"> вторник
  <OPTION VALUE="3"> среда
</SELECT> Раскрывающийся список
</BODY>
```

В примере в качестве функции обработки события щелчка на кнопке OK используется стандартная функция `alert`, которая создает и показывает диалоговое окно с сообщением. В реальной форме для этих целей в текст помещается сценарий, написанный на языке JavaScript или VBScript.

Таблицы

Рассмотренные средства языка не позволяют проектировщику страницы располагать текст по своему усмотрению — этим занимается браузер, исходя из разрешающей способности экрана и размеров открытого окна. Однако иногда бывает необходимо точное позиционирование текста и рисунков. В этом случае в страницу вставляются таблицы, определяемые тегами <TABLE> и </TABLE>. Между ними с помощью тегов <TR> и </TR> описываются все строки таблицы. Описание строки заключается в перечислении содержимого всех ее ячеек, обрамленном тегами <TD> и </TD>. Теги TABLE могут сопровождаться многочисленными

параметрами, определяющими наличие и вид рамки, ширину и высоту таблицы, цвет фона и т. д.:

```
<TITLE>Демонстрация таблицы</TITLE>

<H2>Текст и изображения в одной таблице</H2>

<TABLE BORDER=3 BGCOLOR="blanchedalmond" BORDERCOLOR="black"
WIDTH=300>
<CAPTION ALIGN=bottom>Некоторые рисунки</CAPTION>
<TR><TD>Очки</TD><TD Align=center> <IMG SRC="glasses.bmp"></TD></TR>
<TR><TD>Почта для меня</TD><TD Align=center><IMG
SRC="emailme.gif"></TD></TR>
<TR><TD>Звезда</TD><TD Align=center><IMG SRC="star.bmp"</TD></TR>
</TABLE>
```

Этот код создает таблицу, представленную на рис. В.7. Теги `<CAPTION>` и `</CAPTION>` определяют поясняющую надпись, а необязательный параметр `ALIGN` — расположение надписи относительно таблицы.

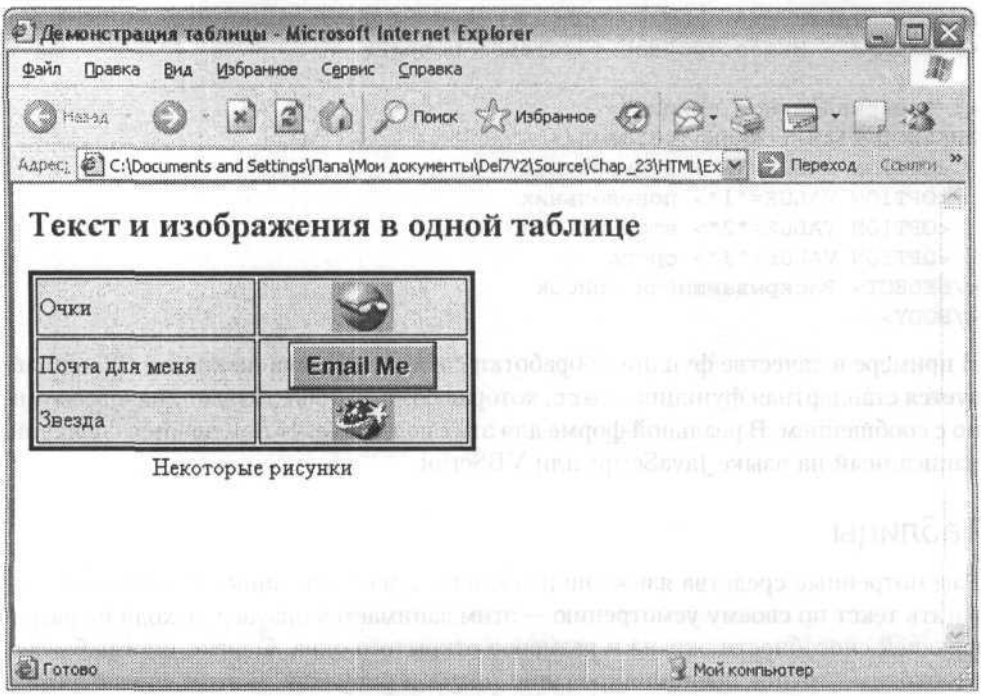


Рис. В.7. Пример таблицы

В таблицы можно вставлять изображения форматов JPEG, GIF и BMP.

Отдельные ячейки таблицы можно объединять, если в тег `<TD>` включить параметр `COLSPAN=число` или `ROWSPAN=число`. Первый служит для объединения

ячеек по горизонтали, второй — по вертикали, а параметр число задает количество объединяемых ячеек (рис. В.8):

```
<TITLE>Объединение ячеек</TITLE>
<TABLE BORDER=3 BGCOLOR="blanchedalmond" BORDERCOLOR="black"
WIDTH=300>
<CAPTION ALIGN=bottom><STRONG>Мои дети</STRONG></CAPTION>
<TR><TD></TD><TD COLSPAN=2 Align=center><B>Параметры</TD></TR>
<TR><TD></TD><TD Align=center><B>Возраст</B></TD><TD Align=center>
<B>Рост</B></TD></TR>
<TR><TD>Ольга</TD><TD Align=center>34</TD><TD Align=center>
183</TD></TR>
<TR><TD>Александр</TD><TD Align=center>15</TD><TD Align=center>
177</TD></TR>
<TR><TD>Василий</TD><TD Align=center>11</TD><TD Align=center>
160</TD></TR>
</TABLE>
```

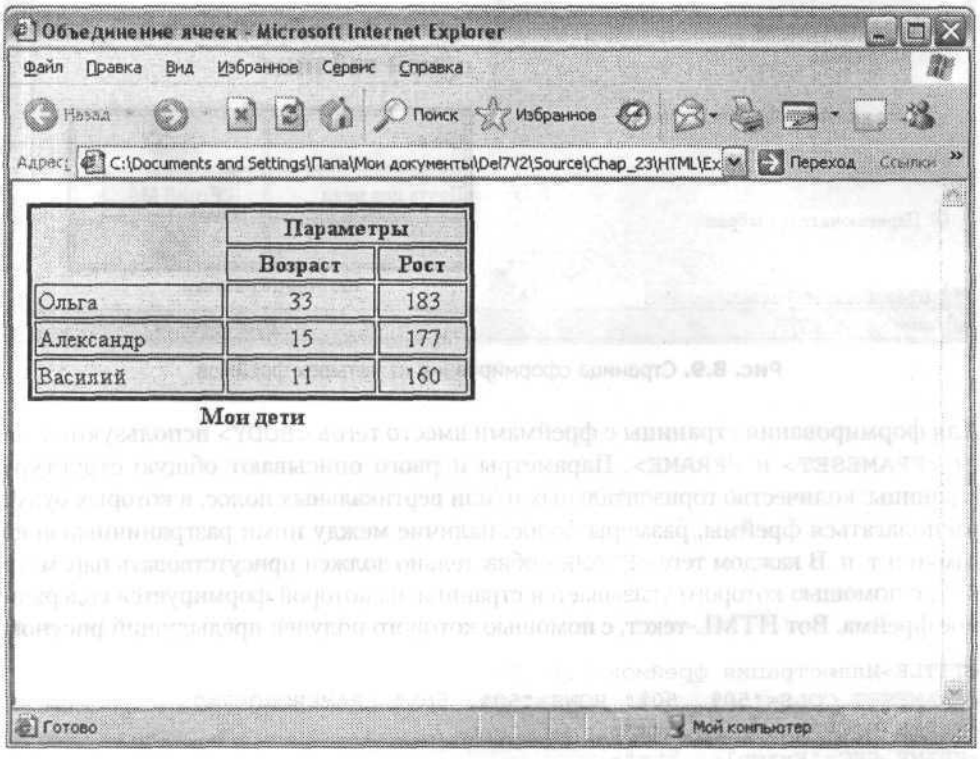


Рис. В.8. Объединение ячеек

Фреймы

Фреймы предоставляют механизм фрагментации страницы: с их помощью, например, можно получить страницу, показанную на рис. В.9.

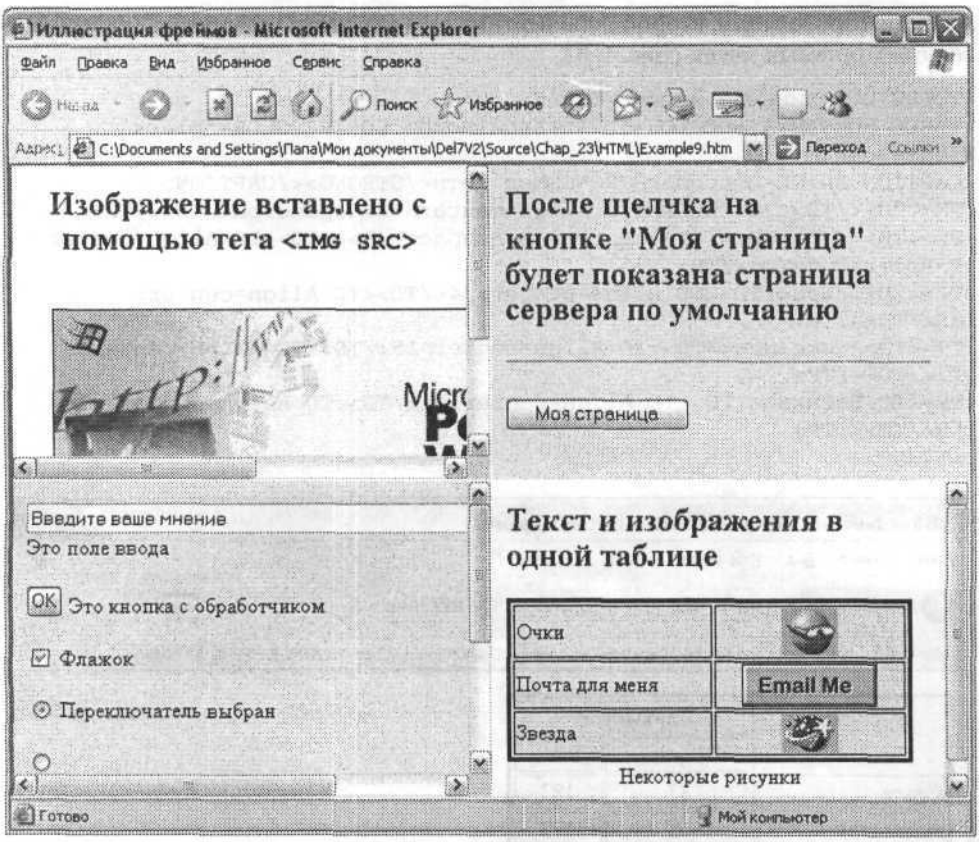


Рис. В.9. Страница сформирована из четырех фреймов

Для формирования страницы с фреймами вместо тегов <BODY> используются теги <FRAMESET> и <FRAME>. Параметры первого описывают общую структуру страницы: количество горизонтальных и/или вертикальных полос, в которых будут располагаться фреймы, размеры полос, наличие между ними разграничивающей рамки и т. п. В каждом теге <FRAME> обязательно должен присутствовать параметр SRC, с помощью которого указывается страница, из которой формируется содержимое фрейма. Вот HTML-текст, с помощью которого получен предыдущий рисунок:

```
<TITLE>Иллюстрация фреймов</TITLE>
<FRAMESET COLS="50%, 50%" ROWS="50%, 50%" FRAMEBORDER=0>
<FRAME SRC="Example4.htm">
<FRAME SRC="Example5.htm">
<FRAME SRC="Example6.htm">
<FRAME SRC="Example7.htm">
</FRAMSET>
```

Если в страницу нужно просто вставить содержимое одной или нескольких других страниц, используется так называемый *плавающий* фрейм. Он задается тегом

<IFRAME> и рядом параметров, в которых указывается источник данных и положение фрейма на странице:

```
<TITLE>Иллюстрация плавающего фрейма</TITLE>
<BODY>
<H2>Плавающий фрейм с рисунком</H2>
Размещение текста относительно плавающего фрейма.
<IFRAME SRC="Example4.htm" ALIGN=right WIDTH="50%" HEIGHT="80%">
</IFRAME>
Текст обтекает рисунок слева. Это достигается
установкой параметра <CODE>ALIGN=right</CODE> (допустимые значения
<CODE>left, right, top, bottom</CODE>).
</BODY>
```

Страница, созданная с помощью этого кода, представлена на рис. В.10.

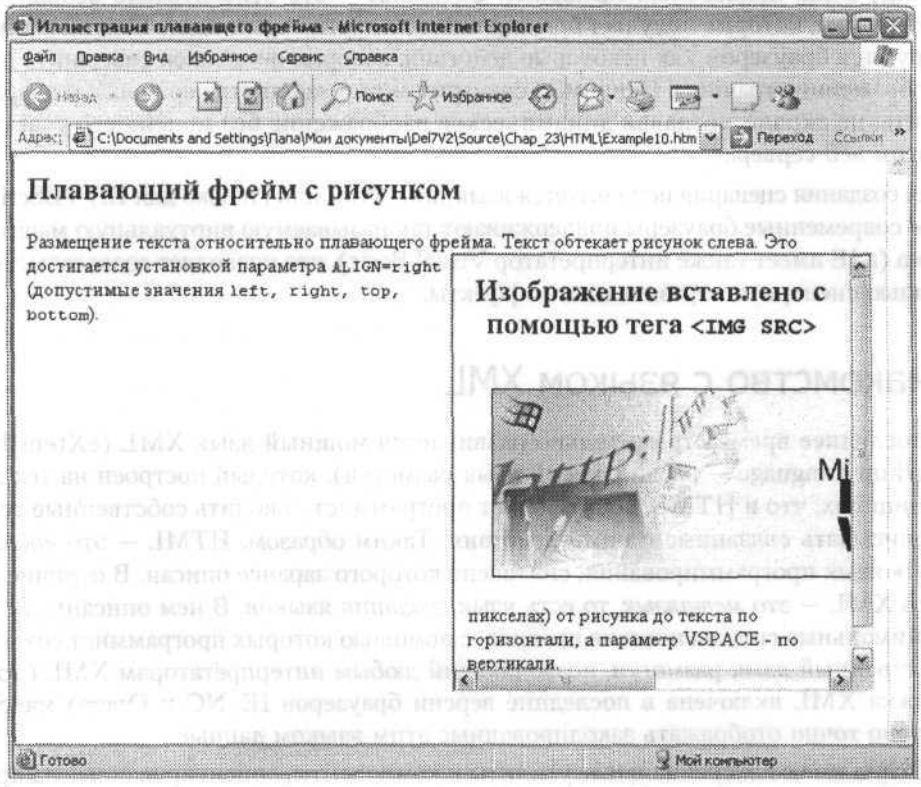


Рис. В.10. Пример плавающего фрейма

Другие возможности

Для выделения отдельных частей страницы (без фреймов) используется тег <DIV>, с помощью параметра STYLE которого можно задать стиль фрагмента: шрифт, расположение фрагмента на странице, фоновый цвет и т. п. Например:

```
<DIV ID=TopDIV STYLE="position: absolute; top: 100; left: 140;
height: 30; Width: 100; background: red; font-size: 30;
color: white; z-index: 3">
Закрывающая область
</DIV>
```

В этом тексте определяется фрагмент с именем TopDIV, располагающийся на расстоянии 100 пикселов ниже и 140 пикселов правее левого верхнего угла окна. Выводимый в нем текст будет иметь белый цвет на красном фоне. Для фрагмента определен так называемый *z-индекс*, или *индекс глубины*. Последний создает эффект 2,5-мерности изображения: чем больше z-индекс, тем «выше» на плоской проекции окна расположен данный фрагмент относительно других фрагментов.

Как видим, фрагменты можно именовать, что позволяет ссылаться на них (и их стили) в так называемых *сценариях*. Сценарии — это относительно небольшие программы, которые получает браузер вместе с другими данными. Они интерпретируются браузером как некоторые действия, которые он должен совершить при отображении страницы. Например, сценарий может заставить перемещаться фрагменты на экране, создавая динамическое изображение без малейшего участия в этом веб-сервера.

Для создания сценария используется язык JawaScript или (только для IE) VBScript. Все современные браузеры поддерживают так называемую виртуальную машину Jawa (а IE имеет также интерпретатор Visual Basic), что позволяет создавать с помощью сценариев поразительные эффекты.

Знакомство с языком XML

В последнее время стремительно развивается мощный язык XML (eXtensible Markup Language — расширяемый язык разметки), который построен на тех же принципах, что и HTML, но позволяет программисту вводить собственные теги и описывать связанные с ними действия. Таким образом, HTML — это «обычный» язык программирования, синтаксис которого заранее описан. В отличие от него XML — это *метаязык*, то есть язык создания языков. В нем описаны лишь минимальные синтаксические правила, с помощью которых программист создает собственный язык разметки, позволяющий любым интерпретаторам XML (поддержка XML включена в последние версии браузеров IE, NC и Opera) максимально точно отображать закодированные этим языком данные.

В Delphi имеются специальные утилиты и компоненты, ориентированные на преобразование информации из БД в XML-документ и обратно.

Причины разработки XML

Естественно спросить: для чего придумывать новый язык, ведь HTML до сих пор прекрасно справлялся со всеми задачами WWW. Дело в том, что HTML представляет собой *статический* набор правил разметки текста и, следовательно,

его возможности ограничены этим набором. В то же время существует множество задач (в том числе задач удобного представления любых структурированных данных, например, полученных из баз данных), которые трудно «втиснуть» в узкие рамки правил HTML. Постоянное расширение HTML невозможно по той простой причине, что это ведет к автоматическому «разбуханию» соответствующих браузеров. Значительно проще не переделывать браузеры, а «научить» их понимать метаязык типа XML, в котором правила разметки описаны в самóм документе и, следовательно, могут меняться в зависимости от конкретной задачи.

Следует оговориться, что и HTML и XML являются подмножествами значительно более мощного языка SGML (Standardized Generalized Markup Language — универсальный стандартизованный язык разметки), который был разработан и принят в качестве международного стандарта (ISO 8879) еще в 1986 году. Этот язык, рассчитанный «на все случаи жизни», не получил широкого распространения из-за своей чрезмерной сложности. Постоянно развивающиеся потребности WWW и статические правила HTML вступили в конфликт, единственным разумным выходом из которого было создание облегченного подмножества «заумного» SGML. Таким языком и стал язык XML (его описание занимает 26 страниц против более 500 страниц описания SGML).

Чтобы пояснить разницу между HTML и XML, рассмотрим такой гипотетический пример. Допустим, в нашей БД хранится информация о книгах (упоминавшаяся в главе 1 таблица BOOKS). Если бы вы захотели показать пользователям все книги по цене не более 50 р., вы не смогли бы этого сделать средствами HTML. В то же время с помощью XML эта задача легко решается, потому что при описании каждой книги на XML мы можем предусмотреть, например, теги <цена> и </цена>, между которыми можно поместить цену книги. После такой разметки нетрудно отобрать только те записи, которые содержат нужные цены.

Структура XML-документа

Особенностью структуры XML-документа является наличие в нем трех частей: раздела определения типа документа (Document Type Definition, DTD), шаблона преобразования и собственно текста документа.

Раздел DTD говорит о том, какая разметка используется в документе. Именно в этом разделе средствами XML описывается любой вновь вводимый тег. Шаблон преобразования указывает способ представления на экране того или иного тега. Наконец, в текст документа помещаются описанные в DTD теги с текстовыми данными.

Вот как, например, может быть описан элемент книга.

```
<!ELEMENT книга (название, автор, издательство, isbn?, цена?)>
<!ELEMENT книга (#PCDATA)>
<!ELEMENT автор (#PCDATA)>
<!ELEMENT издательство (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT цена (#PCDATA)>
```

Сразу же уточню, что *элементом* в XML называется комбинация обрамляющих тегов и заключенной в них текстовой информации:

```
<имя_тега> Текстовая информация </имя_тега>
```

Для описания элемента нужно указать, из каких составных элементов он состоит и что входит в каждую составную часть. В нашем примере элемент `книга` состоит из пяти вложенных элементов, причем три первых (`название`, `автор`, `издательство`) обязаны быть в информации о книге, а два других (`isbn`, `цена`) могут отсутствовать — об этом свидетельствует знак вопроса, стоящий за именем вложенного элемента. Вложенные элементы должны появляться в документе в порядке их перечисления.

Символы `#PCDATA` определяют так называемые анализируемые символьные данные (parsed character data). Анализатор браузера просматривает эти данные в поиске символа `<`, открывающего очередной тег, — вот почему этот символ нельзя помещать в символьные данные. Если понадобится отобразить его на экране, следует использовать символы-заменители `<`.

ПРИМЕЧАНИЕ

Символы-заменители всегда начинаются символом `&`, завершаются символом `;` и не могут быть пробелами.

Таким образом, приведенный выше фрагмент DTD позволяет создать такую разметку:

```
<книга>
  <название>XML. Новые перспективы WWW</название>
  <автор>Бумфрей Ф. и др.</автор>
  <издательство>ДМК</издательство>
  <isbn>5-93700-007-2</isbn>
  <цена>120р.</цена>
</книга>
```

Правильно описанный документ должен содержать *корневой элемент*. Его теги должны обрамлять всю остальную информацию. Если, например, мы хотим опубликовать перечень книг, можно создать такой корневой элемент `книги`:

```
<!DOCTYPE книги [
<!ELEMENT книги (книга*)>
...
]>
<книги>
...
</книги>
```

Тег `DOCTYPE` объявляет имя корневого элемента, а в обрамляющих скобках `[` и `]` описываются этот элемент и все входящие в него элементы. В нашем примере корневой элемент `книги` содержит 0 и более элементов `книга` — об этом свидетельствует символ `*` за именем элемента `книга`.

Шаблон преобразования указывает, как должен отображаться текст. Он всегда создается как внешний файл по отношению к основному XML-документу, в котором создается ссылка на этот файл. Шаблон представляет собой текстовый документ, составленный либо по правилам так называемых каскадных таблиц стилей, либо по правилам XML-подобного языка XSL (eXtensible Stylesheet Language — расширяемый язык таблиц стилей). Каскадные таблицы стилей узаконены и в HTML, однако для XML намного удобнее использовать мощные возможности языка XSL. Необходимость шаблона преобразования определяется тем, что в XML отсутствуют какие-либо средства форматирования отображаемого текста (шрифты, таблицы, блоки и т. п.). Без шаблона браузер не сможет нужным образом воспроизвести теги и отобразит документ так, как он выглядит в окне текстового редактора — с DTD и всеми тегами.

Таким образом, XML-документ содержит только отображаемые данные и ничего другого, в то время как шаблон преобразования определяет вид этих данных в окне браузера. Такое разделение определяет возможность динамического изменения шаблона преобразования в ответ на те или иные действия пользователя. Если, например, шаблон создает в окне браузера поле ввода и кнопку, пользователь может ввести в поле требуемую сумму и щелкнуть на кнопке. Обработчик щелчка, написанный на языке JavaScript (JScript, VBScript), заменит текущий шаблон другим, и в окне появятся сведения только о тех книгах, цена которых соответствует введенной сумме.

Если данные не предназначены для отображения в окне браузера, для XML-документа не создается шаблон преобразования. Вы можете спросить — для чего нужен такой документ? Дело в том, что язык XML чрезвычайно удобен для хранения структурированных данных, какими являются, например, наборы данных. Если вы заглянете в папку Source каталога размещения Delphi, то обнаружите в ее многочисленных вложенных папках XML-документы, не предназначенные для отображения в браузере и хранящие данные из тех или иных ТБД.

Замечательной особенностью языка XML является то, что он допускает отсутствие DTD! Можно не описывать вводимые теги, лишь бы документ отвечал двум требованиям:

- в документе должен быть единственный корневой элемент, то есть элемент, который не может быть частью никакого другого элемента;
- элементы не должны перекрывать друг друга.

Нельзя написать так:

```
<элемент1>
<элемент2>
</элемент1>
</элемент2>
```

Нужно так:

```
<элемент1>
<элемент2>
</элемент2>
</элемент1>
```

Документы, которые удовлетворяют указанным требованиям, называются *правильными*. Правильные документы, содержащие DTD и строго следующие этому описанию, называются *состоятельными*.

Перед тем как перейти к пояснению XML и XSL на примерах, следует сделать два важных замечания.

Во-первых, язык XML (XSL) чувствителен к регистру букв. Теги `<тег>` и `</Тег>` — это разные теги!

Во-вторых, документы XML/XSL должны кодироваться кодировкой Unicode (по два байта на каждый символ). Чтобы создать текстовый файл с кодировкой Unicode, нужно в диалоговом окне сохранения файла открыть список **Тип файла** или **Кодировка** и выбрать пункт **Документ Unicode**. Замечу, что популярный редактор Блокнот в Windows 95/98/2000 (но не в XP!) не может работать с кодировкой Unicode. Для этих ОС нужно использовать редактор WordPad или Word.

В то же время анализаторы браузеров Internet Explorer допускают и обычную однобайтную кодировку. В инструкции анализатору `<?xml?>` можно указать кодировку с помощью атрибута `encoding`:

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
```

Простой пример

Создадим такой XML-документ:

```
<!-- Объявление версии XML -->
<?xml version="1.0" encoding="WINDOWS-1251"?>
<пример>
  <строка.1>Знакомство с языком</строка.1>
  <строка.2>XML</строка.2>
</пример>
```

Теги `<!—Комментарий -->` определяют, как и в HTML, комментарий. Комментарий разрешается вставлять в любом месте, где можно поставить пробел.

С помощью тегов `<?...?>` в документ вставляются так называемые инструкции анализатору (processing instructions) — что-то наподобие директив компилятору в языке Delphi. Следующая инструкция извещает анализатор браузера о том, что документ создан средствами XML и использует кодировку, позволяющую вставлять в текст символы кириллицы:

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
```

Наш документ относится к правильным XML-документам: он имеет корневой элемент `<пример></пример>` с двумя вложенными в него элементами. Имена тегов должны начинаться буквой или символом подчеркивания и могут содержать произвольные буквы Unicode, цифры, точку, тире и двоеточие. Нельзя использовать символы `xml` в любом сочетании регистров как начальные символы имени тега. Спецификация XML 1.0 не накладывает ограничений на длину имен тегов.

Располагаемый между тегами текст может содержать произвольные символы Unicode, кроме открывающей уголковой скобки <. Элемент может и не содержать никакого текста — такой элемент называется *пустым*. Для него используется сокращенный синтаксис:

```
<пустой_элемент/>
```

Пустые элементы обычно имеют атрибуты, применяемые для передачи анализатору дополнительной информации.

Если для документа не указан шаблон преобразования, браузер воспроизводит весь текст документа так, как это показано на рис. В.11.

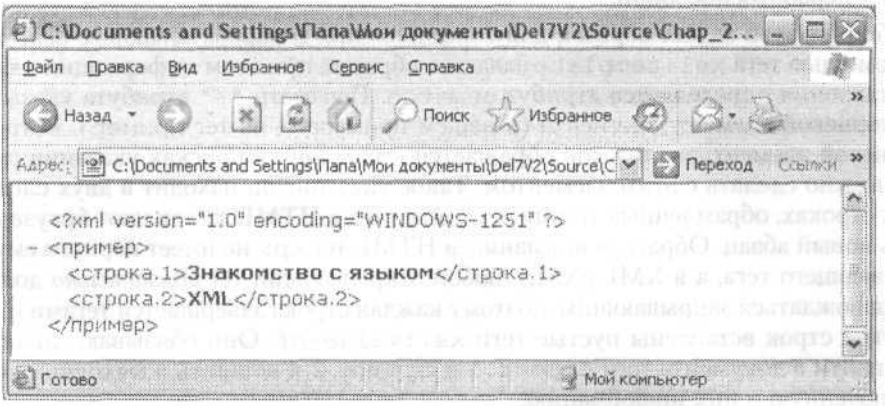


Рис. В.11. Вид документа, не связанного с таблицей стилей

Шаблон преобразования

Создадим такой шаблон преобразования:

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <p><strong><xsl:value-of select="//строка.1"/></strong></p>
    <p><xsl:value-of select="//строка.2"/></p>
  </xsl:template>
</xsl:stylesheet>
```

Теги `xsl:stylesheet` должны обрамлять тело шаблона, созданного средствами языка XSL. Помимо атрибута определения версии, открывающий тег обязан содержать ссылку на так называемое пространство имен. Эту ссылку задает следующий атрибут:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

Смысл атрибута заключается в указании анализатору на то, что где бы ни встретился элемент с префиксом `xsl:`, его нужно интерпретировать в соответствии со спецификацией XSL. И хотя параметр атрибута `"http://www.w3.org/1999/`

XSL/Transform" представляет собой правильный URL-адрес, в момент анализа текста компьютер не обязательно связан с Интернетом. Сочетание xmlns является зарезервированным словом, за которым должны стоять двоеточие и символы xsl или fo, знак равенства и URL-адрес.

При ссылке на пространство имен http://www.w3.org/1999/XSL/Transform мы получаем доступ к новейшей версии XSL — языку XSLT (буква «T» означает Transform — расширяемый язык преобразования таблиц стилей). Этот язык доступен в браузерах IE5 и выше, устанавливаемых в Windows 98/2000/XP. Если ваш браузер не поддерживает XSLT, вам нужно ссылаться на пространство имен http:/www.w3.org/TR/WD-xsl. При этом значительная часть примеров окажется неработоспособной.

Любой шаблон задает *образец* для сопоставления и обрабатывающее *действие*. С помощью тега xsl:template задается образец, при этом информация для сопоставления определяется атрибутом match. Параметр "/" атрибута указывает на корневой элемент документа (в нашем примере — на тег пример). Встретив корневой элемент, анализатор обращается к шаблону в поисках указания на то, что нужно сделать с этим элементом. Такое указание он находит в двух следующих строках, обрамленных тегами . Эти теги в HTML заставляют браузер начать новый абзац. Обратите внимание: в HTML тег не имеет парного ему закрывающего тега, а в XML (XSL) любой открывающий тег *обязательно* должен сопровождаться закрывающим, поэтому каждая строка завершается тегами . Внутри строк вставлены пустые теги xsl:value-of. Они обязывают анализатор найти в документе теги строка.1 и строка.2 и вставить в выходной поток заключенную в них информацию.

Чтобы анализатор смог найти шаблон документа, в документ нужно вставить инструкцию следующего вида:

```
<?xml:stylesheet href="Example_1.xsl" type='text/xsl'?>
```

Атрибут href ссылается на содержащий шаблон файл, а атрибут type сообщает анализатору, что он будет иметь дело с текстом на языке XSL.

В результате в окне браузера документ отображается так, как показано на рис. В.12.

Нетрудно заметить, что, изменяя шаблон, можно управлять порядком появления информации в окне браузера. Если, например, поменять названия тегов в атрибутах select, получим окно, показанное на рис. В.13.

Заметьте: исходный XML-документ остался без изменения, изменен лишь шаблон преобразования.

В рассмотренном примере вместо короткого и предельно простого кода на языке HTML использованы довольно сложные конструкции на языках XML и XSL. Вот как выглядит вариант реализации этого примера на HTML:

```
<HTML>
<BODY>
  <P><STRONG>Знакомство с языком</STRONG>
  <P>XML
</BODY>
</HTML>
```

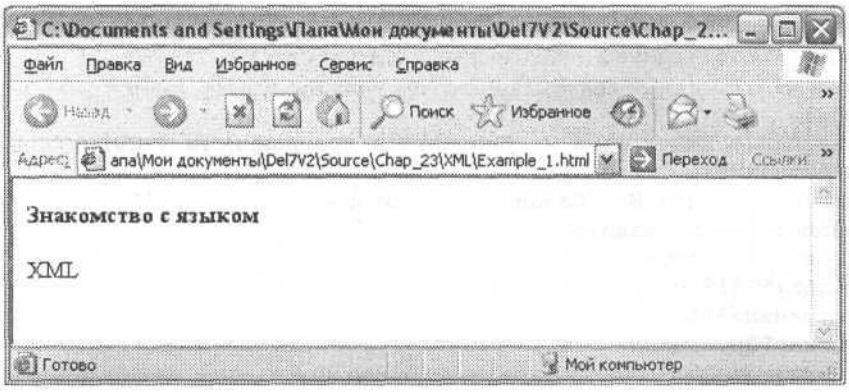


Рис. В.12. Простой пример в окне браузера

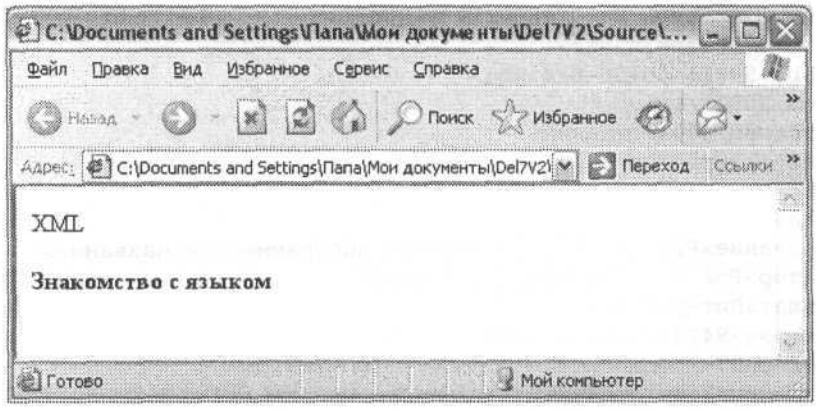


Рис. В.13. Изменение порядка следования строк

Конечно же, для создания простых страниц вроде той, что показана на рис. В.12, лучше использовать HTML. Преимущества XML в полной мере проявляются при обработке информации, содержащей множество однотипных записей, то есть полученной из наборов данных.

Обработка таблицы

Создадим следующий XML-документ:

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<?xml-stylesheet href="Example_2.xsl" type="text/xsl"?>
<книги>
  <книга>
    <название>XML. Новые перспективы WWW</название>
    <автор>Бумфрей Ф. и др.</автор>
    <издат>ДМК</издат>
    <isbn>5-93700-007-2</isbn>
    <год>2000</год>
```

```
    <страниц>688</страниц>
    <цена>120р.</цена>
  </книга>
  <книга>
    <название>XML и Java 2. Библиотека программиста (+CD)
    </название>
    <автор>Даконта М., Сазанич А.</автор>
    <издат>Питер</издат>
    <год>2001</год>
    <isbn>5-318-00187-4</isbn>
    <страниц>384</страниц>
    <цена>70р.</цена>
  </книга>
  <книга>
    <название>Электронный магазин на Java и XML</название>
    <автор>Брогден В., Минник К.</автор>
    <издат>Питер</издат>
    <isbn>5-316-00400-8</isbn>
    <год>2002</год>
    <страниц>400</страниц>
    <цена>100р.</цена>
  </книга>
  <книга>
    <название>Perl и XML. Библиотека программиста</название>
    <автор>Рэй Э., Макинтош Д.</автор>
    <издат>Питер</издат>
    <isbn>6-94723-482-3</isbn>
    <год>2003</год>
    <страниц>208</страниц>
    <цена>90р.</цена>
  </книга>
</книги>
```

В этом XML-документе описываются параметры четырех книг. Для отображения списка в таблице создадим шаблон преобразования:

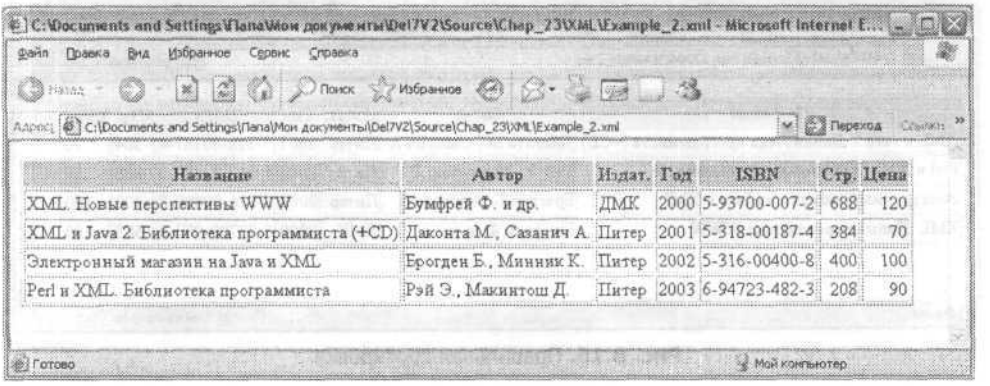
```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<table border="1">
  <tr bgcolor="#CCCCCC">
    <td align="center"><b>Название</b></td>
    <td align="center"><b>Автор</b></td>
    <td align="center"><b>Издат.</b></td>
    <td align="center"><b>Год</b></td>
    <td align="center"><b>ISBN</b></td>
    <td align="center"><b>Стр.</b></td>
    <td align="center"><b>Цена</b></td>
  </tr>
```

```
  <xsl:for-each select="книги/книга">
    <tr>
      <td><xsl:value-of select="название"/></td>
      <td><xsl:value-of select="автор"/></td>
      <td><xsl:value-of select="издат"/></td>
      <td><xsl:value-of select="год"/></td>
      <td><xsl:value-of select="isbn"/></td>
      <td align="right"><xsl:value-of select="страниц"/></td>
      <td align="right"><xsl:value-of select="цена"/></td>
    </tr>
  </xsl:for-each>
</table>
</xsl:template>
</xsl:stylesheet>
```

Для обработки всех книг используется тег `<xls:for-each>`. Его атрибут `select` должен указывать полное имя обрабатываемых тегов `книга` с перечислением всех вышележащих тегов — `"книги/книга"`. Вложенные в него теги определяют форматирование всех параметров каждой книги исходного документа.

Наличие в XSL циклических тегов `xls:for-each` существенно сокращает объем кода. Представьте себе, каким громоздким был бы HTML-документ, описывающий хотя бы сотню книг.

На рис. В.14 показано окно браузера со списком книг.



Название	Автор	Издат.	Год	ISBN	Стр.	Цена
XML. Новые перспективы WWW	Бумфрей Ф. и др.	ДМК	2000	5-93700-007-2	688	120
XML и Java 2. Библиотека программиста (+CD)	Даконта М., Сазанич А.	Питер	2001	5-318-00187-4	384	70
Электронный магазин на Java и XML	Ерогден Б., Минник К.	Питер	2002	5-316-00400-8	400	100
Perl и XML. Библиотека программиста	Рэй Э., Макинтош Д.	Питер	2003	6-94723-482-3	208	90

Рис. В.14. Список книг

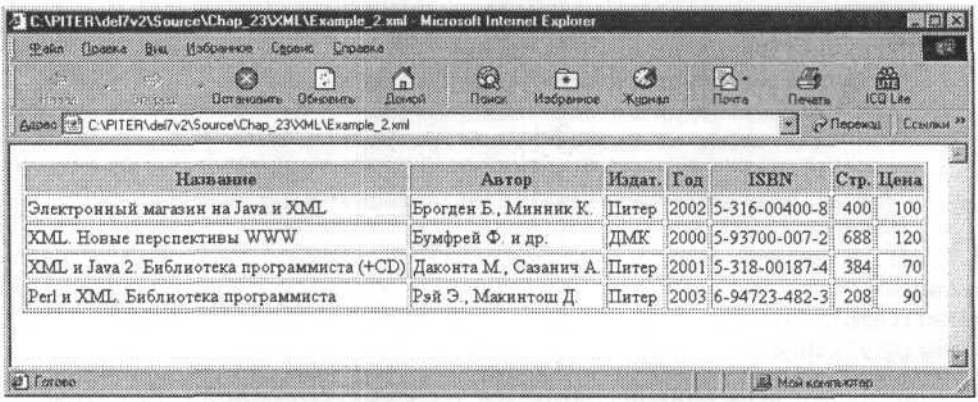
Сортировка

Разделение документа на две части — данные и шаблон преобразования — позволяет, не меняя данные, менять их вид в окне браузера.

Добавим в тег `xsl:for-each` атрибут `order-by`:

```
<xsl:for-each select="книги/книга" order-by="цена">
```

Результат показан на рис. В.15.



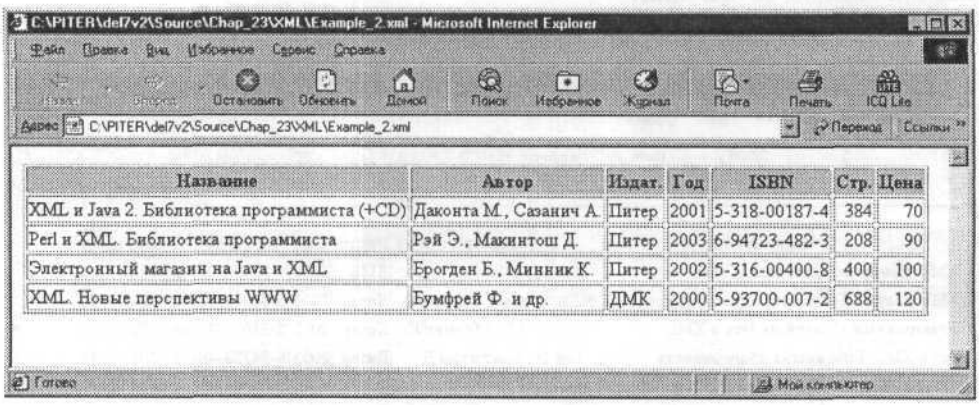
Название	Автор	Издат.	Год	ISBN	Стр.	Цена
Электронный магазин на Java и XML	Брогден Б., Минник К.	Питер	2002	5-316-00400-8	400	100
XML. Новые перспективы WWW	Бумфрей Ф. и др.	ДМК	2000	5-93700-007-2	688	120
XML и Java 2. Библиотека программиста (+CD)	Даконта М., Сазанич А.	Питер	2001	5-318-00187-4	384	70
Perl и XML. Библиотека программиста	Рэй Э., Макинтош Д.	Питер	2003	6-94723-482-3	208	90

Рис. В.15. Неверная сортировка по столбцу «Цена»

Ошибка в сортировке происходит потому, что по умолчанию анализатор считает сортируемые данные символьными. Уточним атрибут `order-by`:

```
<xsl:for-each select="книги/книга" order-by="number(цена)">
```

Теперь столбец будет отсортирован правильно (рис. В.16).



Название	Автор	Издат.	Год	ISBN	Стр.	Цена
XML и Java 2. Библиотека программиста (+CD)	Даконта М., Сазанич А.	Питер	2001	5-318-00187-4	384	70
Perl и XML. Библиотека программиста	Рэй Э., Макинтош Д.	Питер	2003	6-94723-482-3	208	90
Электронный магазин на Java и XML	Брогден Б., Минник К.	Питер	2002	5-316-00400-8	400	100
XML. Новые перспективы WWW	Бумфрей Ф. и др.	ДМК	2000	5-93700-007-2	688	120

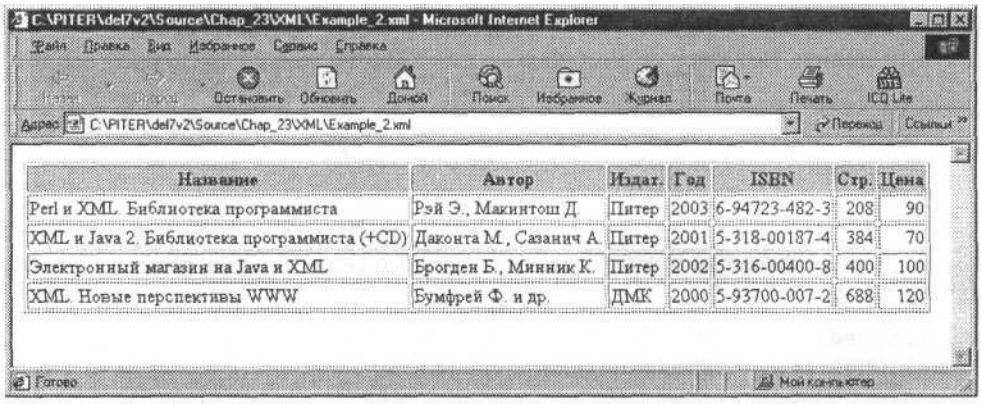
Рис. В.16. Правильная сортировка

В XSLT есть более мощные средства сортировки. Удалите атрибут `order-by` из тега `for-each` и сразу за этим тегом вставьте такой тег:

```
<xsl:sort order="ascending" select="страниц"/>
```

Таблица будет отсортирована по возрастанию количества страниц в книгах (рис. В.17).

Замена параметра `"ascending"` на `"descending"` позволяет отсортировать книги в обратном порядке.



Название	Автор	Издат.	Год	ISBN	Стр.	Цена
Perl и XML. Библиотека программиста	Рэй Э., Макинтош Д.	Питер	2003	6-94723-482-3	208	90
XML и Java 2. Библиотека программиста (+CD)	Даконта М., Сазанич А.	Питер	2001	5-318-00187-4	384	70
Электронный магазин на Java и XML	Брогден Б., Минник К.	Питер	2002	5-316-00400-8	400	100
XML. Новые перспективы WWW	Бумфрей Ф. и др.	ДМК	2000	5-93700-007-2	688	120

Рис. В.17. Сортировка по количеству страниц

Фильтрация

В XSLT определен тег `xsl:if`, с помощью которого можно отбирать записи. Единственным атрибутом этого тега может быть `test`:

```
<xsl:if test="условное_выражение">
...
</xsl:if>
```

Условное выражение содержит два операнда и знак операции сравнения между ними. Операции сравнения перечислены в табл. В.1

Таблица В.1. Операции сравнения языка XSLT

Операция	Смысл	Операция	Смысл	Операция	Смысл
`<`	Меньше	`<=`	Меньше или равно	`=`	Равно
`>`	Больше	`>=`	Больше или равно	`!=`	Не равно

Операндами могут быть имена тегов, некоторые встроенные функции и строковые последовательности. Если условное выражение истинно, выполняются все теги, вложенные в тег `xsl:if`. Вот как, например, можно показать книги по цене 100 и менее рублей:

```
<xsl:if test="цена&lt;=100">
    <tr>
      <td><xsl:value-of select="название"/></td>
      <td><xsl:value-of select="автор"/></td>
      <td><xsl:value-of select="издат"/></td>
      <td><xsl:value-of select="год"/></td>
      <td><xsl:value-of select="isbn"/></td>
      <td align="right"><xsl:value-of select="страниц"/></td>
      <td align="right"><xsl:value-of select="цена"/></td>
    </tr>
  </xsl:if>
```

Встроенная функция position() возвращает число, соответствующее номеру текущей записи. Вот как с ее помощью можно вывести только две первые записи:

```
<xsl:if test="position()&lt;3">
    <tr>
      ...
    </tr>
  </xsl:if>
```

Более сложные фильтры можно создать с помощью тегов choose, when и otherwise:

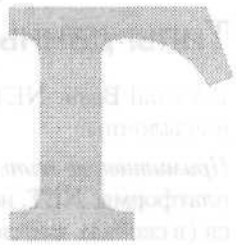
```
<xsl:choose>
  <xsl:when test="Выражение_для_проверки">
    <!- Обработка -->
  </xsl:when>
  <xsl:when test="Другое_выражение">
    <!—Другая обработка -->
  </xsl:when>
  <xsl:otherwise>
    <!--Обработка, если не выполнена ни одна другая обработка -->
  </xsl:otherwise>
</xsl:choose>
```

Концепция объектной модели документа

Концепция объектной модели документа (Document Object Model, DOM) предназначена для средств обработки, какими являются прежде всего сценарии на языках JavaScript, JScript, VBScript. В этой модели XML-документ рассматривается как набор именованных объектов, имеющих связанные с ними свойства и методы. Например, любой документ имеет объекты document и documentElement. Первый содержит весь документ в целом, второй — корневой элемент со всеми вложенными в него дочерними элементами. Для объекта documentElement определены свойство childList, содержащее список всех дочерних элементов, и метод addChild, с помощью которого можно вставить в документ новый элемент.

Замечу, что концепция DOM строго стандартизована, то есть она не зависит от конкретного языка программирования.

Краткая справка по языку Visual Basic .NET



В этом приложении приведены краткие справочные данные по языку программирования Visual Basic .NET. В нем описаны синтаксис языка, типы данных, переменные, преобразования типов, выражения и операторы.

Синтаксис языка

Подобно Delphi, алфавит Visual Basic .NET содержит все латинские буквы и знак подчеркивания. Последний является также знаком переноса части оператора на следующую строку. В этом случае он не должен заканчивать идентификатор:

```
    'Следующий перенос неверен:
Console.WriteLine_
            ("Hello!")
    'Нужно так:
Console.WriteLine _
            ("Hello!")
```

Естественным признаком конца оператора является конец строки. Оператор, как это показано выше, может занимать несколько строк. На одной строке могут располагаться несколько операторов. В этом случае между соседними операторами ставится двоеточие:

```
Console.WriyeLine("Hello!") : Console.ReadLine()
```

Как и в Delphi, в Visual Basic .NET не учитывается регистр букв, но среда Delphi (и Visual Studio) выделяет зарезервированные слова прописными начальными символами (**Dim**, **As**, **Sub** и т. д.).

Символ апострофа (**'**) применяется для вставки комментария в текст программы. Его можно вставить в любое место строки, при этом все символы от апострофа до конца строки образуют комментарий.

Типы данных

В Visual Basic .NET имеется три группы типов данных: примитивные, значащие и ссылочные.

Примитивные типы данных совпадают с таковыми в CTS (общей системой типов) платформы .NET, но их названия в некоторых случаях отличаются. К ним относятся (в скобках дается название типа в CTS) `Boolean`, `Byte`, `Char`, `Date` (`DateTime`), `Decimal` (вещественное, 16 байт), `Double` (вещественное, 8 байт), `Integer` (`Int32`), `Long` (`Int64`), `Short` (`Int16`), `Single` (вещественное, 4 байта). Названия всех примитивных типов являются зарезервированными словами.

Значащие типы содержат в себе примитивные типы, а также перечисления и структуры (аналог записей в Delphi).

Ссылочные типы содержат ссылки на реальные объекты. К ним относятся классы, массивы, интерфейсы, делегаты и строки.

Объявления переменных

При объявлении переменных используются зарезервированное слово **`Dim`** или модификаторы доступа **`Public`**, **`Protected`**, **`Friend`**, **`Protected Friend`**, **`Private`** и **`Static`**. При объявлении сначала указывается одно из перечисленных слов, затем имя переменной, зарезервированное слово **`As`** и тип переменной. Переменную можно объявлять в любом месте программы до ее первого использования. При объявлении разрешается инициировать переменную (присваивать ей начальное значение). Например:

```
Dim Counter As Integer = 1
Private IsCheck As Boolean = False
Protected Btn As Button = New Button
```

Конструкция **`New`** `имя_класса` используется для создания экземпляра нужного класса.

Массивы

Массивы объявляются так же, как и скалярные переменные, но после имени массива должны идти круглые скобки, в которых через запятую указываются верхние значения индексов размерностей (нижнее значение индекса по любой размерности равно нулю). Максимальное количество размерностей — до 60. Например:

```
Dim Matrix(2, 2) As Double ' Массив 3 x 3 вещественных чисел
Dim Vector(11) As Integer  ' Одномерный массив из 12 целых чисел
```

Как и обычную переменную, массив можно инициировать в момент объявления. При этом в скобках не указываются верхние границы индексов, а лишь запятые, если массив многомерный:

```
Dim Array1() As Integer = (1, 2, 3) 'Массив из трех целых чисел
Dim Array2( , ) As Integer = (0+1, 0+2), (1+0, 1+3)
```

Последнее объявление идентично таким строкам:

```
Dim Array2(1, 1) As Integer;
  Array2(0, 0) = 1: Array2(0, 1) = 2
  Array2(1, 0) = 0: Array2(1, 1) = 3
```

Если в качестве типа элементов массива используется тип `Object`, массив может хранить данные разных типов:

```
Dim Book(2) As Object;
  Book(0) = "Название"
  Book(1) = "Автор"
  Book(2) = 123.00
```

В программе можно использовать динамические массивы, то есть массивы, длина которых изменяется в ходе прогона программы. Для этого массив объявляется с пустыми скобками. На этапе прогона определяется необходимая длина массива по каждому измерению и вызывается оператор **ReDim**:

```
Dim Arr() As Object
Dim I, J As Integer
...
  I = 2
  J = 2
  ReDim Arr(I, J)
```

Преобразования типов

Различают явное и неявное преобразования типов. Первое делается в контексте программы, без ведома программиста. Например:

```
Dim I As Integer
Dim X As Double
X = 123.45
I = X    ' Неявное преобразование значения типа Double к типу Integer
```

Замечу, что неявное преобразование может привести к потере точности (в нашем случае `I` получит значение `123`) и, как правило, не способствует повышению надежности программ.

Для явного преобразование используются функции `CBool`, `CByte`, `CChar`, `CDate`, `CObj`, `CInt`, `CShort`, `CSng`, `CStr`, `CType`. Все они (кроме `CType`) получают единственный параметр типа **Object** (то есть любого типа: в Visual Basic .NET тип **Object** трактуется как `Variant`) и возвращают результат нужного типа (`Bool`, `Byte`, `Char` и т. д.). Функция `CType` получает два параметра: значение и тип, к которому приводится это значение:

```
Dim S As String = "123"
Dim X As Integer = CType(S, Integer)
```

Выражения и операции

В табл. Г.1 перечислены арифметические и логические операции Visual Basic .NET.

Таблица Г.1. Арифметические и логические операции

Операция	Знак операции
Сложение	+
Вычитание	-
Перемена знака	-
Умножение	*
Деление	/
Целочисленное деление	\
Остаток от деления по модулю	`Mod`
Возведение в степень	^
Сравнение	< > <= >= = <>
Сравнение объектов	`Is`
Подобие	`Like`
Логическое И	`And`
Логическое ИЛИ	`Or`
Логическое НЕТ	`Not`
Исключающее ИЛИ	`Xor`
Сокращенное И	`AndAlso`
Сокращенное ИЛИ	`OrAlso`

Сокращенные операции **`AndAlso`** и **`OrAlso`** имеют такой синтаксис:

```
выражение1 AndAlso выражение2
выражение1 OrAlso выражение2
```

Они действуют подобно операциям И и ИЛИ, но не вычисляют `выражение2`, если `выражение1` дает однозначный результат (`False` для **`AndAlso`** и `True` для **`OrAlso`**).

Операторы присваивания

Оператор присваивания в Visual Basic .NET выполняет ту же функцию, что и в Delphi: он вычисляет значение выражения, стоящего справа от знака присваивания, и присваивает его переменной, стоящей слева от знака. В отличие от Delphi, знаком присваивания является знак равенства (=), а не символы двоеточия и знака равенства (:=).

В Visual Basic .NET используются также префиксные формы операторов присваивания: &=, +=, -=, *=, /=, \=, ^=. Они действуют следующим образом: над

правым (или левым, см. ниже) выражением осуществляется нужная операция и полученное суммируется со значением переменной слева от знака операции:

```
Dim S As String = "Hello, "
  S &= "World!"   ' S = "Hello, World! "
Dim i As Integer = 123
  I += 7          ' i = 130
Dim k As Integer = 2
  k ^= 3          ' k = 8
Dim b As Double = 123.45
  b *= -1         ' b = -123.45
```

Условный оператор

Условный оператор Visual Basic .NET подобен такому же оператору Delphi, но позволяет обойтись без операторных скобок. Для этого используются зарезервированные слова **End If**:

```
 If условие Then
< произвольное количество операторов >
Else
< произвольное количество операторов >
End If
```

Если используется по одному оператору в каждом блоке, формы условных операторов в Visual Basic .NET и Delphi совпадают:

```
If условие Then оператор: If условие Then оператор1 Else оператор2
```

Для сложных проверок можно использовать зарезервированное слово **ElseIf**:

```
If A = 0 Then
  Consile.Writeln("ноль")
ElseIf A > 0 Tneh
  Console.Writeln("больше нуля")
Else
  Console.Writeln("меньше нуля")
End If
```

Оператор выбора

Оператор выбора в Visual Basic .NET подобен оператору **case** в Delphi, но позволяет обойтись без операторных скобок:

```
Select Case переменная
    Case значение1
            < произвольное количество операторов >
    Case значение2
            < произвольное количество операторов >
     ...
End Select
```

В приведенной синтаксической форме значениеX может быть конкретным значением переменной выбора, но может также определять диапазоны значений:

```
Sub ChackValue(ByVal num As Integer)
    `Select Case num
      . Case 1
            Console.WriteLine("1")
        Case 2, 3
            Console.WriteLine("2 или 3")
        Case 4 To 6
            Console.WriteLine("от 4 до 6")
        Case Is > 7
            Console.WriteLine("больше 7")
    End Select
End Sub
```

Оператор For

Как и во многих других случаях, оператор позволяет обходиться без операторных скобок:

```
For k = начальное_значение To конечное_значение
  < группа операторов >
Next k
```

Оператор While

Оператор с предварительной проверкой условия выполняет вложенные в него операторы до тех пор, пока тестируемое условие остается истинным:

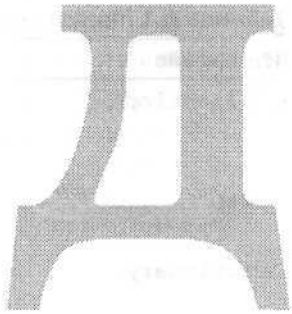
```
While условие
< группа операторов >
End While
```

Оператор Do

Оператор с последующей проверкой условия выполняет вложенные в него операторы до тех пор, пока тестируемое условие остается истинным (вариант **While**) или ложным (вариант **Until**):

```
Do
    < группа операторов >
Loop While условие
Do
    < группа операторов >
Loop Until условие
```

Использование классов общего назначения платформы .NET Framework



Классы общего назначения играют вспомогательную роль, но их значение трудно переоценить. В этом приложении рассматриваются коллекции, классы для работы со строками и файлами, а также сериализация объектов.

Коллекции

Концепция *коллекций* не нова — в VCL существуют классы `TList`, `TStringList` и `TCollection`. Все они решают схожие задачи: обеспечивают удобные средства хранения объектов разного типа. В CTS существует пространство имен `System.Collection`, которое имеет свои интерфейсы и классы для обслуживания коллекций. К коллекциям относятся списки, стеки, массивы, хэш-таблицы, очереди и словари.

Интерфейсы пространства имен System.Collection

В табл. Д.1 перечислены специальные интерфейсы для работы с коллекциями.

Таблица Д.1. Интерфейсы для работы с коллекциями

Интерфейс	Назначение
`IEnumerable`	Предоставляет интерфейс для перечисления элементов коллекции в одном направлении

продолжение

Таблица Д.1 (продолжение)

Интерфейс	Назначение
`ICollection`	Определяет размер и методы синхронизации коллекции. Наследует методы интерфейса `IEnumerable`
`IList`	Обслуживает коллекцию, элементы которой упорядочены по индексу
`IDictionary`	Обслуживает коллекцию, состоящую из пар «ключ–значение»
`IEnumerator`	Определяет методы коллекции с перебором в одном направлении
`IDictionaryEnumerator`	Определяет перечисление на базе словаря (класс `Dictionary`)
`ICompare`	Определяет методы для сравнения двух коллекций
`IHashCodeProvire`	Определяет хэш-код для указанного объекта

Интерфейс `IEnumerable` содержит единственный метод:

```
function GetEnumerator: IEnumerator;
```

Свойства и методы интерфейса `IEnumerator` перечислены в табл. Д.2.

Таблица Д.2. Свойства и методы интерфейса IEnumerator

Свойство, метод	Назначение
`property Current: Object;`	Возвращает текущий элемент коллекции
`function MoveNext: Boolean;`	Делает текущим следующий элемент коллекции и возвращает `False`, если коллекция исчерпана
`procedure Reset;`	Делает текущим первый элемент коллекции

Свойства и методы интерфейса `ICollection` представлены в табл. Д.3.

Таблица Д.3. Свойства и методы интерфейса ICollection

Свойство, метод	Назначение
`property Count: Integer;`	Возвращает количество элементов, содержащихся в коллекции
`property IsSynchronized: Boolean;`	Возвращает `True`, если класс `Collection` безопасен в отношении потоков
`property SynchRoot: Object;`	Возвращает объект, обеспечивающий безопасный в отношении потоков доступ к коллекции
`procedure CopyTo(Ar: array; Index: Integer);`	Копирует коллекцию в массив `Ar`, начиная с индекса `Index`

ПРИМЕЧАНИЕ

Безопасными в отношении потоков называются экземпляры классов, которые позволяют сразу нескольким потокам команд обращаться к методам и свойствам объекта без угрозы возникновения конфликтов.

Свойства и методы интерфейса `IList` показаны в табл. Д.4.

Таблица Д.4. Свойства и методы интерфейса IList

Свойство, метод	Назначение
`property IsFixedSize: Boolean;`	Возвращает `True`, если объект `List` имеет фиксированный размер
`property IsReadOnly: Boolean;`	Возвращает `True`, если объект `List` доступен только для чтения
`property Item(Index: Integer): Object;`	Возвращает элемент коллекции по его индексу
`function Add(Value: Object): Integer;`	Вставляет элемент в коллекцию и возвращает его индекс
`procedure Clear;`	Очищает коллекцию
`function Contains(Value: Object): Boolean;`	Возвращает `True`, если коллекция содержит указанный элемент
`function IndexOf(Value: Object): Integer;`	Возвращает индекс указанного элемента
`procedure Insert(Index: Integer; Value: Object);`	Вставляет элемент в нужное место коллекции
`procedure Remove(Value: Object);`	Удаляет элемент из коллекции
`procedure RemoveAt(Index: Integer);`	Удаляет из коллекции элемент с указанным индексом

Свойства и методы интерфейса `IDictionary` перечислены в табл. Д.5.

Таблица Д.5. Свойства и методы интерфейса IDictionary

Свойства, методы	Назначение
`property IsFixedSize: Boolean;`	Возвращает `True`, если объект `Dictionary` имеет фиксированный размер
`property IsReadOnly: Boolean;`	Возвращает `True`, если объект `Dictionary` доступен только для чтения
`property Item(Key: Object): Object;`	Возвращает элемент коллекции по его ключу
`property Kyes: ICollection;`	Возвращает коллекцию ключей
`property Values: ICollection;`	Возвращает коллекцию значений

продолжение

Таблица Д.5 (продолжение)

Свойства, методы	Назначение
`procedure Add(Key, Value: Objrct);`	Добавляет элемент в коллекцию
`procedure Clear;`	Очищает коллекцию
`function Contains(Key: Object): Boolean;`	Возвращает `True`, если элемент с указанным ключом содержится в коллекции
`function Remove(Key: Object);`	Удаляет элемент с указанным ключом

Даже беглый обзор интерфейсов показывает, что элементами, хранимыми в коллекциях, могут быть только значения ссылочных типов. Это связано с самой природой коллекций, которые представляют собой наборы указателей на хранящиеся в куче данные. Если в коллекции потребуется сохранить значение значимого типа (например, `Integer`), то можно использовать один из следующих способов:

- Перевести это значение в строку:

```
var
  i: Integer;
...
  Collection.Add(i.ToString);
```

- Создать класс-оболочку и сохранять экземпляры этого класса:

```
type
  WrapperClass = class(&Objetc)
    public
      Value: Integer;
      constructor Create(Val: Integer); override;
  end;

constructor WrapperClass.Create(Val: Integer);
begin
  inherited Create;
  Value := Val;
end;

...
Collection.Add(WrapperClass.Create(123));
```

Классы пространства имен System.Collection

Помимо описанных интерфейсов, пространство имен содержит соответствующие классы-коллекции.

Класс Stack

Класс `Stack` хранит объекты по принципу программного стека: первым пришел, последним ушел (FILO — First In, Last Out). Он исполняет интерфейсы `ICollection`,

IEnumerable и IClonable. Последний не входит в пространство имен System.Collection. Он имеет метод Clone, который используется для создания новых идентичных объектов.

Собственные методы класса Stack перечислены в табл. Д.6.

Таблица Д.6. Методы класса Stack

Метод	Назначение
`function Peek: Object;`	Возвращает элемент, находящийся в вершине стека
`function Pop: Object;`	Удаляет элемент с вершины стека и коллекции
`procedure Push(Obj: Object);`	Помещает новый элемент в вершину стека
`function ToArray: Object;`	Помещает все элементы стека в объект класса ArrayList

Для демонстрации класса Stack подготовлена такая программа (проект Add_5\Collections\StackDemo.dpr):

```
program StackDemo;

{$APPTYPE CONSOLE}

uses
  System.Collections;

procedure ShowCollection(Collection: IEnumerable; Rem: String);
// Эта вспомогательная процедура печатает строку REM
// и текущее содержимое стека
var
  EnumStack: IEnumerator;
begin
  EnumStack := Collection.GetEnumerator;
  Writeln;
  Writeln(Rem);
  while EnumStack.MoveNext do
    Writeln(EnumStack.Current);
end;

var
  MyStack: Stack;
  k: Integer;
begin
  // Создаем коллекцию:
  MyStack := Stack.Create;

  // Наполняем стек шестью строками:
  for k := 1 to 6 do
```

```
    MyStack.Push('Строка ' + k.ToString);

  // Показываем его содержимое:
  ShowCollection(MyStack, 'Начальное состояние стека:');

  // Удаляем вершину стека:
  MyStack.Pop;
  ShowCollection(MyStack, 'Метод POP:');

  // Вставляем в стек новый элемент:
  MyStack.Push('===Новая вершина стека===');
  ShowCollection(MyStack, 'Метод PUSH:');

  Readln;
end.
```

На рис. Д.1 показан результат работы программы.

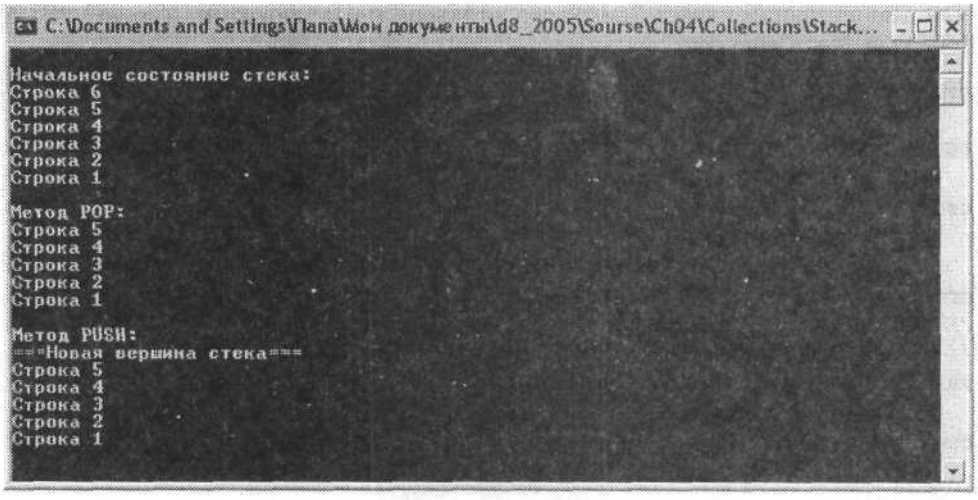


Рис. Д.1. Демонстрация стека

Специально для удобства работы с коллекциями в язык Delphi введен оператор **for... in...do**. С его помощью можно несколько упростить процедуру ShowCollection:

```
procedure ShowCollection(Collection: IEnumerable; Rem: String);
var
  k: &Object; // Тип параметра цикла должен совпадать с типом
              // элементов коллекции или быть &Object
begin
  Writeln;
  Writeln(Rem);
    for k in Collection do
      Writeln(k.ToString);
end;
```

ПРИМЕЧАНИЕ

Параметр цикла **for...in...do** должен иметь такой же тип, как и элементы коллекции. Если тип элементов неизвестен или в коллекции хранятся элементы разных типов, назначайте ему тип `&Obect`, который совместим с любым типом CTS, а имеющийся в нем метод `ToString` преобразует содержимое значения в строку.

Класс Queue

Класс `Queue` хранит элементы по принципу классической очереди: первым пришел, первым ушел (FIFO – First In, First Out). Подобно стеку, очередь исполняет интерфейсы `ICollection`, `IEnumerable`, `IClonable`.

В табл. Д.7 представлены собственные методы коллекции `Queue`.

Таблица Д.7. Методы класса Queue

Метод	Назначение
procedure `Enqueue;`	Добавляет очередной элемент в конец очереди
function `Dequeue: Object;`	Возвращает первый элемент очереди и удаляет его из коллекции
function `Peek: Object;`	Возвращает (но не удаляет) первый элемент очереди
function `ToArray: Object;`	Копирует элементы очереди в массив
procedure `TrimToSize;`	Устанавливает емкость коллекции равной количеству хранящихся в ней элементов

Класс `Queue` демонстрирует следующая программа, результат работы которой показан на рис. Д.2 (проект Add_5\Collections\QueueDemo.dpr):

```
program QueueDemo;

{$APPTYPE CONSOLE}

uses
  System.Collections;

procedure ShowCollection(Collection: IEnumerable; Rem: String);
var
  S: String;
begin
  Writeln;
  Writeln(Rem);
  for S in Collection do
    Writeln(S);
end;

var
  MyQueue: Queue;
  k: Integer;
```

```
begin
  // Создаем очередь на 6 элементов:
  MyQueue := Queue.Create(6);

  // Наполняем очередь шестью строками:
  for k := 1 to 6 do
   MyQueue.Enqueue('Строка ' + k.ToString);

  // Показываем содержимое очереди:
  ShowCollection(MyQueue, 'Начальное состояние очереди:');

  // Удаляем первый элемент очереди:
  MyQueue.Dequeue;
  ShowCollection(MyQueue, 'Метод DEQUEUE:');

  // Вставляем в очередь новый элемент:
  MyQueue.Enqueue('===Новый элемент очереди===');
  ShowCollection(MyQueue, 'Метод ENQUEUE:');
  Readln;
end.
```

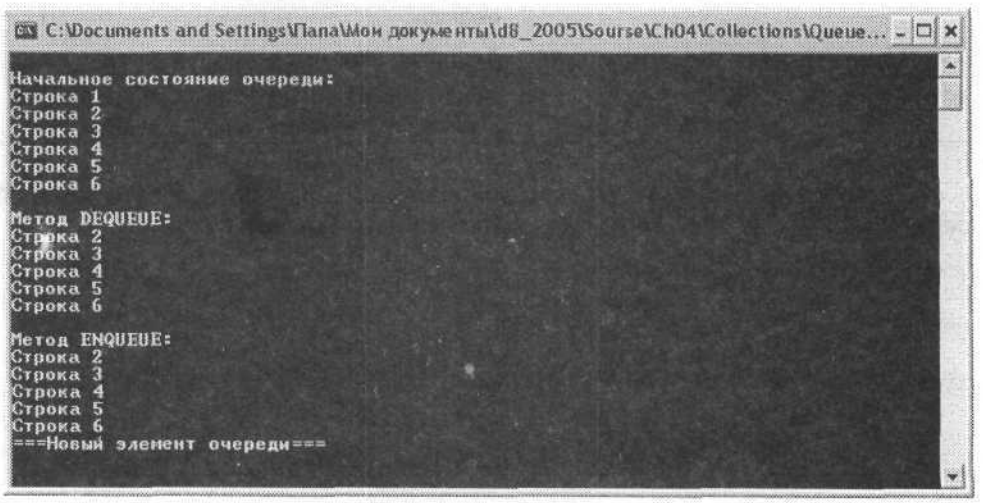


Рис. Д.2. Результат работы программы

Как видите, обработка очереди отличается от обработки стека только порядком удаления и добавления элементов.

Класс ArrayList

Класс `ArrayList` предназначен для хранения объектов произвольного типа (в том числе и `ArrayList`, то есть допускается создание многомерных хранилищ данных). В отличие от рассматривавшихся ранее классов `Stack` и `Queue`, в нем нет четко оговоренного способа пополнения/удаления коллекции. Вместе с тем

он имеет многочисленные методы для сортировки элементов, их поиска и работы с диапазонами. Он исполняет интерфейсы IList, ICollection. IEnumerabe и IClonable.

В табл. Д.8 перечислены свойства и методы класса.

Таблица Д.8. Свойства и методы класса ArrayList

Свойство, метод	Назначение
`property Capacity: Integer;`	Определяет текущую емкость коллекции
`function Adapter(List: IList): ArrayList;`	Создает на основе списка коллекцию ArrayList
`procedure AddRange(C: ICollection);`	Добавляет в конец ArrayList коллекцию C
`function BinarySearch(Obj: Object): Integer;`	Осуществляет двоичный поиск элемента в отсортированной коллекции и возвращает индекс элемента или отрицательное число, если элемент не найден
`function FixedSize(A: ArrayList): ArrayList;`	Возвращает коллекцию фиксированного размера, элементы которой можно изменять, но нельзя добавлять или удалять
`function GetRange(Start, Count: Integer): ArrayList;`	Возвращает диапазон элементов
`procedure InsertRange(Ar: ArrayList);`	Вставляет диапазон элементов
`function LastIndexOf(Obj: Object): Integer;`	Возвращает индекс последнего вхождения элемента в коллекцию
`function ReadOnly(AL: ArrayList): ArrayList;`	Возвращает коллекцию только для чтения
`procedure Remove(Obj: Object);`	Удаляет элемент из коллекции
`procedure RemoveIt(Index: Integer);`	Удаляет элемент с заданным индексом
`procedure RemoveRange(Index, Count: Integer);`	Удаляет диапазон элементов
`fubction Repeat(Obj: Object; Count: Integer): ArrayList;`	Создает коллекцию, в которой элемент Obj повторяется Count раз
`procedure Reverse;`	Изменяет порядок следования элементов на обратный
`procedure SetRange(Index: Integer; C: Collection);`	Вставляет коллекцию C, начиная с индекса Index
`procedure Sort;`	Сортирует коллекцию
`procedure TrimToSize;`	Устанавливает емкость коллекции равной количеству ее элементов

Класс ArrayList иллюстрирует следующая программа, результат работы которой показан на рис. Д.3 (проект Add_5\Collections\ArrayListDemo.dpr):

```
program ArrayListDemo;

{$APPTYPE CONSOLE}

uses
  System.Collections;

procedure ShowCollection(Coll: ICollection; Rem: String);
var
  Book: String;
begin
  Writeln;
  Writeln(Rem);
  for Book in Coll do
    Writeln(Book);
end;

var
  Books, SubList, AddList: ArrayList;
begin
  // Создаем и наполняем коллекцию:
  Books := ArrayList.Create;
  Books.Add('Д.Бокс, К.Селлз. Основы платформы .NET, том 1.');
  Books.Add('А.Гарнаев. Visual Studio .NET 2003');
  Books.Add('Э.Троелсен. C# и платформа .NET');
  Books.Add('К.Пачеко. Delphi for .NET');
  Books.Add('С.Уолтер. ASP.NET');
  ShowCollection(Books, 'Исходная коллекция:');

  // Сортируем коллекцию:
  Books.Sort;
  ShowCollection(Books, 'Метод SORT:');

  // Сортируем в обратном порядке:
  Books.Reverse;
  ShowCollection(Books, 'Метод REVERSE:');

  // Выделяем диапазон:
  SubList := Books.GetRange(2, 3);
  ShowCollection(SubList, 'Метод GETRANGE:');

  // Создаем и вставляем диапазон:
  AddList := ArrayList.Create(2);
  AddList.Add('Н.Секунов. Разработка приложений на C++ и C#');
  AddList.Add('М.Кэнту. Delphi 7 для профессионалов');
  Books.AddRange(AddList);
  ShowCollection(Books, 'Метод ADDRANGE:');
```

```
  // Иллюстрация метода TrimToSize:
  Writeln;
  Writeln('Емкость перед TRIMTOSIZE:' + Books.Capacity.ToString);
  Books.TrimToSize;
  Writeln('Емкость после TRIMTOSIZE:' + Books.Capacity.ToString);
  Readln;
end.
```

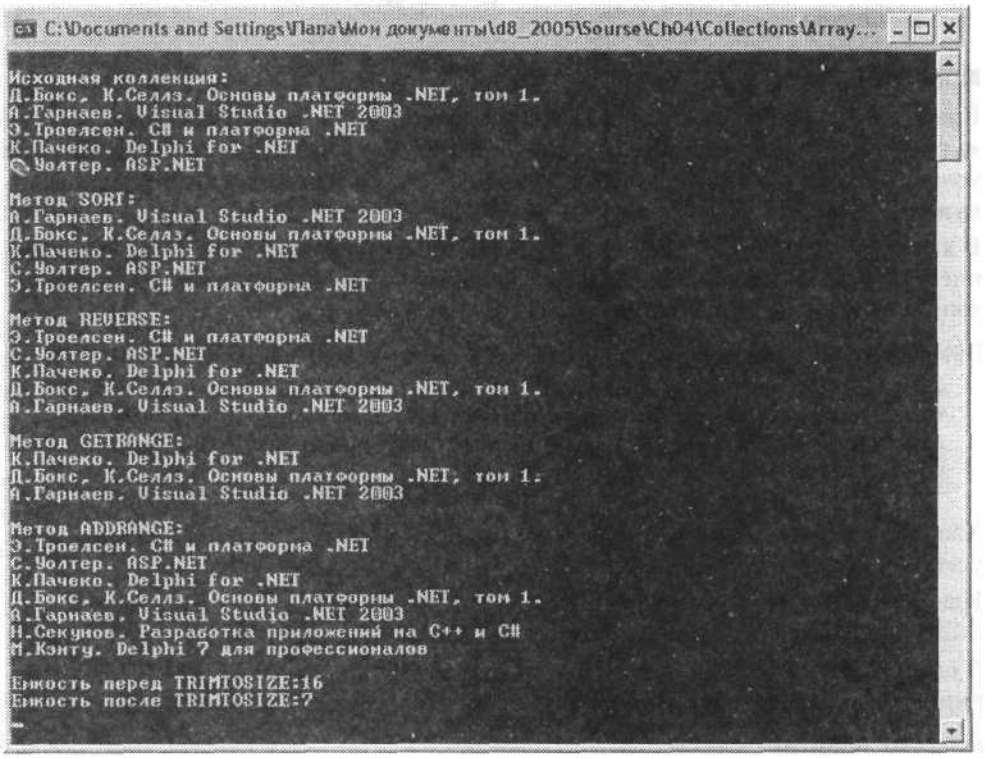


Рис. Д.3. Иллюстрация работы с коллекцией ArrayList

Замечу, что по умолчанию начальная емкость коллекции `ArrayList` составляет 16 элементов. При расширении коллекции ее емкость удваивается, составляя 16, 32, 64 и т. д. элементов. Метод `TrimToSize` отсекает не используемые в настоящий момент элементы.

Метод `BinarySearch` реализует алгоритм двоичного поиска элемента и, по идее, работает значительно быстрее, чем поиск перебором. При сравнении по умолчанию учитывается разница в регистре букв. Чтобы не учитывать разницу, следует воспользоваться перекрытой версией метода, в которой вторым параметром обращения задается метод сравнения:

```
var
  AL: ArrayList;
  k: Integer;
```

```
.....
  AL.Sort;
  k := AL.BinarySearch('АлФаВиТ', CaseInsensitiveComparer.Create);
```

Для работы метода коллекцию необходимо отсортировать, так как двоичный поиск основан на том, что элементы располагаются по возрастанию. Если в коллекции есть несколько экземпляров искомого элемента, метод не гарантирует выбор первого или последнего элемента и возвращает просто индекс одного из одинаковых элементов.

Класс HashTable

Класс `HashTable` предназначен для создания коллекций, состоящих из пар «ключ— значение». Все ключи должны быть уникальными — это непременное условие. За счет уникальности ключей коллекция позволяет быстро отыскать нужный элемент.

В качестве ключа может использоваться любой набор символов. Перед помещением ключа в коллекцию он *хэшируется*, то есть к нему применяется один из механизмов получения хэш-кода.

ПРИМЕЧАНИЕ

Хэш-код — это зашифрованный ключ, характерной особенностью которого является то, что по этому коду невозможно восстановить исходный ключ. Другая особенность — даже малейшее изменение ключа (например, пропуск одной из сотен букв) порождает совершенно другой хэш-код.

Хэш-коды и связанные с ними значения хранятся в разных сегментах памяти, что ускоряет поиск нужных элементов.

Класс исполняет интерфейсы `IDectionary`, `ICollection`, `IEnumerable`, `IClonable` и `ISerilizable`. Последний обеспечивает хранение и чтение данных.

В следующей программе, результат работы которой показан на рис. Д.4, иллюстрируется использование коллекции (проект Add_5\Collections\HashTableDemo.dpr):

```
program HashTableDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils, System.Collections;

type
  StateCapitalInfo = class // Класс для хранения
    Name: String;          // имен столиц и
    Foundation: Integer;   // дат их основания
    constructor Create(aName: String; aFound: Integer); override;
  end;

constructor StateCapitalInfo.Create(aName: String; aFound: Integer);
begin
  inherited Create;
```

```
  Name := aName;
  Foundation := aFound;
end;

var
  HT: HashTable;

procedure PrintCapitalInfo(Key: String);
// Выводит элемент данных по его ключу
var
  CI: StateCapitalInfo;
begin
  CI := HT[Key] as StateCapitalInfo;
  Writeln('Столица: ', CI.Name, '    Дата основания: ',
CI.Foundation);
end;

begin
  HT := HashTable.Create;  // Создаем и наполняем коллекцию
  HT.Add('RF', StateCapitalInfo.Create('Москва', 1047));
  HT.Add('US', StateCapitalInfo.Create('Вашингтон', 1791));
  HT.Add('UK', StateCapitalInfo.Create('Лондон', 700));
  HT.Add('DN', StateCapitalInfo.Create('Берлин', 731));

  // Выводим элементы по их ключам:
  PrintCapitalInfo('RF');
  PrintCapitalInfo('US');
  PrintCapitalInfo('UK');
  PrintCapitalInfo('DN');
  Readln;
end.
```

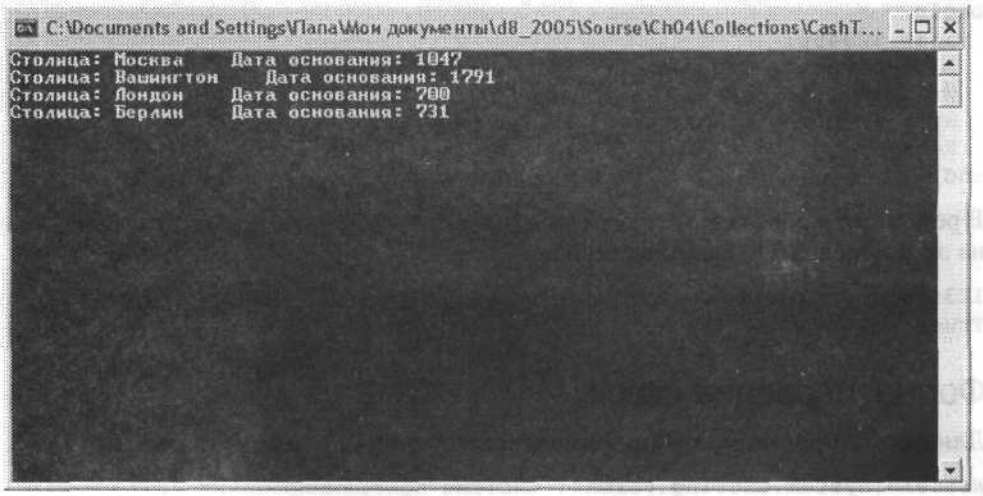


Рис. Д.4. Результат работы программы

Обработка строк

Типы **String** и System.&String в Delphi 2005 содержат символы Unicode. В этом, собственно, и заключается главное различие типов **String** в Delphi 2005 и предыдущих версиях. Однако тип System.&String от **String** отличается существенно. В этом разделе описываются эти отличия.

Преобразование значений других типов в строку и обратно

Несмотря на внешнюю схожесть типов **String** (VCL) и System.&String (CTS), они имеют совершенно разные функции преобразования. В предыдущих версиях Delphi для преобразования значения какого-либо типа в строку использовались многочисленные функции типа *XXX*ToStr (IntToStr, FloatToStr, TimeToStr и т. п.). Обратное преобразование осуществлялось симметричными функциями StrTo*XXX*. В CTS базовый тип System.&Object и, таким образом, все порожденные от него классы имеют метод ToString, преобразующий значение типа в строку. Все значимые типы объявлены в виде записей (структур), имеющих методы ToString и Parse. Наличие этих методов предельно упрощает преобразование типов в строку и обратно:

```
program ConvertTypesDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  i: Integer;
  b: Boolean;
begin
  i := 123456;
  b := b.Parse('True');
  Writeln(i.toString);
  Writeln(b.Tostring);
  Readln;
end.
```

В результате прогона этой программы (проект Add_5\Strings\ConvertTypesDemo.dpr) на экране появятся две такие строки:

```
123456
True
```

Форматирование строк

Для форматирования строки применяется функция Format:

```
Writeln(System.&String.Format('Система программирования {0}',
                              'Delphi 2005'));
```

В функции сначала указывается форматирующая строка, имеющая от одного до трех заполнителей ({0}, {1}, {2}), за ней через запятую располагается до трех заменителей[1]. Порядок следования цифр в заполнителях должен соответствовать порядку следования заменителей:

```
Writeln(System.&String.Format('{2} {1} {0}', 'ноль', 'один', 'два'));
```

Будет выведено:

```
два  один  ноль
```

Для каждого заполнителя можно указать количество символов, которое должен занимать соответствующий заменитель. Если длина поля вывода больше длины заменителя, последний выравнивается вправо, если меньше, длина поля игнорируется. Если длина отрицательная, заменитель выравнивается влево. Длина поля указывается в фигурных скобках заполнителя через запятую справа от его номера:

```
Writeln(System.&String.Format('{0, 5}{1, -4}', 'ноль', 'один'));
```

Вывод:

```
нольодин
```

Форматирование чисел

Платформа .NET Framework допускает использование спецификаторов для форматирования чисел. Эти спецификаторы можно указывать в методе `ToString`. Спецификатор представляет собой латинскую букву (строчную или прописную — не имеет значения).

В табл. Д.9 перечислены спецификаторы для форматирования чисел.

Таблица Д.9. Спецификаторы для форматирования чисел

Символ	Влияние на формат
E e	Представляет число в экспоненциальном формате, например 3,2 e+003
F f	Представляет число с фиксированной точкой (запятой), например 3200,1
G g	Представляет число в десятичном или экспоненциальном виде в зависимости от размера
N n	Представляет число с разделителями тысяч, например 3 200,00
C c	Представляет число в денежном формате

Следующая программа, результат работы которой показан на рис. Д.5, иллюстрирует форматирование чисел (проект Add_5\Strings\NumericFormatsDemo.dpr):

```
program NumericFormatsDemo;

{$APPTYPE CONSOLE}
```

[1] Точнее, заполнителей может быть сколько угодно, лишь бы цифры в них принадлежали диапазону 0...2, а вот заменителей может быть не более трех.

```
uses
  SysUtils;
var
  D: Double;
begin
  D := 3141592.654;
  WriteLn('Формат E: ', D.ToString('e'));
  WriteLn('Формат F: ', D.ToString('f'));
  WriteLn('Формат G: ', D.ToString('g'));
  WriteLn('Формат N: ', D.ToString('n'));
  WriteLn('Формат C: ', D.ToString('c'));
  Readln;
end.
```

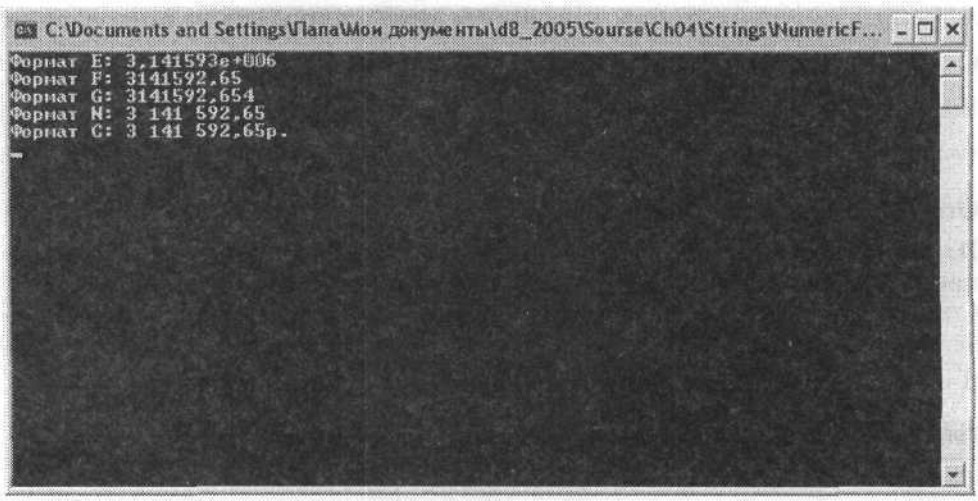


Рис. Д.5. Форматирование чисел

Замечу, что форматы N и C по умолчанию представляют дробную часть числа округленной до двух цифр. Если после спецификатора указать целое неотрицательное число, дробная часть будет округляться до указанного количества цифр:

```
Writeln(D.ToString('n6')); // Вывод 3 141 592,654000
Writeln(D.TSstring('c0')); // Вывод 3 141 593р.
```

Форматирование даты и времени

Спецификаторы для форматирования даты и времени представлены в табл. Д.10.

Таблица Д.10. Спецификаторы для форматирования даты и времени

Спецификатор	Влияние на формат
d	Дата в коротком формате
D	Дата в расширенном формате

Спецификатор	Влияние на формат
f	Дата в расширенном формате и время в коротком формате
F	Дата и время в расширенном формате
g	Дата и время в коротком формате
G	Дата в коротком формате и время в расширенном формате
M m	Месяц и число
R r	Дата и время в формате RIC 1123
s	Дата и время в формате ICO 8601
t	Время в коротком формате
T	Время в расширенном формате
u	Дата и время в формате ICO 8601 для всемирного времени
U	Всемирные дата и время
Y y	Месяц и год

Следующая программа, результат работы которой показан на рис. Д.6, иллюстрирует форматирование даты и времени (проект Add_5\Strings\DateTimeFormatesDemo.dpr):

```
program DataTimeFormatesDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  N: System.DateTime;
```

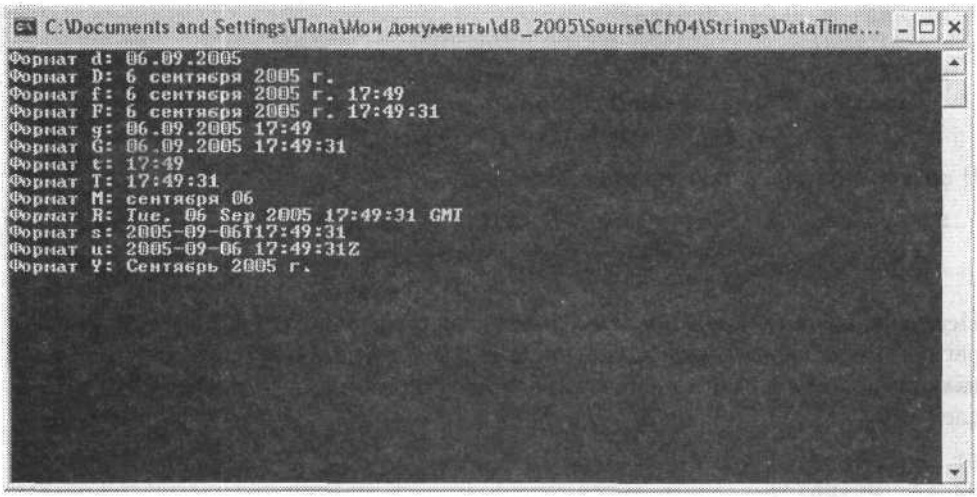


Рис. Д.6. Форматирование даты и времени

```
begin
  N := Now;
  Writeln('Формат d: ', N.ToString('d'));
  Writeln('Формат D: ', N.ToString('D'));
  Writeln('Формат f: ', N.ToString('f'));
  Writeln('Формат F: ', N.ToString('F'));
  Writeln('Формат g: ', N.ToString('g'));
  Writeln('Формат G: ', N.ToString('G'));
  Writeln('Формат t: ', N.ToString('t'));
  Writeln('Формат T: ', N.ToString('T'));
  Writeln('Формат M: ', N.ToString('m'));
  Writeln('Формат R: ', N.ToString('r'));
  Writeln('Формат s: ', N.ToString('s'));
  Writeln('Формат u: ', N.ToString('u'));
  Writeln('Формат Y: ', N.ToString('y'));
  Readln;
end.
```

Использование методов и свойств строк

Класс `System.&String` имеет богатый набор свойств и методов, которые могут использоваться для работы со строками.

ПРИМЕЧАНИЕ

Индексация символов в классе `System.&String` начинается с нуля.

Сравнение строк

Наиболее простой способ сравнения двух строк — использование логического оператора равно (=):

```
var
  S1, S2: System.&String;
.....
  if S1 = S2 then
    Writeln('Строки равны') else
    Writeln('Строки не равны');
```

В классе определен также метод `Compare`:

```
  if System.&String.Compare(S1, S1) = 0 then
    Writeln('Строки равны') else
    Writeln('Строки не равны');
```

Несмотря на явную сложность, он имеет свои преимущества. Дело в том, что в перегруженном варианте он может дополняться логическим выражением, указывающим, следует ли игнорировать возможную разницу в регистре букв (`True` — следует):

```
S1 := 'DeLpHi';
S2 := 'Delphi';
if System.&String.Compare(S1, S2, True) = 0 then
```

```
  Writeln('Строки равны') else  // Этот оператор сработает
  Writeln('Строки не равны');   // Этот оператор будет пропущен
```

В еще одном перегруженном варианте метод принимает четвертый параметр в виде указания на язык (`CultureInfo`), на котором написаны строки.

Во всех случаях метод возвращает 0, если строки идентичны, –1, если первая строка в алфавитном порядке меньше второй, и +1, если больше.

Удаление символов из строки

Для удаления символов используются методы `Trim`, `TrimStart`, `TrimEnd` и `Remove`. Методы `Trim`, `TrimStart` и `TrimEnd` удаляют пробелы, причем метод `Trim` — в начале и конце строки, метод `TrimStart` — только в начале, а метод `TrimEnd` — только в конце. Помимо пробелов удаляются также символы табуляции или любые другие непечатаемые символы (для которых метод `Char.IsWhiteSpace` возвращает `True`). В некоторых случаях могут пригодиться перегруженные варианты этих методов, в которых разрешается указывать, какие символы нужно удалять:

```
program TrimDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  ArrChr: array of Char;
begin
  SetLength(ArrChr, 4);
  ArrChr[1] := '1';
  ArrChr[2] := '2';
  Writeln('123456789'.Trim(ArrChr)); // Вывод: 3456789
  Readln;
end.
```

Метод `Remove` принимает два целых параметра — индекс первого удаляемого символа (индексация начинается с 0) и количество удаляемых символов:

```
Writeln('123456789'.Remove(3, 2)); // Вывод: 1236789
```

Поиск символов в строке

Для поиска символов предусмотрен метод `IndexOf`:

```
Writeln('1234'.IndexOf('1'));  // Вывод: 0
```

Как и в случае методов `Trim`*XXX*, можно указать массив искомых символов, который рассматривается в этом случае как подстрока:

```
uses
  SysUtils;
var
  ArrChr: array of Char;
```

```
begin
  SetLength(ArrChr, 4);
  ArrChr[1] := '1';
  ArrChr[2] := '0';
  Writeln('123456789'.IndexOf(ArrChr)); // Вывод: -1
  Readln;
end.
```

Этот же результат дает и такой поиск:

```
Wrieln('123456789'.IndexOf('10'));
```

Изменение строки

Метод `Replace` заменяет часть строки:

```
Writeln('1234'.Replace('34', '56')); // Вывод: 1256
```

Метод `Insert` вставляет часть строки. Первым параметром обращения к методу указывается индекс, который должен получить первый вставляемый символ. Если этот индекс превышает длину строки, возбуждается исключение:

```
Writeln('1234'.Insert(2, '56'));    // Вывод: 125634
Writeln('1234'.Insert(5, '56'));    // Ошибка! Исключение
```

Метод `SubString` извлекает из строки подстроку. Он принимает два параметра: индекс первого символа и количество символов подстроки. Если подстрока выходит за границы строки, возбуждается исключение:

```
Writeln('1234'.SubString(1, 2));  //Вывод: 23
```

Для изменения регистра букв строки используются методы `ToUpper` и `ToLower`:

```
Writeln('pascal'.ToUpper);  // Вывод: PASCAL
```

Разделение и объединение строк

Для разделения строки на подстроки используется метод `Split`. Этот метод в качестве входного параметра принимает массив символов, которые будут определять границы подстрок, и возвращает массив подстрок. Например (проект Add_5\Strings\SplitDemo.dpr):

```
program SplitDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  ArrStr: array of System.&String;
  S: System.&String;
  ArrChr: array of Char;
begin
```

```
 S := ' -:';                    // Строка с ограничителями подстрок
 ArrChr := S.ToCharArray;       // Преобразуем ее в массив символов
 S := 'Система программирования:Delphi-2005'; // Строка для расщепления
 ArrStr := S.Split(ArrChr); // Получаем массив подстрок
 for S in ArrStr do
   Writeln(S);                  // Печатаем подстроки
 Readln;
end.
```

Результат работы этой программы показан на рис. Д.7.

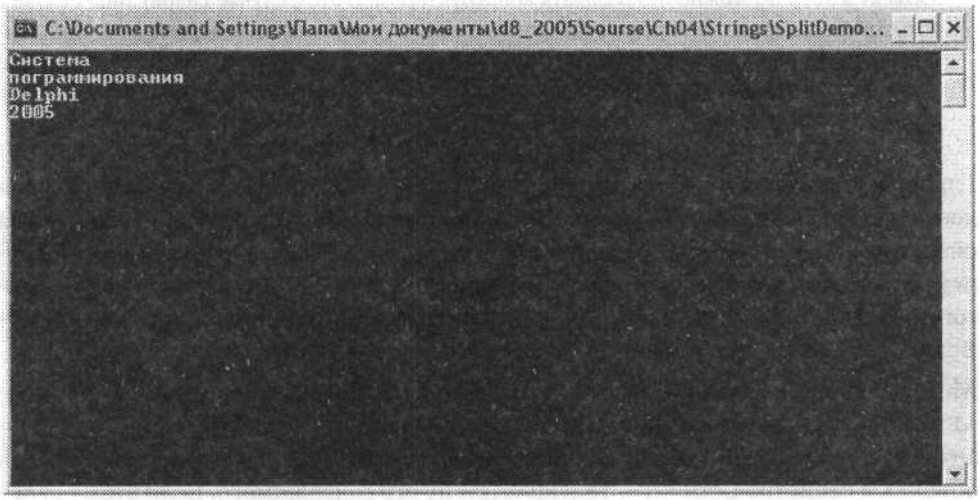


Рис. Д.7. Результат работы программы

Метод `ToCharArray` преобразует строку в массив символов Unicode.

Объединение строк реализует метод `Join`. Он принимает два параметра: символ, который будет отделять объединяемые строки друг от друга, и массив строк.

Вставим в конце предыдущего примера такой оператор:

```
Writeln(System.&String.Join(' ', ArrStr));
```

Будет выведена такая строка:

```
Система программирования Delphi 2005
```

Класс StringBuilder

Класс `StingBuilder` (пространство имен `System.Text`) представляет собой модификацию класса `&String` для работы со строками большого размера — он более эффективно использует память. Во всем остальном он во многом похож на обычную строку и легко преобразуется к ней методом `ToString`. При обращении к этому методу можно указать начальный индекс и длину подстроки содержимого `StringBuilder`.

По умолчанию объекту класса выделяется память для хранения 16 символов. Если содержимое превышает начальный размер, этот объем удваивается. Начальный размер можно указать в конструкторе объекта:

```
var
  StrBldr: StringBuilder;
.....
  StrBldr := StringBuilder.Create('Указание начальной длины', 10000);
```

ПРИМЕЧАНИЕ

Объект `&String` можно не создавать конструктором. В этом случае Delphi 2005 считает, что строка содержит неопределенное значение **`NIL`** (в предыдущих версиях неинициализированная строка **`String`** эквивалентна пустой строке). Объект `StringBuilder` всегда создается конструктором. Обращение к неинициализированному объекту `StringBuilder` вызывает исключение.

Главное различие объектов `&String` и `StingBuilder` заключается в способе доступа к содержимому объектов. Для размещения строки в первом используется оператор присваивания, в то время как в `StringBuilder` для этого предусмотрен метод `Append`. Чтобы передать содержимое строки строковому свойству другого объекта, снова требуется присваивание. Для аналогичных целей в объекте `StringBuilder` имеется метод `ToString`.

Методы `Insert`, `Remove`, `Replace` имеют такое же назначение, как и в классе `&String`. Для доступа к отдельным символам объекта `StringBuilder` предназначено его свойство `Chars`. Свойство `Capacity` определяет текущую емкость объекта, а свойство `Length` — текущую длину его содержимого. Если этому свойству присвоить 0, содержимое объекта будет уничтожено, однако его текущая емкость останется прежней. Методом `AppendFormat` к содержимому объекта можно добавить форматированную строку.

Работа с файловой системой

В приложениях .NET Framework часто возникают две в общем случае схожие задачи: сохранить (прочитать) содержимое данных (файла) и сохранить (прочитать) текущее состояние объекта в файле или таблице базы данных. Несмотря на несомненную схожесть указанных задач, в рамках .NET Framework для решения каждой из них предусмотрены свои классы. Классы обработки файлов сосредоточены в пространстве имен `System.IO`. Работа с этими классами рассматривается в этом разделе. Процесс сохранения текущего состояния объекта в памяти или на носителе информации называется *сериализацией* объекта, а обратный процесс — *десериализацией*. Эти процессы управляются классами, находящимися в пространствах имен `System.Runtime.Serialization`, которые рассматриваются в следующем разделе.

Классы для работы с файловой системой

Работа с файловой системой подразумевает обработку как папок (каталогов), так и зарегистрированных в них файлов. Соответствующие классы определены в пространстве имен `System.IO`.

В табл. Д.11 перечислены классы, использующиеся для работы с файловой системой.

Таблица Д.11. Классы для работы с файловой системой

Класс	Назначение
`Directory`	Содержит статические методы для создания и использования папок. Эти методы можно вызывать без создания соответствующих объектов
`DirectoryInfo`	Содержит свойства и методы для создания и использования папок. Доступ к членам класса возможен только после создания его экземпляра
`File`	Содержит статические методы для работы с файлами. Эти методы можно вызывать без создания объекта
`FileInfo`	Содержит методы для работы с классами, которые становятся доступными после создания объекта этого класса
`Path`	Вспомогательный класс для работы с маршрутами доступа (путями)

Статические классы `Directory` и `File` не требуют создания соответствующих объектов. Однако при обращении к их методам запускается система проверки безопасности доступа к коду (Code Access System, CAS), что замедляет работу этих методов. В связи с этим при обработке нескольких папок (файлов) удобнее использовать классы `DirectoryInfo` и `FileInfo`.

В 32-разрядных версиях Windows, как известно, для передачи данных между различными устройствами (оперативной памятью, дисковой памятью, сетью) используется концепция *потоков данных* (stream). Классы для работы с потоками данных также определены в пространстве имен `System.IO` и представлены в табл. Д.12.

Таблица Д.12. Классы для работы с потоками данных

Класс	Назначение
`MemoryStream`	Хранилище данных в оперативной памяти
`NetworkStream`	Позволяет передавать поток по сетевому соединению
`FileStream`	Представляет собой базовый класс потока для записи и чтения файлов
`BinaryReader`	Считывает данные из двоичных файлов
`BinaryWriter`	Записывает данные в двоичный файл
`StreamReader`	Считывает данные из текстовых файлов
`StreamWriter`	Записывает данные в текстовый файл

Создание и уничтожение папок

Создание и уничтожение папки проще всего осуществлять методами CreateDirectory и Delete класса Directory. Например:

```
const
  DirPath = 'c:\D2005_proba';
begin
  if not Directory.Exists(DirPath) then
    Directory.CreateDirectory(DirPath);  // Создаем папку
  if Directory.Exists(DirPath) then
    DirDirectory.Delete(DirPath, True);  // Уничтожаем папку
end.
```

Строка DirPath может содержать вложенные папки. Если они не существуют, то при выполнении метода CreateDirectory они будут созданы.

Метод Delete удаляет пустую папку. Если нужно уничтожить также пустые вложенные папки, ему передается параметр True.

Использовать методы класса DirectoryInfo для тех же целей менее удобно:

```
program DirCreate;

{$APPTYPE CONSOLE}

uses
  SysUtils, System.IO;
var
  Dir: DirectoryInfo;
begin
  Dir := DirectoryInfo.Create('c:\D2005_proba');
  if not Dir.Exists then
    Dir.&Create;
  if Dir.Exists then
    Dir.Delete(True);
  Readln;
end.
```

При обращении к конструктору DirectoryInfo.Create ему передается маршрут доступа к создаваемой папке, но сама папка не создается. Лишь после проверки факта отсутствия папки она создается методом Create.

ПРИМЕЧАНИЕ

Обратите внимание — именем Create по принятой в Delphi практике называются конструкторы объектов (в CTS имя конструктора совпадает с именем класса). В классе DirectoryInfo именем Create называется метод. Чтобы компилятор Delphi обратился к методу, а не к конструктору, перед именем метода Create стоит знак амперсанта (&).

Проверка существования папки необходима, так как попытка создать заново существующую папку или уничтожить несуществующую вызовет исключение.

При обращении к методу Delete ему передается логический параметр, указывающий на необходимость уничтожения вложенных папок.

Перемещение и копирование папок

Для перемещения папки используется метод Move класса Directory или метод MoveTo класса DirectoryInfo. В качестве параметров первому передаются имена обеих папок, а второму — только имя папки назначения:

```
Directory.Move('FromMove', 'ToMove');
...
Dir := DirectoryInfo.Create('FromMove');
Dir.&Create;
Dir.MoveTo('ToMove');
```

Фактически методы Move и MoveTo не перемещают папку, а лишь переименовывают ее, причем обе папки должны находиться в одном и том же разделе одного и того же диска. После выполнения метода начальная папка исчезает, если ее имя отличается от имени папки назначения. Методы обычно используют для добавления вложенных папок в уже существующие. В этом случае имя папки назначения повторяет имя исходной папки и расширяет ее именами вложенных папок:

```
Directory.Move('FromMove', 'FromMove\SubFolder1\SubFolder2');
```

На практике значительно чаще возникает проблема физического изменения положения папки, например переноса ее на другой дисковый носитель. Для этого необходимо скопировать содержимое исходной папки в папку назначения. Я не буду детально рассматривать эту в общем случае не тривиальную задачу. Скажу лишь, что она решается рекурсивным вызовом некоторой процедуры, которой передаются имена обеих папок. В процедуре просматривается содержимое исходной папки, и ее файлы копируются в папку назначения. Если среди файлов встречается вложенная папка, процедура вызывает сама себя (это и называется рекурсивным вызовом), но уже с другими именами папок.

При копировании нужно предусмотреть возможность отсутствия папки назначения, существование в ней копируемого файла и пр. Поскольку все это выходит за рамки обсуждаемой темы, в следующем примере возможность существования такого рода сопутствующих проблем игнорируется. Показанная далее программа (проект Add_5\Files\CopyDirectory.dpr) копирует файлы из одной поставляемой с Delphi 2005 папки в другую. В выбранной для копирования папке отсутствуют вложенные папки, что исключает необходимость рекурсии. Копируемые файлы заменяют одноименные файлы, существующие в папке назначения:

```
program CopyDirectory;

{$APPTYPE CONSOLE}

uses
  SysUtils,
```

```
  System.IO;
var
  Files: Array of String;
  k: Integer;
  FromFile, ToFile: String;
  FileAttr: FileAttributes;
const
  FromDir = 'c:\Program Files\Common Files\Borland ' +
            'Shared\Images\GlyFX\Small';
  ToDir = 'c:\D2005_Images\';
begin
  // Создаем папку назначения:
  if not Directory.Exists(ToDir) then
    Directory.CreateDirectory(ToDir);
  // Получаем содержимое исходной папки:
  Files := Directory.GetFileSystemEntries(FromDir);
  // Цикл копирования файлов:
  for k := Low(Files) to High(Files) do
  begin
    // Имя очередного файла или вложенной папки:
    FromFile := Files[k];
    // Получаем атрибуты файла (папки):
    FileAttr := &File.GetAttributes(FromFile);
    // Блокируем рекурсивную обработку вложенной папки:
    if (FileAttr and FileAttributes.Directory) <>
                              FileAttributes.Directory then
    begin
      ToFile := ToDir + Path.GetFileName(FromFile);
      Writeln(ToFile);
      &File.Copy(FromFile, ToFile, True);
    end;
  end;
  Writeln('Всего файлов: ', High(Files) + 1);
  Readln;
end.
```

Напомню, что практически во всех современных операционных системах (и в Windows в том числе) в файловой системе используются так называемые таблицы размещения файлов (File Allocation Tables, FAT). В этих таблицах указываются файлы корневого каталога и вложенные папки верхнего уровня. В FAT каждой вложенной папки, в свою очередь, указываются файлы и вложенные папки. В каждой строке FAT помимо имени файла (папки) указываются также дополнительные сведения — начальный кластер расположения файла (папки), дата создания и, в том числе, так называемые файловые атрибуты, которые задаются перечислением `System.IO.FileAttributes`. Значения этого перечисления указаны в табл. Д.13.

Таблица Д.13. Значения перечисления FileAttributes

Значение	Описание
`Archived`	Обычный файл, доступный для копирования и удаления
`Compressed`	Сжатый файл
`Device`	Зарезервировано для будущего использования
`Directory`	Вложенная папка
`Encrypted`	Файл зашифрован (если речь идет о файле) или все включенные в папку файлы зашифрованы (если речь идет о папке)
`Hidden`	Скрытый файл
`Normal`	Обычный файл, для которого не могут быть установлены никакие другие атрибуты
`NotContentIndexed`	Файл не может быть индексирован службой индексации контекста операционной системы
`Offline`	Файл не подключен, и его данные в настоящее время недоступны
`ReadOnly`	Файл только для чтения
`ReparsePoint`	Файл содержит реперную точку, то есть блок данных, ассоциированных с другими файлом или папкой
`SparseFile`	«Рыхлый» файл, бóльшая часть которого содержит нули
`System`	Системный файл, который может использовать только операционная система
`Temporary`	Временный файл

Хотя в справочной литературе `FileAttributes` называется перечислением (*enumeration*), с позиций Delphi это «перечисление» является множеством, то есть файловым атрибутом может быть комбинация перечисленных значений. Используемая в примере проверка блокирует обработку любой папки:

```
if (FileAttr and FileAttributes.Directory) <>
                    FileAttributes.Directory then
```

Исследование информации о записи FAT

Информация о единичной строке FAT хранится в свойствах класса `FileSystemInfo`. В табл. Д.14 перечислены эти свойства.

Таблица Д.14. Свойства класса FileSystemInfo

Свойство	Описание
`Attributes`	Атрибуты
`CreationTime`	Время создания файла (папки)
`CreationTimeUTC`	Время создания в формате UTC (Universal Time Coordinated — всемирное координированное время). Время в формате UTC соответствует времени по гринвичскому меридиану и координируется с показаниями атомных часов

продолжение

Таблица Д.14 (продолжение)

Свойство	Описание
Exists	Признак существования
Extension	Расширение файла
FullName	Полное имя (с маршрутом доступа)
LastAccessTime	Время последнего обращения к файлу (папке)
LastAccessTimeUTC	Время последнего обращения в формате UTC
LastWriteTime	Время последнего изменения
LastWriteTimeUTC	Время последнего изменения в формате UTC
Name	Имя файла (папки)

Классы DirectoryInfo и FileInfo являются наследниками базового класса FileSystemInfo и получают все его свойства.

Следующий пример (проект Add_5\Files\GetDirInfo.dpr) иллюстрирует использование класса DirectoryInfo:

```
program GetDirInfo;

{$APPTYPE CONSOLE}

uses
  SysUtils, System.IO;
const
  i = 30;  // Выравнивание правого края поясняющей надписи
var
  DirInfo: DirectoryInfo;
begin
  DirInfo := DirectoryInfo.Create('C:\Program Files');
  Writeln('Полное имя:':i, DirInfo.FullName);
  Writeln('Имя:':i, DirInfo.Name);
  Writeln('Время создания:':i, DirInfo.CreationTime.ToString);
  Writeln('Время создания UTC:':i,
DirInfo.CreationTimeUtc.Tostring);
  Writeln('Признак существования:':i, DirInfo.Exists.ToString);
  Writeln('Расширение:':i, DirInfo.Extension);
  Writeln('Время последнего доступа:':i,
                          DirInfo.LastAccessTime.ToString);
  Writeln('Время последнего обновления:':i,
                          DirInfo.LastWriteTime.ToString);
  Writeln('Атрибуты:':i, Enum(DirInfo.Attributes).ToString);
  Readln;
end.
```

Результат работы программы показан на рис. Д.8.

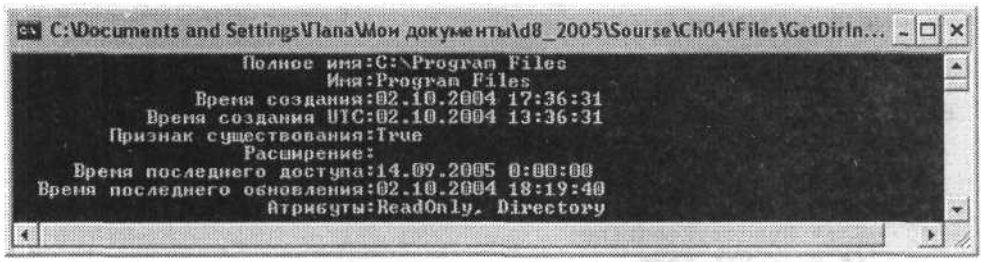


Рис. Д.8. Результат работы программы

Запись и чтение файлов

В этом разделе рассматривается обработка файлов на основе классов CTS.

Обработка текстовых файлов

При работе с текстовыми файлами сначала создается поток класса `FileSteram`. Данные записываются в файл с помощью вспомогательного объекта класса `StreamWriter`, а читаются объектом класса `StreamReader`. Например (проект Add_5\Files\StringFileDemo.dpr):

```
program StringFileDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils, System.IO;

var
  Stream: FileStream;
  Writer: StreamWriter;
  Reader: StreamReader;
  S: String;
begin
  // Запись в текстовый файл
  Stream := FileStream.Create('strings.dat',
                    FileMode.Create, FileAccess.Write);
  try
    Writer := StreamWriter.Create(Stream);
    try
      Writer.WriteLine('Мело, мело по всей Земле,');
      Writer.WriteLine('Во все пределы.');
      Writer.WriteLine('Свеча горела на столе.');
      Writer.WriteLine('Свеча горела...');
    finally
      Writer.Close;
    end;
  finally
    Stream.Close;
  end;
```

```
// Чтение из текстового файла
Stream := FileStream.Create('strings.dat',
                  FileMode.Open, FileAccess.Read);
try
  Reader := StreamReader.Create(Stream);
  try
    repeat
      S := Reader.ReadLine;
      if S <> NIL then
        Writeln(S);
    until S = NIL;
  finally
    Reader.Close;
  end;
finally
  Stream.Close;
end;
Readln
end.
```

При создании объекта `FileStream` ему передаются имя файла и два параметра: `FileMode`, определяющий способ создания потока, и `FileAccess`, регулирующий доступ потока к данным. В табл. Д.15 и Д.16 указываются допустимые значения этих параметров.

Таблица Д.15. Значения параметра FileMode

Значение	Описание
`Append`	Добавляет записи в существующий файл или создает новый. Требует, чтобы параметр `FileMode` имел значение `Write`
`Create`	Создает новый файл или переписывает существующий. Требует, чтобы параметр `FileMode` имел значение `Write`
`CreateNew`	Создает новый файл, а если он уже существует, возникает исключение. Требует, чтобы параметр `FileMode` имел значение `Write`
`Open`	Открывает существующий файл. Если файла нет, возникает исключение
`OpenOrCreate`	Открывает существующий или создает новый файл, если он еще не создан
`Truncate`	Открывает существующий файл и делает его размер равным нулю

Таблица Д.16. Значения параметра FileAccess

Значение	Описание
`Read`	Поток может читать данные
`ReadWrite`	Поток может читать и записывать данные
`Write`	Поток может записывать данные

Для потока существует понятие текущей записи — в эту запись помещаются данные, и из нее они считываются. Положением текущей записи можно управлять с помощью метода Seek, имеющего такую сигнатуру:

```
function Seek(Offset: Long; Origin: SeekOrigin): Long;
```

Здесь Offset — смещение относительно позиции, указанной параметром Origin. Перечисление SeekOrigin может иметь значение, указанное в табл. Д.17.

Таким образом, указанный ниже вызов сделает текущей запись с индексом 10 от начала потока (индексация начинается с 0):

```
Stream.Seek(10, SeekOrigin.Begin);
```

Таблица Д.17. Значения перечисления SeekOrigin

Значение	Описание
Begin	Соответствует началу потока
Current	Соответствует текущей записи потока
End	Соответствует концу потока

Физическая запись данных в файл реализуется в момент закрытия потока методом Close или выталкивания записей из промежуточного буфера методом Flush. Во втором случае поток не закрывается и готов к продолжению операций.

При чтении строк из текстового файла нужно контролировать конец файла. Для этого переменная типа **String** в Delphi 2005 может принимать значение **NIL**, если из файла ничего не прочитано, то есть если файл исчерпан (см. показанный ранее пример). Другим способом контроля является обращение к методу StreamReader.Peek, который возвращает положительное число, если файл не исчерпан, или −1 — в противном случае. Таким образом, цикл чтения записей из текстового файла предельно упрощается:

```
while Reader.Peek <> -1 do
  Writeln(Reader.ReadLine);
```

Обработка двоичных данных

Обработка двоичных файлов во многом подобна обработке текстовых: сначала создается поток, затем — объекты BinaryWriter или BinaryReader в зависимости от направления передачи данных (в файл или из файла).

Объект BinaryWrite имеет метод Write, с помощью которого данные передаются в файл. Метод Write имеет множество перегруженных вариантов, позволяющих записывать в файл любые данные (в том числе строки **String**, так что деление файлов на строковые и двоичные носит чисто условный характер).

Объект BinaryReader имеет метод Read, предназначенный для чтения из потока массива символов или байтов. Кроме того, он имеет многочисленные методы

ReadXXXX (ReadBoolean, ReadByte, ReadDouble, ReadString и т. д.) для чтения значений примитивных типов. Его метод PeekChar позволяет контролировать конец файла: подобно рассмотренному ранее методу StreamReader.Peek, он возвращает –1, если файл исчерпан.

Сериализация

Как уже говорилось в начале предыдущего раздела, .NET Framework поддерживает интересную технологию сохранения текущего состояния объекта на некотором носителе информации. Эта технология называется *сериализацией объекта*. Сериализованный объект может быть передан по сети на другой компьютер и там восстановлен в первоначальном состоянии — этот процесс называется *десериализацией*.

Техника сериализации

Особенностью сериализации является то, что ее можно применять только к классам, поддерживающим эту процедуру. Для этого класс должен объявляться с атрибутом [Serializable] или в его объявлении должно быть явное указание на то, что он исполняет интерфейс ISerializable. Например:

```
type
  [Serializable]
  Book = class(System.Object)
  public
    FName: String;
    FAuthor: String;
    FPublish: String;
    FYear: Integer;
  end;
```

Лишь относительно небольшое количество (менее 40) стандартных классов CTS реализуют интерфейс ISerializable. Это в основном классы, используемые как хранилища данных (DataSet, DataTable, TreeNode и некоторые другие). В интерфейсе ISerializable определен единственный метод GetObectData такого вида:

```
procedure GetObjectData(Info: SerializationInfo;
                        Context: StreamingContext);
```

Параметр Info содержит данные, а Context — текущее состояние потока данных. На практике специальные классы, реализующие интерфейс ISerializable, создаются для обслуживания нестандартных типов данных. В большинстве случаев этого можно не делать — достаточно лишь вставить атрибут [Serializable] перед объявлением класса, как это сделано в предыдущем примере.

При сериализации среда CLR создает граф объекта и с помощью этого графа сохраняет объект вместе со связанными с ним объектами. Каждый связанный с ос-

новным объектом объект получает уникальный идентификатор. При десериализации может встретиться случай, когда объект ссылается на идентификатор еще не восстановленного объекта. В этом случае CLR корректирует порядок восстановления объектов.

Объявив тем или иным способом, что класс поддерживает сериализацию, мы должны выбрать формат представления его объекта. Технология .NET Framework предлагает для этой цели два формата: двоичный и SOAP (Simple Object Access Protocol — простой протокол доступа к объекту). В первом случае создается объект класса `BinaryFormatter`, принадлежащего пространству имен `System.Runtime.Serialization.Formatters.Binary`. Во втором — объект класса `SoapFormatter` (пространство имен `System.Runtime.Serialization.Formatters.Soap`).

ПРИМЕЧАНИЕ

Класс `BinaryFormatter` находится в сборке mscorlib.dll. Чтобы воспользоваться классом, достаточно его пространство имен поместить в предложение uses. Класс `SoapFormatter` включен в сборку System.Runtime.Serialization.Formatters.SOAP.dll. Чтобы воспользоваться классом, перед вставкой имени его пространства имен в предложение **uses** следует добавить в проект ссылку, в которой описывается местонахождение библиотеки. Эта ссылка добавляется с помощью диалогового окна, открываемого командой Project ▸ AddReference.

Пример

В этом разделе представлен простой пример, иллюстрирующий описанную технологию (проект Add_5\Files\SerialDemo.dpr):

```
program SerialDemo;

{$APPTYPE CONSOLE}

// Следующая строка вставлена в проект с помощью диалогового
// окна, открываемого командой Project>AddReference.
// Она не должна разрываться на части:
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework
\v1.1.4322\System.Runtime.Serialization.Formatters.Soap.dll')

uses
  SysUtils,
  System.IO,
  System.Runtime.Serialization.Formatters.Soap;

type
  [Serializable]
  TBook = class(System.Object) // Класс для сериализации
  public
    FName: String;
```

```
    FAuthor: String;
    FPublish: String;
    FYear: Integer;
    constructor Create(Name, Author, Publish: String; Year: Integer);
  end;
{ TBook }
  TBooks = array[1..3] of TBook;  // Массив для трех объектов Book
constructor TBook.Create(Name, Author, Publish: String; Year: Integer);
begin
  inherited Create;
  FName := Name;
  FAuthor := Author;
  FPublish := Publish;
  FYear := Year;
end;
var
  Book: TBook;
  Stream: FileStream;
  Fmtr: SOAPFormatter;
  Books: TBooks;
  k: Integer;
begin
  // Создаем три книги и помещаем их в массив:
  Book := TBook.Create('C# и платформа .NET', 'Троелсен Э.',
                       'Питер', 2005);
  Books[1] := Book;
  Book := TBook.Create('ASP.NET. Искусство содания web-сайтов',
                       'Уолтер С.', 'Диа-Софт', 2002);
  Books[2] := Book;
  Book := TBook.Create('Грибной царь', 'Поляков Ю.', 'РОСМЭН',
2005);
  Books[3] := Book;
  // Создаем поток:
  Stream := FileStream.Create('books.dat', FileMode.Create,
                                           FileAccess.Write);
  // Создаем объект класса SoapFormatter:
  Fmtr := SoapFormatter.Create;
 // Сериализация массива:
  Fmtr.Serialize(Stream, Books);
  Stream.Close;

  // Создаем поток для десериализации:
  Stream := FileStream.Create('books.dat', FileMode.Open,
                                           FileAccess.Read);
  // Десериализация массива:
  Books := Fmtr.Deserialize(Stream) as TBooks;
  // Печать восстановленных объектов:
  for k := 1 to 3 do
```

```
  begin
    Writeln('Автор: ':15, Books[k].FAuthor);
    Writeln('Название: ':15, Books[k].FName);
    Writeln('Изд-во: ':15, Books[k].FPublish);
    Writeln('Год: ':15, Books[k].FYear.ToString);
    Writeln;
  end;
  Readln;
end.
```

При десериализации используется явное приведение получаемого массива объектов к типу Books:

```
Books := Fmtr.Deserialize(Stream) as TBooks;
```

Если этого не сделать, компилятор сообщит о несовместимости типов TBooks и Object: метод Deserialize всегда возвращает данные как объекты класса Object.

Алфавитный указатель

A—Z

ADO.NET, 57
ASP.NET, 455, 468
BDE, 31
BDE.NET, 117
dbExpress.NET, 242
dbGo.NET, 211
IBX.NET, 268
ODBC, 31
SQL, 515
URL, 360

A

авторизация пользователя, 455
агрегатные функции, 520
архитектура
  двухзвенная, 20
  однозвенная, 20
  трехзвенная, 20
аутентификация
  Windows, 455
  на основе форм, 455
  по паспорту, 455
  пользователя, 455

Б

база данных
  нормализация, 25
  определение, 20
база данных (*продолжение*)
  проектирование, 25
  псевдоним, 44
  реляционная, 23
  сеанс связи, 139
  система управления, 20
базовый объект, 222
баннер, 374
бизнес-правила, 54
блокировка таблиц, 165
браузер, 358

В

веб-приложение, 358
веб-сервер, 357
  модули расширения, 358
  обработка запроса, 361
  сетевое имя, 360
веб-служба, 365, 436
веб-форма, 362, 369
визуализация данных
  в BDE.NET, 185, 197
  свойство DataBindings, 58
внешний ключ, 484
выборка
  в BDE.NET, 174
  из связанных таблиц, 515
  простая, 514

Г

генератор, 484, 495
гиперссылка, 397, 528
графическая кнопка, 375
группировка записей, 520

Д

данные
  абстрактные, 122
  двоично-десятичные, 121
  отображение, 182
  привязка, 410
двунаправленный курсор, 242, 259
десериализация, 580
документ
  правильный, 544
  состоятельный, 544
домен, 496
драйвер
  ODBC, 213
  языковый, 35, 39

З

закладка
  в BDE.NET, 159
  в dbGo.NET, 237
запись
  визуализация, 201
  вставка, 524
  выборка, 174
  группировка, 520
  поиск, 159, 173
  программный доступ, 155
  редактирование, 524
  сортировка, 517
  текущая, 157
  удаление, 173, 524
  фильтрация, 163
запрос
  выборки, 514
  изменяемый, 182
  параметрический, 71, 180
  создание, 514
зарезервированное слово
  AND, 517
  BLOB, 523
зарезервированное слово (*продолжение*)
  BOOLEAN, 523
  BY, 517
  CHAR, 523
  CHARACTER, 523
  CREATE, 523
  DATE, 523
  DELETE, 525
  DROP, 523
  FLOAT, 523
  FROM, 515, 525
  GROUP, 521
  IN, 519
  INDEX, 523
  INSERT, 524
  INT, 523
  INTEGER, 523
  INTO, 524
  KEY, 523
  LIKE, 518
  NOT, 517, 523
  NULL, 523
  ON, 523
  OR, 517
  ORDER, 517
  PRIMARY, 523
  SELECT, 515
  SET, 525
  TABLE, 523
  UNIQUE, 523
  UPDATE, 525
  VALUES, 524
  WHERE, 516
знак равенства, 529
значащий тип данных, 554

И

изоляция транзакций, 146
индекс, 24
  вторичный, 33
  добавление, 171
  смена, 171
  создание, 523
  составной, 34, 172
  текущий, 171
  удаление, 171, 523

интерфейс
IDataReader, 73
IDbCommand, 69
IDbConnection, 66
IDbDataAdapter, 75
ISQLMetadata, 68
интранет, 355
источник данных
в ASP.NET, 412, 415
в BDE.NET, 117, 185
в dbGo.NET, 212

К

календарь, 386
каскадные изменения, 166
класс
ArrayList, 566
BdpCommand, 69
Binding, 110
Connection, 66
DataAdapter, 75
DataColumn, 86, 87
DataGrid, 99
методы, 104
свойства, 99
DataRow, 95
методы, 97
свойства, 95
DataSet, 57, 80, 407
методы, 80
свойства, 80
DataTable, 83
методы, 83
свойства, 83
HashTable, 570
HttpServerUtility, 470
Page, 369, 468
Queue, 565
Stack, 562
StringBuilder, 579
TADOCommand, 232
TADOConnection, 217
TADODataSet, 239
TBlobField
методы, 138
свойства, 137
TColumn, 189
класс (*продолжение*)
TCustomADODataSet
методы, 236
события, 238
TDatabase
методы, 146
свойства, 145
TDataSource
свойства, 185
события, 187
TDBCheckBox, 203
TDBComboBox, 203
TDBCtrlGrid, 206
методы, 208
свойства, 207
события, 209
TDBDataSet, 149
методы, 152
свойства, 150
события, 154, 166
TDBEdit, 202
TDBGrid
методы, 190
свойства, 187
события, 191
TDBGridColumns, 189
TDBListBox, 203
TDBLookupComboBox, 203
TDBLookupListBox, 203
TDBMemo, 205
TDBNavigator, 209
TDBRadioGroup, 203
TDBRichEdit, 205
TDBText, 202
TField, 118
методы, 124
свойства, 118
события, 124
TIBBackupRestoreService, 504
TIBBase
методы, 272
свойства, 271
события, 272
TIBControlAndQueryService, 504
TIBControlService, 503
TIBCustomService, 503
TMemoField, 134
TParam, 178

класс (*продолжение*)
TParams
методы, 177
свойства, 177
TQuery
методы, 178
свойства, 176
TSession, 139
TSQLQuery, 257
TSQLTable, 257
TSQLTimeStamp, 137
TStringField, 134
TTable
методы, 168
свойства, 167
прокси, 443
разъединенный, 58
соединенный, 58
клиент, 242
клиентское место, 22
ключ
внешний, 484
первичный, 23, 484
кнопка
графическая, 375
обычная, 375
с гиперссылкой, 375
коллекция, 559
объектов, 75
ограничений, 83
параметров запроса, 69
полей таблицы, 83
таблиц, 59
комментарий, 520, 532
компонент
CalcController, 353
CalcText, 352
CalcTotal, 352
DataText, 328
HTML, 362
TADOConnection, 224
TADOQuery, 241
TADOTable, 240
TADQStoredProc, 300
TDatabase, 47
TDataSource, 48
TDQLClientDataSet, 259
TIBBackupService, 506
компонент (*продолжение*)
TIBConfigService, 505
TIBDatabase
методы, 274
свойства, 272
события, 275
TIBDatabaseInfo, 287
TIBDataSet, 281
методы, 284
свойства, 282
события, 285
TIBInstall, 511
TIBLicensingService, 510
TIBLogService, 509
TIBQuery, 281
TIBRestoreService, 507
TIBSecurityService, 509
TIBServerProperties, 510
TIBSQL, 285
методы, 286
свойства, 285
события, 287
TIBSQLMonitor, 289
TIBStatisticalService, 508
TIBStoredProc, 301
TIBTable, 280
TIBTransaction
методы, 277
свойства, 276
TIBUnInstall, 513
TQuery, 175
TRvCustomConnection, 339
TRvDataSetConnection, 336
TRvQueryConnection, 336
TRvTableConnection, 336
TSession
использование, 141
методы, 140
свойства, 139
события, 141
TSimpleDataSet
методы, 261
свойства, 259
события, 267
TSQLConnection, 248
методы, 250
свойства, 248
события, 254

компонент (*продолжение*)
TSQLDataSet, 254
методы, 256
свойства, 254
события, 257
TSQLMonitor, 257
методы, 258
свойства, 258
события, 259
TSQLStoredProc, 301
TStoredProc, 297
TTable, 48
адаптер, 57
визуализации, 57, 105
связи, 57, 117
списочный, 203
управления сервером, 362
корневой элемент, 542
курсор
двунаправленный, 242
однонаправленный, 242
кэширование, 235

M

маска EditMask, 123
массив, 554
метаданные, 289
метод
Compute, 86
ExecuteNonQuery, 70
ExecuteReader, 73
ExecuteScalar, 72
Fill, 77
Format, 73
GET, 359
HEAD, 359
POST, 359, 440
PUT, 359
Update, 57, 79

H

набор данных, 48, 75
в ADO.NET, 57, 80, 83, 407
в BDE.NET, 117, 149
в dbGo.NET, 234
закрытие, 155
курсор, 157
набор данных (*продолжение*)
навигация, 157
наполнение, 65
настройка, 63
открытие, 155
редактирование, 156
фильтрация, 163

O

обратная сортировка, 184
объект
Command, 223
Connection, 222
Error, 223
Field, 223
Page, 468
Parameter, 223
Property, 223
Recordset, 223
базовый, 222
данных, 335, 336
Data Lookup Security Control, 341
Database Connection, 337
Direct Data View, 336
Driver Data View, 337
Simple Security Control, 341
поле, 50, 125
стилевой, 102
столбец, 52, 191
обычная кнопка, 375
ограничение, 484, 497
однонаправленный курсор, 242
оператор
DO, 558
EXCEPTION, 296
EXECUTE PROCEDURE, 295
EXIT, 295
FOR, 294, 558
IF, 294
SELECT, 294
SUSPEND, 294
WHILE, 295, 558
отчет
библиотека, 333
главный–детальный, 344
группирующий, 349

отчет (*продолжение*)
защита, 341
машина генерации, 323
печать, 334
проект, 323
простой, 342
с агрегатными функциями, 351
список, 334
страница, 334
умалчиваемый, 334
экспорт, 353

П

пакет сообщения, 359
первичный ключ, 484
переключатель, 379
перехват сообщений, 257
подстановочное поле, 127
подтип, 494
поиск записей
в таблице, 173
неточный, 174
точный, 173
поле
BLOB, 137, 485, 494
вещественное, 136
вычисляемое, 51
в ADO.NET, 88
в BDE.NET, 128
даты и времени, 137, 491
конструктор, 51
логическое, 136
массив, 490
мемо, 134
обращение к значению
в ADO.NET, 112
в BDE.NET, 129
объект, 125
подстановочное, 50, 51, 434
в ADO.NET, 91
в BDE.NET, 127
проверка значения, 132
пустое, 129
редактор, 50
совместимость типов, 130, 496
строковое, 134, 492
текстовое представление, 133
поле (*продолжение*)
фиксированно-десятичное, 491
целочисленное, 134
пользователь
авторизация, 455
аутентификация, 455
роль, 455
учетная запись, 455
пользовательская функция, 485
пользовательский элемент управления, 447
право доступа к данным, 220
представление, 485
необновляемое, 307
обновляемое, 307
определение, 306
удаление, 307
привязка, 106
примитивный тип данных, 554
провайдер
BDP, 61
OLE DB, 212, 219
выбор, 219
данных, 60
настройка, 220
состояний, 480
проверка данных
клиентская, 398
серверная, 398
прокси-класс, 443
протокол
HTTP, 358
SOAP, 440
процедура
кодирование, 293
объявление, 304
параметры, 291
создание, 291
псевдоним, 44, 520
локальный, 47
создание, 44, 45

Р

редактор
полей в BDE.NET, 125
столбцов, 191, 192
роль пользователя, 455

С

свойство
ConnectionString, 67
DataBindings, 105
связь
настройка, 214
реляционная, 64
с объектом ADO, 216
таблиц, 216
сеанс
как объект, 480
связи с базой данных, 139
сервер
InterBase, 248, 484, 488
MS SQL Server, 225
Oracle, 248
базы данных, 21, 22
приложений, 21
серверный элемент управления, 372
сериализация, 580, 589
сетка, 187
символ
Unicode, 30
национального алфавита, 30
словообразующий, 402
специальный, 527
условный, 35
сортировка
записей, 517
обратная, 184
составной индекс, 172
состояние вида, 371, 478
список, 383
значений столбца, 194
полей таблицы, 50
типов, 32
ссылочная целостность, 38
ссылочный тип данных, 554
столбец
массив, 490
объект, 191
пустой, 194
редактор, 52
сетки, 52
список значений, 194
страница
активная серверная, 361
глобальная, 334
отчета, 334
строка связи
структура, 217
формирование, 217
СУБД
архитектура, 20
заказная, 20
клиент-серверная, 21
локальная, 21
масштабирование, 22
промышленная, 20, 25
распределенная, 21
сетевая, 21
специализированная, 20
специального назначения, 20
типы, 20
универсальная, 20
файл-серверная, 21
сценарий, 368, 373, 540

Т

таблица, 535
блокировка, 165
виртуальная, 306
главная, 23
детальная, 23
дочерняя, 23
запись, 23
индекс, 523
подстановки, 36
поиск записей, 173
поле, 23
псевдоним, 520
родительская, 23
связи, 23
создание, 523
столбец, 23
строка, 23
удаление, 173, 523
эксклюзивный доступ, 173
тег, 526
!-, 532
B, 529
BODY, 527

тег (*продолжение*)
BR, 527
CAPTION, 536
CODE, 529
DFN, 529
DIV, 539
EM, 529
FONT, 529
FORM, 533
FRAME, 538
FRAMESET, 538
HTML, 526
I, 529
IFRAME, 539
IMG, 531
INPUT, 533
LI, 529
OL, 529
P, 527
STRONG, 529
TABLE, 535
TD, 535
TR, 535
TT, 529
U, 529
UL, 529
A, 528
технология
ADO.NET, 57
ASP.NET, 372, 455, 468
BDE.NET, 117
dbExpress.NET, 242
dbGo.NET, 211
IBX, 268
OLE DB, 212
тип данных
значащий, 554
примитивный, 554
ссылочный, 554
транзакция, 47
изоляция, 55
в ADO.NET, 68
в BDE.NET, 146
в dbGo.NET, 224
определение, 55
старт, 68, 112
транзакция (*продолжение*)
управление, 55
в BDE.NET, 146
в dbExpress.NET, 252
в dbGo.NET, 224, 225
в IBX.NET, 278
триггер, 54
изменение, 306
кодирование, 293
объявление, 304
определение, 484
создание, 296, 302
удаление, 306

У

уведомление, 485
утилита DataExplorer, 61
учетная запись пользователя, 455

Ф

файл cookie, 457
файл-сервер, 21
фильтрация записей, 163
флажок, 381
форма
для визуализации данных, 201
для редактирования записей, 104
нормальная, 25
фрейм
определение, 537
плавающий, 538
положение на странице, 539
функция
alert, 535
агрегатная, 90, 520
пользовательская, 485

Х

хеш-код, 464, 570
хранимая процедура, 485
изменение, 306
кодирование, 293
объявление, 304
параметры, 291
создание, 291
удаление, 306

Ш

шаблон преобразования, 543

Э

эксклюзивный доступ к таблице, 173
элемент, 542
    AdRotator, 374
    BDWebDataSource, 428
    Calendar, 386
    CheckBox, 381
    CheckBoxList, 381
    CompareValidator, 403
    CustomValidator, 405
    DataList, 415
    DBWebGrid, 432
    DBWebNavigator, 431
    DropDownList, 383
    HyperLink, 397
    Image, 384
    Label, 382
    ListBox, 383
    Literal, 389
    Panel, 392
    PlaceHolder, 396
    RadioButton, 379
элемент (*продолжение*)
    RadioButtonList, 379
    RangeValidator, 401
    RegularExpressionValidator, 402
    Repeater, 412
    RequiredFieldValidator, 399
    Table, 393
    TextBox, 382
    ValidationSunnary, 404
    XML, 397
    корневой, 542
    пустой, 545
    управления
        пользовательский, 447
        серверный, 372

Я

язык
    DDL, 232
    HTML, 526
    SGML, 541
    Visual Basic .NET, 553
    WSDL, 440
    XML, 540
    XSL, 543

Валерий Васильевич Фаронов

Delphi 2005. Разработка приложений для баз данных и Интернета

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Адаменко*
Литературный редактор	*А. Жданов*
Художник	*Л. Адуевская*
Иллюстрации	*Л. Родионова*
Корректоры	*Н. Рощина, И. Тимофеева*
Верстка	*Р. Гришанов*

Лицензия ИД № 05784 от 07.09.01.

Подписано к печати 29.11.05. Формат 70×100/16. Усл. п. л. 49,02. Тираж 2500. Заказ 449

ООО «Питер Принт», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»

190005, Санкт-Петербург, Измайловский пр., 29

В 1997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал»**. Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал»**. Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте **www.piter.com**;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com** или по тел. **(812) 103-73-74**;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект **«Библиотека профессионала»**.

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на **«Библиотеку профессионала»**. Она для тех, кто экономит не только время, но и деньги. Покупая комплект – книжную полку **«Библиотека профессионала»**, вы получаете:

- **скидку 15%** от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов – дополнительную **скидку 3%**;
- членство в **«Клубе Профессионал»**;
- подарок – журнал **«Клуб Профессионал»**.

Закажите бесплатный журнал «Клуб Профессионал».

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- по телефону: **(812) 103-73-74;**
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: **www.piter.com;**
- по почте: **197198, Санкт-Петербург, а/я 619, ЗАО «Питер Пост».**

ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ И ОПЛАТЫ ИЗДАНИЙ:

Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой **«Книга-почтой»** в течение двух недель после получения заказа или выхода книги из печати.

Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал»

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM