Петр Дарахвелидзе
Евгений Марков

Программирование в Delphi 7

+Дискета

- Кроссплатформенное программирование
- Рекомендации по разработке приложений в стиле Windows XP
- Современные технологии доступа к данным: ADO, dbExpress, InterBase Express
- Распределенные многозвенные приложения и технология DataSnap

Петр Дарахвелидзе
Евгений Марков

Программирование в Delphi 7

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.06
ББК 32.973.26-018.2
Д20

Дарахвелидзе П. Г., Марков Е. П.

Д20 Программирование в Delphi 7. — СПб.: БХВ-Петербург, 2003. — 784 с.: ил.

ISBN 5-94157-116-X

В книге обсуждаются вопросы профессиональной разработки приложений в среде Borland Delphi 7. Приводится детальное описание объектной концепции, стандартных и программных технологий, используемых при работе программистов. Значительная часть материала посвящена разработке приложений, базирующихся на широко используемых и перспективных технологиях доступа к данным: ADO, dbExpress, InterBase Express. Достойное место отведено распределенным многозвенным приложениям и технологии DataSnap. Все рассматриваемые в этой книге темы сопровождаются подробными примерами.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Эльвира Максунова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.10.02.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 63,21.
Тираж 6000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-116-X

© Дарахвелидзе П. Г., Марков Е. П., 2003
© Оформление, издательство "БХВ-Петербург", 2003

Содержание

ЧАСТЬ I. ОБЪЕКТНАЯ КОНЦЕПЦИЯ DELPHI 7 15

Глава 1. Объектно-ориентированное программирование 16

Объект и класс 17
Поля, свойства и методы 20
События 23
Инкапсуляция 27
Наследование 27
Полиморфизм 28
Методы 30
Перегрузка методов 34
Области видимости 35
Объект изнутри 38
Резюме 42

Глава 2. Библиотека визуальных компонентов VCL и ее базовые классы 44

Иерархия базовых классов 44
Класс *TObject* 47
Класс *TPersistent* 49
Класс *TComponent* 51
Базовые классы элементов управления 53
Класс *TControl* 54
Группа свойств *Visual*. Местоположение и размер элемента управления 54
Выравнивание элемента управления 56
Внешний вид элемента управления 57
Связь с родительским элементом управления 59
Класс *TWinControl* 60
Класс *TCustomControl* 63
Класс *TGraphicControl* 63
Резюме 63

Глава 3. Обработка исключительных ситуаций 64

Исключительная ситуация как класс 64
Защитные конструкции языка Object Pascal 69
Блок *try..except* 70
Блок *try..finally* 72
Использование исключительных ситуаций 74
Протоколирование исключительных ситуаций 76
Коды ошибок в исключительных ситуациях 78
Исключительная ситуация *EAbort* 82
Функция *Assert* 82
Резюме 83

Глава 4. Кроссплатформенное программирование для Linux 84

Проект CLX 86
Объектная концепция кроссплатформенного программирования 87
Библиотека компонентов CLX 88
Сходства и различия визуальных компонентов CLX и VCL 90
Особенности программирования для Linux 92
Приложения баз данных для Linux 94
Internet-приложения для Linux 94
Резюме 95

ЧАСТЬ II. ИНТЕРФЕЙС И ЛОГИКА ПРИЛОЖЕНИЯ 97

Глава 5. Элементы управления Win32 98

Что такое библиотека ComCtl32 98
Многостраничный блокнот — компоненты *TTabControl* и *TPageControl* 100
Компонент *TToolBar* 105
Компонент *TImageList* 110
Компоненты *TTreeView* и *TListView* 112
Календарь 125
Компонент *TMonthCalendar* 126
Компонент *TDateTimePicker* 127
Панель состояния *TStatusBar* 129
Расширенный комбинированный список *TComboBoxEx* 130
Создание нового компонента на базе элементов управления из библиотеки ComCtl32 131
Резюме 141

Глава 6. Элементы управления Windows XP 142

Пользовательский интерфейс Windows XP 142
Манифест Windows XP 143
Компонент *TXPManifest* 145
Включение манифеста Windows XP в ресурсы приложения 145
Визуальные стили и темы оформления 146
Визуальные стили в Delphi 147
Theme API 149

Компоненты настройки цветовой палитры ... 151
Резюме ... 152

Глава 7. Списки и коллекции ... 153

Список строк ... 154
Класс *TStrings* ... 154
Класс *TStringList* ... 155
Список указателей ... 162
Класс *TList* ... 163
Пример использования списка указателей ... 166
Коллекции ... 170
Класс *TCollection* ... 171
Класс *TCollectionItem* ... 172
Резюме ... 173

Глава 8. Действия (Actions) и связанные с ними компоненты ... 174

Действия. Компонент *TActionList* ... 175
События, связанные с действиями ... 176
Свойства, распространяемые на клиентов действия ... 178
Прочие свойства ... 179
Стандартные действия ... 180
Категория *Edit* ... 183
Категория *Search* ... 183
Категория *Help* ... 183
Категория *File* ... 183
Категория *Dialog* ... 184
Категория *Window* ... 184
Категория *Tab* ... 184
Категория *List* ... 184
Категория *Internet* ... 185
Категория *Format* ... 187
Категория *Dataset* ... 187
Категория *Tools* ... 187
Компонент *TActionManager* ... 187
Изменение и настройка внешнего вида панелей ... 189
Ручное редактирование коллекций панелей и действий ... 191
Резюме ... 194

Глава 9. Файлы и устройства ввода/вывода ... 195

Использование файловых переменных. Типы файлов ... 195
Операции ввода/вывода ... 197
Ввод/вывод с использованием функций Windows API ... 204
Отложенный (асинхронный) ввод/вывод ... 208
Контроль ошибок ввода/вывода ... 210
Атрибуты файла. Поиск файла ... 211
Потоки ... 213
Базовые классы *TStream* и *THandleStream* ... 213
Класс *TFileStream* ... 215

Класс *TMemoryStream* ... 217
Класс *TStringStream* ... 218
Оповещение об изменениях в файловой системе ... 218
Использование отображаемых файлов ... 220
Резюме ... 223

Глава 10. Использование графики ... 224

Графические инструменты Delphi ... 224
Класс *TFont* ... 225
Класс *TPen* ... 226
Класс *TBrush* ... 227
Класс *TCanvas* ... 227
Класс *TGraphic* ... 233
Класс *TPicture* ... 235
Класс *TMetafile* ... 237
Класс *TIcon* ... 238
Класс *TBitmap* ... 238
Графический формат JPEG. Класс *TJPEGImage* ... 243
Компонент *TImage* ... 245
Использование диалогов для загрузки и сохранения графических файлов ... 247
Класс *TClipboard* ... 254
Класс *TScreen* ... 256
Вывод графики с использованием отображаемых файлов ... 259
Класс *TAnimate* ... 263
Резюме ... 264

ЧАСТЬ III. ПРИЛОЖЕНИЯ БАЗ ДАННЫХ ... 265

Глава 11. Архитектура приложений баз данных ... 266

Как работает приложение баз данных ... 268
Модуль данных ... 271
Подключение набора данных ... 272
Настройка компонента *TDataSource* ... 274
Отображение данных ... 276
Резюме ... 278

Глава 12. Набор данных ... 279

Абстрактный набор данных ... 281
Стандартные компоненты ... 286
Компонент таблицы ... 287
Компонент запроса ... 289
Компонент хранимой процедуры ... 292
Индексы в наборе данных ... 293
Механизм подключения индексов ... 294
Список описаний индексов ... 295
Описание индекса ... 295
Использование описаний индексов ... 297
Параметры запросов и хранимых процедур ... 298

Класс *TParams* ... 301
Класс *TParam* ... 302
Состояния набора данных ... 304
Резюме ... 307

Глава 13. Поля и типы данных ... 308

Объекты полей ... 309
Статические и динамические поля ... 311
Класс *TField* ... 313
Виды полей ... 317
Поля синхронного просмотра ... 317
Вычисляемые поля ... 320
Внутренние вычисляемые поля ... 321
Агрегатные поля ... 321
Объектные поля ... 322
Типы данных ... 323
Ограничения ... 328
Резюме ... 331

Глава 14. Механизмы управления данными ... 333

Связанные таблицы ... 334
Отношение "один-ко-многим" ... 334
Отношение "многие-ко-многим" ... 336
Поиск данных ... 337
Поиск по индексам ... 337
Поиск в диапазоне ... 338
Поиск по произвольным полям ... 339
Фильтры ... 340
Быстрый переход к помеченным записям ... 342
Диапазоны ... 344
Резюме ... 346

Глава 15. Компоненты отображения данных ... 347

Классификация компонентов отображения данных ... 347
Табличное представление данных ... 349
Компонент *TDBGrid* ... 349
Компонент *TDBCtrlGrid* ... 359
Навигация по набору данных ... 361
Представление отдельных полей ... 364
Компонент *TDBText* ... 364
Компонент *TDBEdit* ... 365
Компонент *TDBCheckBox* ... 365
Компонент *TDBRadioGroup* ... 366
Компонент *TDBListBox* ... 366
Компонент *TDBComboBox* ... 366
Компонент *TDBMemo* ... 367
Компонент *TDBImage* ... 367
Компонент *TDBRichEdit* ... 368

Синхронный просмотр данных ... 368
Механизм синхронного просмотра ... 369
Компонент *TDBLookupListBox* ... 372
Компонент *TDBLookupComboBox* ... 372
Графическое представление данных ... 372
Резюме ... 375

ЧАСТЬ IV. ТЕХНОЛОГИИ ДОСТУПА К ДАННЫМ ... 377

Глава 16. Процессор баз данных Borland Database Engine ... 378

Архитектура и функции BDE ... 379
Псевдонимы баз данных и настройка BDE ... 383
Интерфейс прикладного программирования BDE ... 392
Соединение с источником данных ... 401
Компоненты доступа к данным ... 406
Класс *TBDEDataSet* ... 406
Класс *TDBDataSet* ... 412
Компонент *TTable* ... 413
Компонент *TQuery* ... 419
Компонент *TStoredProc* ... 421
Резюме ... 423

Глава 17. Технология dbExpress ... 424

Драйверы доступа к данным ... 425
Соединение с сервером баз данных ... 426
Управление наборами данных ... 431
Транзакции ... 434
Использование компонентов наборов данных ... 435
Класс *TCustomSQLDataSet* ... 435
Компонент *TSQLDataSet* ... 438
Компонент *TSQLTable* ... 438
Компонент *TSQLQuery* ... 439
Компонент *TSQLStoredProc* ... 439
Компонент *TSimpleDataSet* ... 440
Способы редактирования данных ... 443
Интерфейсы dbExpress ... 447
Интерфейс *ISQLDriver* ... 447
Интерфейс *ISQLConnection* ... 448
Интерфейс *ISQLCommand* ... 449
Интерфейс *ISQLCursor* ... 450
Отладка приложений с технологией dbExpress ... 451
Распространение приложений с технологией dbExpress ... 453
Резюме ... 454

Глава 18. Сервер баз данных InterBase и компоненты InterBase Express ... 455

Механизм доступа к данным InterBase Express ... 456
Компонент *TIBDatabase* ... 456
Компонент *TIBTransaction* ... 461

Компоненты доступа к данным ... 465
Область дескрипторов *XSQLDA* ... 467
Структура *XSQLVAR* ... 468
Компонент *TIBTable* ... 469
Компонент *TIBQuery* ... 470
Компонент *TIBStoredProc* ... 471
Компонент *TIBDataSet* ... 472
Компонент *TIBSQL* ... 474
Обработка событий ... 477
Информация о состоянии базы данных ... 479
Компонент *TIBDatabaseInfo* ... 479
Компонент *TIBSQLMonitor* ... 481
Резюме ... 482

Глава 19. Использование ADO средствами Delphi ... 483

Основы ADO ... 483
Перечислители ... 486
Объекты соединения с источниками данных ... 487
Сессия ... 487
Транзакции ... 488
Наборы рядов ... 488
Команды ... 489
Провайдеры ADO ... 490
Реализация ADO в Delphi ... 491
Компоненты ADO ... 491
Механизм соединения с хранилищем данных ADO ... 492
Компонент *TADOConnection* ... 492
Настройка соединения ... 493
Управление соединением ... 498
Доступ к связанным наборам данных и командам ADO ... 501
Объект ошибок ADO ... 503
Транзакции ... 504
Наборы данных ADO ... 505
Класс *TCustomADODataSet* ... 506
Набор данных ... 506
Курсор набора данных ... 507
Локальный буфер ... 509
Состояние записи ... 510
Фильтрация ... 511
Поиск ... 512
Сортировка ... 513
Команда ADO ... 513
Групповые операции ... 515
Параметры ... 516
Класс *TParameters* ... 516
Класс *TParameter* ... 517
Компонент *TADODataSet* ... 519
Компонент *TADOTable* ... 520

Компонент *TADOQuery* ... 520
Компонент *TADOStoredProc* ... 521
Команды ADO ... 521
Объект ошибок ADO ... 523
Пример приложения ADO ... 524
Соединение с источником данных ... 527
Групповые операции ... 528
Фильтрация ... 529
Сортировка ... 529
Резюме ... 529

ЧАСТЬ V. РАСПРЕДЕЛЕННЫЕ ПРИЛОЖЕНИЯ БАЗ ДАННЫХ ... 531

Глава 20. Технология DataSnap. Механизмы удаленного доступа ... 532

Структура многозвенного приложения в Delphi ... 533
Трехзвенное приложение в Delphi ... 535
Сервер приложений ... 536
Клиентское приложение ... 538
Механизм удаленного доступа к данным DataSnap ... 538
Компонент *TDCOMConnection* ... 539
Компонент *TSocketConnection* ... 540
Компонент *TWebConnection* ... 543
Провайдеры данных ... 545
Вспомогательные компоненты — брокеры соединений ... 548
Компонент *TSimpleObjectBroker* ... 548
Компонент *TLocalConnection* ... 550
Компонент *TSharedConnection* ... 551
Компонент *TConnectionBroker* ... 551
Резюме ... 552

Глава 21. Сервер приложения ... 553

Структура сервера приложения ... 554
Интерфейс *IAppServer* ... 555
Интерфейс *IProviderSupport* ... 558
Удаленные модули данных ... 558
Удаленный модуль данных для сервера Автоматизации ... 559
Дочерние удаленные модули данных ... 563
Регистрация сервера приложения ... 564
Пример простого сервера приложения ... 565
Главный удаленный модуль данных ... 566
Дочерний удаленный модуль данных ... 567
Регистрация сервера приложения ... 568
Резюме ... 569

Глава 22. Клиент многозвенного распределенного приложения ... 570

Структура клиентского приложения ... 571
Клиентские наборы данных ... 572

Компонент *TClientDataSet* ... 574
Получение данных от компонента-провайдера ... 575
Кэширование и редактирование данных ... 577
Управление запросом на сервере ... 579
Использование индексов ... 580
Сохранение набора данных в файлах ... 582
Работа с данными типа BLOB ... 582
Представление данных в формате XML ... 583
Агрегаты ... 583
Объекты-агрегаты ... 584
Агрегатные поля ... 586
Группировка и использование индексов ... 587
Вложенные наборы данных ... 587
Дополнительные свойства полей клиентского набора данных ... 588
Обработка ошибок ... 589
Пример "тонкого" клиента ... 592
Соединение клиента с сервером приложения ... 594
Наборы данных клиентского приложения ... 595
Резюме ... 596

ЧАСТЬ VI. ГЕНЕРАТОР ОТЧЕТОВ RAVE REPORTS 5.0 ... 597

Глава 23. Компоненты Rave Reports и отчеты в приложении Delphi ... 598

Генератор отчетов Rave Reports 5.0 ... 599
Компоненты Rave Reports и их назначение ... 600
Отчет в приложении Delphi ... 601
Компонент отчета *TRvProject* ... 602
Компонент управления отчетом *TRvSystem* ... 605
Резюме ... 610

Глава 24. Визуальная среда создания отчетов ... 611

Инструментарий визуальной среды создания отчетов ... 612
Проект отчета ... 614
Библиотека отчетов ... 615
Каталог глобальных страниц ... 616
Словарь просмотров данных ... 616
Стандартные элементы оформления и их свойства ... 617
Элементы для представления текста и изображений ... 618
Графические элементы управления ... 619
Штрихкоды ... 619
Обработка событий ... 620
Внешние источники данных в отчете ... 620
Соединение с источником данных и просмотры ... 621
Безопасность доступа к данным ... 622
Отображение данных в отчетах ... 623
Структурные элементы отчета ... 623
Элементы отображения данных ... 625
Резюме ... 626

Глава 25. Разработка, просмотр и печать отчетов ... 627

Этапы создания отчета и включение его в приложение ... 628
Простой отчет в визуальной среде Rave Reports ... 628
Нумерация страниц отчета ... 629
Использование элемента *FontMaster* ... 630
Добавление страниц к отчету ... 630
Отчет в приложении ... 631
Просмотр и печать отчета ... 633
Сохранение отчета во внешнем файле ... 634
Компонент *TRvNDRWriter* ... 635
Преобразование форматов данных ... 637
Резюме ... 638

Глава 26. Отчеты для приложений баз данных ... 639

Соединения с источниками данных в Rave Reports ... 640
Соединения с источниками данных в визуальной среде Rave Reports ... 642
Соединение через драйвер Rave Reports ... 642
Соединение через компонент приложения Delphi ... 644
Соединения с источниками данных в приложении ... 645
Компонент *TRvDataSetConnection* ... 645
Компоненты, использующие BDE ... 647
Компонент *TRvCustomConnection* ... 648
Аутентификация пользователя в отчете ... 651
Типы отчетов ... 652
Простой табличный отчет ... 652
Отчет "один-ко-многим" ... 653
Группирующий отчет ... 656
Использование вычисляемых значений ... 657
Вычисляемые значения по одному источнику ... 657
Вычисляемые значения по нескольким источникам ... 659
Управляющие вычислительные элементы ... 661
Резюме ... 662

ЧАСТЬ VII. ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ ... 663

Глава 27. Стандартные технологии программирования ... 664

Интерфейс переноса Drag-and-Drop ... 664
Интерфейс присоединения Drag-and-Dock ... 669
Усовершенствованное масштабирование ... 674
Управление фокусом ... 674
Управление мышью ... 675
Ярлыки ... 680
Резюме ... 683

Глава 28. Динамические библиотеки ... 684

Проект DLL ... 685
Экспорт из DLL ... 687

Соглашения о вызовах ... 689
Директива *register* ... 689
Директива *pascal* ... 690
Директива *stdcall* ... 690
Директива *cdecl* ... 690
Директива *safecall* ... 690
Инициализация и завершение работы DLL ... 690
Вызов DLL ... 694
Неявный вызов ... 694
Явный вызов ... 696
Ресурсы в DLL ... 699
Использование модуля *ShareMem* ... 703
Резюме ... 703

Глава 29. Потоки и процессы ... 704

Обзор потоков ... 705
Потоки и процессы ... 706
Фоновые процедуры, или способ обойтись без потоков ... 707
Приоритеты потоков ... 707
Класс *TThread* ... 711
Пример создания многопоточного приложения в Delphi ... 715
Проблемы при синхронизации потоков ... 719
Тупики ... 719
Гонки ... 720
Средства синхронизации потоков ... 721
Событие ... 722
Взаимные исключения ... 724
Семафор ... 724
Критическая секция ... 724
Процесс. Порождение дочернего процесса ... 725
Поток ... 727
Консольный ввод ... 727
Оповещение об изменении в файловой системе ... 727
Локальные данные потока ... 729
Как избежать одновременного запуска двух копий одного приложения ... 729
Резюме ... 730

Глава 30. Многомерное представление данных ... 732

Понятие кросстаба ... 732
Взаимосвязь компонентов многомерного представления данных ... 733
Подготовка набора данных ... 735
Компонент *TDecisionQuery* ... 739
Компонент *TDecisionCube* ... 739
Компонент *TDecisionSource* ... 743
Отображение данных ... 744
Компонент *TDecisionGrid* ... 744
Компонент *TDecisionGraph* ... 747

Управление данными 747
Компонент *TDecisionPivot* 748
Пример многомерного представления данных 749
Резюме 751

Глава 31. Использование возможностей Shell API 752

Понятие пространства имен 752
Размещение значка приложения на *System Tray* 753
Интерфейс *IShellLink* 758
Интерфейс *IShellFolder* 760
Добавление пунктов в системное контекстное меню 767
Резюме 771

Приложение. Описание дискеты 773

Предметный указатель 776

ЧАСТЬ I

Объектная концепция Delphi 7

Глава 1. Объектно-ориентированное программирование

Глава 2. Библиотека визуальных компонентов VCL и ее базовые классы

Глава 3. Обработка исключительных ситуаций

Глава 4. Кроссплатформенное программирование для Linux

ГЛАВА 1

Объектно-ориентированное программирование

Несколько лет назад книгу по Delphi 2 или 3 надо было начинать с азов объектно-ориентированного программирования (ООП). Многие только переходили к Delphi из DOS, многие использовали Borland Pascal for Windows и работали с Windows API напрямую. Объекты еще были в диковинку, и полное разъяснение новых принципов было просто обязательно.

Но и сейчас писать об этом вполне актуально. Конечно, выросло поколение программистов, которые "с молоком матери" впитали новые понятия. Но от понимания объектов до их грамотного использования — дистанция огромного размера. Для создания более или менее сложных приложений нужны навыки объектно-ориентированного дизайна, а для приложений в свою очередь — четкое знание возможностей вашей среды программирования. Поэтому в данной главе мы постараемся акцентировать внимание читателя на применение ООП в среде Delphi 7.

По сравнению с традиционными способами программирования ООП обладает рядом преимуществ. Главное из них заключается в том, что эта концепция в наибольшей степени соответствует внутренней логике функционирования операционной системы (ОС) Windows. Программа, состоящая из отдельных объектов, отлично приспособлена к реагированию на события, происходящие в ОС. К другим преимуществам ООП можно отнести большую надежность кода и возможность повторного использования отработанных объектов.

В этой главе рассматриваются способы реализации основных механизмов ООП в Object Pascal и Delphi:

- ❒ понятия объекта, класса и компонента;
- ❒ основные механизмы ООП: инкапсуляция, наследование и полиморфизм;
- ❒ особенности реализации объектов;
- ❒ взаимодействие свойств и методов.

Материал главы рассчитан на читателя, имеющего представление о самом языке Object Pascal, его операторах и основных возможностях.

Объект и класс

Перед началом работы необходимо ввести основные понятия и определения.

Классом в Object Pascal называется структура языка, которая может иметь в своем составе переменные, функции и процедуры. Переменные в зависимости от предназначения именуются *полями* или *свойствами (см. ниже)*. Процедуры и функции класса — *методами*. Соответствующий классу тип будем называть *объектным типом*:

```
type
  TMyObject = class(TObject)
    MyField: Integer;
    function MyMethod: Integer;
end;
```

В этом примере описан класс `TMyObject`, содержащий поле `MyField` и метод `MyMethod`.

Поля объекта аналогичны полям записи (record). Это данные, уникальные для каждого созданного в программе экземпляра класса. Описанный здесь класс `TMyObject` имеет одно поле — `MyField`.

Методы — это процедуры и функции, описанные внутри класса и предназначенные для операций над его полями. В состав класса входит указатель на специальную таблицу, где содержится вся информация, нужная для вызова методов. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут поля именно того объекта, который вызвал метод. Внутри метода указатель на вызвавший его объект доступен под зарезервированным именем `Self`.

Понятие свойства будет подробно рассмотрено ниже. Пока можно определить его как поле, доступное для чтения и записи не напрямую, а через соответствующие методы.

Классы могут быть описаны либо в секции интерфейса модуля, либо на верхнем уровне вложенности секции реализации. Не допускается описание классов "где попало", т. е. внутри процедур и других блоков кода.

Разрешено опережающее объявление классов, как в следующем примере:

```
type
  TFirstObject = class;
  TSecondObject = class(TObject)
```

```
  F1st : TFirstObject;
 ...
  end;
  TFirstObject = class(TObject)
 ...
  end;
```

Чтобы использовать класс в программе, нужно, как минимум, объявить переменную этого типа. Переменная объектного типа называется *экземпляром класса* или *объектом*:

```
var
  AMyObject: TMyObject;
```

До введения термина "класс" в языке Pascal существовала двусмысленность определения "объект", который мог обозначать и тип, и переменную этого типа. Теперь же существует четкая граница: класс — это описание, объект — то, что создано в соответствии с этим описанием.

Как создаются и уничтожаются объекты?

Те, кто раньше использовал ООП в работе на C++ и особенно в Turbo Pascal, будьте внимательны: в Object Pascal экземпляры объектов могут быть только динамическими. Это означает, что в приведенном выше фрагменте переменная `AMyObject` на самом деле является указателем, содержащим адрес объекта.

Объект "появляется на свет" в результате вызова специального метода, который инициализирует объект — *конструктора*. Созданный экземпляр уничтожается другим методом — *деструктором*:

```
AMyObject := TMyObject.Create;
{ действия с созданным объектом }
...
AMyObject.Destroy;
```

Но, скажет внимательный читатель, ведь объекта еще нет, как мы можем вызывать его методы? Справедливое замечание. Однако обратите внимание, что вызывается метод `TMyObject.Create`, а не `AMyObject.Create`. Есть такие методы (в том числе конструктор), которые успешно работают до (или даже *без*) создания объекта. О подобных методах, называемых *методами класса*, пойдет речь чуть ниже.

В Object Pascal конструкторов у класса может быть несколько. Общепринято называть конструктор `Create` (в отличие от Turbo Pascal, где конструктор обычно назывался `Init`, и от C++, где его имя совпадает с именем класса). Типичное название деструктора — `Destroy`.

```
type
  TMyObject = class(TObject)
    MyField: Integer;
```

```
    Constructor Create;
    Destructor Destroy;
    Function MyMethod: Integer;
end;
```

Для уничтожения экземпляра объекта рекомендуется использовать метод `Free`, который первоначально проверяет указатель (не равен ли он `Nil`) и только затем вызывает `Destroy`:

```
AMyObject.Free;
```

До передачи управления телу конструктора происходит собственно создание объекта — под него отводится память, значения всех полей обнуляются. Далее выполняется код конструктора, написанный программистом для инициализации экземпляров данного класса. Таким образом, хотя на первый взгляд синтаксис конструктора схож с вызовом процедуры (не определено возвращаемое значение), но на самом деле конструктор — это функция, возвращающая созданный и инициализированный объект.

Примечание

Конструктор создает новый объект только в том случае, если перед его именем указано имя класса. Если указать имя уже существующего объекта, он поведет себя по-другому: не создаст новый объект, а только выполнит код, содержащийся в теле конструктора.

Чтобы правильно инициализировать в создаваемом объекте поля, относящиеся к классу-предку, нужно сразу же при входе в конструктор вызвать конструктор предка при помощи зарезервированного слова `inherited`:

```
constructor TMyObject.Create;
begin
 inherited Create;
...
end;
```

Взяв любой из примеров, прилагаемых к этой книге или поставляемых вместе в Delphi, вы почти не увидите там вызовов конструкторов и деструкторов. Дело в том, что любой компонент, попавший при визуальном проектировании в ваше приложение из Палитры компонентов, включается в определенную иерархию. Иерархия эта замыкается на форме (класс `TForm`): для всех ее составных частей конструкторы и деструкторы вызываются автоматически, незримо для программиста. Кто создает и уничтожает формы? Это делает приложение (глобальный объект с именем `Application`). В файле проекта (с расширением dpr) вы можете увидеть вызовы метода `Application.CreateForm`, предназначенного для этой цели.

Что же касается объектов, создаваемых динамически (во время выполнения приложения), то здесь нужен явный вызов конструктора и метода `Free`.

Поля, свойства и методы

Поля класса являются переменными, объявленными внутри класса. Они предназначены для хранения данных во время работы экземпляра класса (объекта). Ограничений на тип полей в классе не предусмотрено. В описании класса поля должны предшествовать методам и свойствам. Обычно поля используются для обеспечения выполнения операций внутри класса.

Примечание

При объявлении имен полей принято к названию добавлять заглавную букву `F`. Например `FSomeField`.

Итак, поля предназначены для использования внутри класса. Однако класс должен каким-либо образом взаимодействовать с другими классами или программными элементами приложения. В подавляющем большинстве случаев класс должен выполнить с некоторыми данными определенные действия и представить результат.

Для получения и передачи данных в классе применяются свойства. Для объявления свойств в классе используется зарезервированное слово `property`.

Свойства представляют собой атрибуты, которые составляют индивидуальность объекта и помогают описать его. Например, обычная кнопка в окне приложения обладает такими свойствами, как цвет, размеры, положение. Для экземпляра класса "кнопка" значения этих атрибутов задаются при помощи свойств — специальных переменных, определяемых ключевым словом `property`. Цвет может задаваться свойством `Color`, размеры — свойствами `Width` и `Height` и т. д.

Так как свойство обеспечивает обмен данными с внешней средой, то для доступа к его значению используются специальные методы класса. Поэтому обычно свойство определяется тремя элементами: полем и двумя методами, которые осуществляют его чтение/запись:

```
type
 TAnObject = class(TObject)
  function GetColor: TSomeType;
  procedure SetColor(ANewValue: TSomeType);
  property AColor: TSomeType read GetColor write SetColor;
 end;
```

В данном примере доступ к значению свойства `AColor` осуществляется через вызовы методов `GetColor` и `SetColor`. Однако в обращении к этим методам в явном виде нет необходимости: достаточно написать:

```
AnObject.AColor := AValue;
AVariable := AnObject.AColor;
```

и компилятор самостоятельно оттранслирует обращение к свойству `AColor` в вызовы методов `GetColor` или `SetColor`. То есть внешне свойство выглядит в точности как обычное поле, но за всяким обращением к нему могут стоять нужные вам действия. Например, если у вас есть объект, представляющий собой квадрат на экране, и его свойству "цвет" вы присваиваете значение "белый", то произойдет немедленная перерисовка, приводящая реальный цвет на экране в соответствие со значением свойства. Выполнение этой операции осуществляется методом, который связан с установкой значения свойства "цвет".

В методах, входящих в состав свойств, может осуществляться проверка устанавливаемой величины на попадание в допустимый диапазон значений и вызов других процедур, зависящих от вносимых изменений. Если же потребности в специальных процедурах чтения и/или записи нет, можно вместо имен методов применять имена полей. Рассмотрим следующую конструкцию:

```
TPropObject = class(TObject)
    FValue: TSomeType;
    procedure DoSomething;
    function Correct(AValue: Integer):boolean;
    procedure SetValue(NewValue: Integer);
    property AValue: Integer read FValue write SetValue;
  end;
...
procedure TPropObject.SetValue(NewValue: Integer);
begin
  if (NewValue<>FValue) and Correct(NewValue) then FValue := NewValue;
  DoSomething;
end;
```

В этом примере чтение значения свойства `AValue` означает просто чтение поля `FValue`. Зато при присвоении значения внутри `SetValue` вызывается сразу два метода.

Если свойство должно только читаться или записываться, в его описании может присутствовать соответствующий метод:

```
type
  TAnObject = class(TObject)
    property AProperty: TSomeType read GetValue;
  end;
```

В этом примере вне объекта значение свойства можно лишь прочитать; попытка присвоить свойству `AProperty` значение вызовет ошибку компиляции.

Для присвоения свойству значения по умолчанию используется ключевое слово `default`:

```
property Visible: boolean read FVisible write SetVisible default True;
```

Это означает, что при запуске программы свойство будет установлено компилятором в `True`.

Свойство может быть и векторным; в этом случае оно внешне выглядит как массив:

```
property APoints[Index : Integer]:TPoint read GetPoint write SetPoint;
```

На самом деле в классе может и не быть соответствующего поля — массива. Напомним, что вся обработка обращений к внутренним структурам класса может быть замаскирована.

Для векторного свойства необходимо описать не только тип элементов массива, но также имя и тип индекса. После ключевых слов `read` и `write` в этом случае должны стоять имена методов — использование здесь полей массивов недопустимо. Метод, читающий значение векторного свойства, должен быть описан как функция, возвращающая значение того же типа, что и элементы свойства, и имеющая единственный параметр того же типа и с тем же именем, что и индекс свойства:

```
function GetPoint(Index:Integer):TPoint;
```

Аналогично, метод, помещающий значения в такое свойство, должен первым параметром иметь индекс, а вторым — переменную нужного типа (которая может быть передана как по ссылке, так и по значению):

```
procedure SetPoint(Index:Integer; NewPoint:TPoint);
```

У векторных свойств есть еще одна важная особенность. Некоторые классы в Delphi (списки `TList`, наборы строк `TStrings`) "построены" вокруг основного векторного свойства *(см. гл. 7)*. Основной метод такого класса дает доступ к некоторому массиву, а все остальные методы являются как бы вспомогательными. Специально для облегчения работы в этом случае векторное свойство может быть описано с ключевым словом `default`:

```
type
  TMyObject = class;
  property Strings[Index: Integer]: string read Get write Put; default;
end;
```

Если у объекта есть такое свойство, то можно его не упоминать, а ставить индекс в квадратных скобках сразу после имени объекта:

```
var AMyObject: TMyObject;
begin
  ...
```

```
  AMyObject.Strings[1] := 'First'; {первый способ}
  AMyObject[2] := 'Second'; {второй способ}
  ...
end.
```

Будьте внимательны, применяя зарезервированное слово `default`, — как мы увидели, для обычных и векторных свойств оно употребляется в разных случаях и с различным синтаксисом.

О роли свойств в Delphi красноречиво говорит следующий факт: у всех имеющихся в распоряжении программиста стандартных классов 100% полей недоступны и заменены базирующимися на них свойствами. Рекомендуем при разработке собственных классов придерживаться этого же правила.

Внимательный читатель обратил внимание, что при объяснении терминов "поле" и "свойство" мы использовали понятие метода, и наверняка понял его общий смысл. Итак, методом называется объявленная в классе функция или процедура, которая используется для работы с полями и свойствами класса. Согласно принципам ООП *(см. разд. "Инкапсуляция" далее в этой главе)*, обращаться к свойствам класса можно только через его методы. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут данные именно того объекта, который вызвал метод. На некоторых особенностях использования методов мы остановимся ниже.

События

Программистам, давно работающим в Windows, наверное, не нужно пояснять смысл термина "событие". Сама эта среда и написанные для нее программы управляются событиями, возникающими в результате воздействий пользователя, аппаратуры компьютера или других программ. Весточка о наступлении события — это сообщение Windows, полученное так называемой функцией окна. Таких сообщений сотни, и, по большому счету, написать программу для Windows — значит определить и описать реакцию на некоторые из них.

Работать с таким количеством сообщений, даже имея под рукой справочник, нелегко. Поэтому одним из больших преимуществ Delphi является то, что программист избавлен от необходимости работать с сообщениями Windows (хотя такая возможность у него есть). Типовых событий в Delphi — не более двух десятков, и все они имеют простую интерпретацию, не требующую глубоких знаний среды.

Рассмотрим, как реализованы события на уровне языка Object Pascal. *Событие* — это свойство процедурного типа, предназначенное для создания пользовательской реакции на то или иное входное воздействие:

```
property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
```

Здесь `FOnMyEvent` — поле процедурного типа, содержащее адрес некоторого метода. Присвоить такому свойству значение — значит указать объекту адрес метода, который будет вызываться в момент наступления события. Такие методы называют обработчиками событий. Например, запись:

```
Application.OnActivate := MyActivatingMethod;
```

означает, что при активизации объекта `Application` (так называется объект, соответствующий работающему приложению) будет вызван метод-обработчик `MyActivatingMethod`.

Внутри библиотеки времени выполнения Delphi вызовы обработчиков событий находятся в методах, обрабатывающих сообщения Windows. Выполнив необходимые действия, этот метод проверяет, известен ли адрес обработчика, и, если это так, вызывает его:

```
if Assigned(FOnMyEvent) then FOnMyEvent(Self);
```

События имеют разное количество и тип параметров в зависимости от происхождения и предназначения. Общим для всех является параметр `Sender` — он указывает на объект-источник события. Самый простой тип — `TNotifyEvent` — не имеет других параметров:

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

Тип метода, предназначенный для извещения о нажатии клавиши, предусматривает передачу программисту кода этой клавиши о передвижении мыши — ее текущих координат и т. п. Все события в Delphi принято предварять префиксом `On`: `OnCreate`, `OnMouseMove`, `OnPaint` и т. д. Дважды щелкнув в Инспекторе объектов на странице `Events` в поле любого события, вы получите в программе заготовку метода нужного типа. При этом его имя будет состоять из имени текущего компонента и имени события (без префикса `On`), а относиться он будет к текущей форме. Пусть, например, на форме `Form1` есть текст `Label1`. Тогда для обработки щелчка мышью (событие `OnClick`) будет создана заготовка метода `TForm1.Label1Click`:

```
procedure TForm1.Label1Click(Sender: TObject);
begin

end;
```

Поскольку события — это свойства объекта, их значения можно изменять в любой момент во время выполнения программы. Эта замечательная возможность называется *делегированием*. Можно в любой момент взять способы реакции на события у одного объекта и присвоить (делегировать) их другому:

```
Object1.OnMouseMove := Object2.OnMouseMove;
```

Принцип делегирования позволяет избежать трудоемкого процесса порождения новых дочерних классов для каждого специфического случая, заменяя его простой подстановкой процедур. При необходимости можно выбирать один из нескольких возможных вариантов обработчиков событий.

Но какой механизм позволяет подменять обработчики, ведь это не просто процедуры, а методы? Здесь как нельзя кстати приходится существующее в Object Pascal понятие указателя на метод. Отличие метода от процедуры состоит в том, что помимо явно описанных параметров методу всегда неявно передается еще и указатель на вызвавший его экземпляр класса (переменная `Self`). Вы можете описать процедурный тип, который будет совместим по присваиванию с методом (т. е. предусматривать получение `Self`). Для этого в описание процедуры нужно добавить зарезервированные слова `of object`. Указатель на метод — это указатель на такую процедуру.

```
type
 TMyEvent = procedure(Sender: TObject; var AValue: Integer) of object;

 T1stObject = class;
  FOnMyEvent: TMyEvent;
  property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
 end;

 T2ndObject = class;
  procedure SetValue1(Sender: TObject; var AValue: Integer);
  procedure SetValue2(Sender: TObject; var AValue: Integer);
 end;
...
var
  Obj1: T1stObject;
  Obj2: T2ndObject;
begin
 Obj1 := T1stObject.Create;
 Obj2 := T2ndObject.Create;
 Obj1.OnMyEvent := Obj2.SetValue1;
 Obj1.OnMyEvent := Obj2.SetValue2;
...
end.
```

Этот пример показывает, что при делегировании можно присваивать методы других классов. Здесь обработчиком события `OnMyEvent` объекта `Obj1` по очереди выступают методы `SetValue1` и `SetValue2` объекта `Obj2`.

Обработчики событий нельзя сделать просто процедурами — *они обязательно должны быть чьими-то методами*. Но их можно "отдать" какому-либо другому объекту. Более того, для этих целей можно описать и создать спе-

циальный объект. Его единственное предназначение — быть носителем методов, которые затем делегируются другим объектам. Разумеется, такой объект надо не забыть создать до использования его методов, а в конце — уничтожить. Можно и не делать этого, объявив методы методами класса, о которых речь пойдет в одном из последующих разделов.

Мы сейчас решили задачу использования нескольких разных обработчиков того или иного события для одного объекта. Но не менее часто требуется решить обратную задачу — а как использовать для различных событий разных объектов один и тот же обработчик?

Если никакой "персонификации" объекта, вызвавшего метод, не нужно, все делается тривиально и проблемы не возникает. Самый простой пример: в современных программах основные функции дублируются дважды — в меню и на панели инструментов. Естественно, сначала нужно создать и наполнить метод содержимым (скажем, для пункта меню), а затем в Инспекторе объектов указать его же для кнопки панели инструментов.

Более сложный случай, когда внутри такого метода нужно разобраться, кто собственно его вызвал. Если потенциальные кандидаты имеют разный объектный тип (как в предыдущем абзаце — кнопка и пункт меню), то именно объектный тип можно применить в качестве критерия:

```
If Sender is TMenuItem then ShowMessage('Выбран пункт меню');
```

Если же все объекты, разделяющие между собой один обработчик события, относятся к одному классу, то приходится прибегать к дополнительным ухищрениям. Типовой прием — использовать свойство `Tag`, которое имеется у всех компонентов, и, вполне вероятно, именно для этого и задумывалось:

```
const colors : array[0..7] of TColor =
 (clWhite,clRed,clBlue,clYellow,clAqua,clGreen,clMaroon,clBlack);

procedure TForm1.CheckBoxClick(Sender: TObject);
begin
  with TCheckBox(Sender) do
    if Checked
      then Color := Colors[Tag]
    else Color := clBtnFace;
end;
```

Пусть в форме имеется несколько переключателей. Для того чтобы при нажатии каждый из них окрашивался в свой цвет, нужно в Инспекторе объектов присвоить свойству `Tag` значения от 0 до 7 и для каждого связать событие `OnClick` с методом `CheckBoxClick`. Этот единственный метод справится с задачей для всех переключателей.

Инкапсуляция

В предыдущих разделах мы ввели ряд новых понятий, которыми будем пользоваться в дальнейшем. Теперь поговорим о принципах, составляющих суть объектно-ориентированного программирования. Таких принципов три — инкапсуляция, наследование и полиморфизм.

Как правило, объект — это сложная конструкция, свойства и поведение составных частей которой находятся во взаимодействии. К примеру, если мы моделируем взлетающий самолет, то после набора им определенной скорости и отрыва от земли принципы управления им полностью изменяются. Поэтому в объекте "самолет" явно недостаточно просто увеличить значение поля "скорость" на несколько километров в час — такое изменение должно автоматически повлечь за собой целый ряд других.

При создании программных объектов подобные ситуации можно моделировать, связывая со свойствами необходимые методы. Понятие инкапсуляции соответствует этому механизму.

Классическое правило объектно-ориентированного программирования утверждает, что для обеспечения надежности нежелателен прямой доступ к полям объекта: чтение и обновление их содержимого должно производиться посредством вызова соответствующих методов. Это правило и называется *инкапсуляцией*. В старых реализациях ООП (например, в Turbo Pascal) эта мысль внедрялась только посредством призывов и примеров в документации; в языке же Object Pascal есть соответствующая конструкция. В Delphi пользователь вашего объекта может быть полностью отгорожен от полей при помощи свойств *(см. выше)*.

Примечание

Дополнительные возможности ограничения доступа к нужным данным обеспечивают области видимости *(см. ниже)*.

Наследование

Вторым "столпом" ООП является *наследование*. Этот простой принцип означает, что если вы хотите создать новый класс, лишь немного отличающийся от старого, то совершенно нет необходимости в переписывании заново уже существующих полей и методов. Вы объявляете, что новый класс `TNewObject`

```
TNewObject = class(TOldObject);
```

является *потомком* или *дочерним классом* старого класса `TOldObject`, называемого *предком* или *родительским классом*, и добавляете к нему новые поля, методы и свойства — иными словами, то, что нужно при переходе от общего к частному.

Примечание

Прекрасный пример, иллюстрирующий наследование, представляет собой иерархия классов VCL.

В Object Pascal все классы являются потомками класса `TObject`. Поэтому, если вы создаете дочерний класс прямо от `TObject`, то в определении его можно не упоминать. Следующие два выражения одинаково верны:

```
TMyObject = class(TObject);
TMyObject = class;
```

Первый вариант, хотя он и более длинный, предпочтительнее — для устранения возможных неоднозначностей. Класс `TObject` несет очень серьезную нагрузку и будет рассмотрен отдельно.

Унаследованные от класса-предка поля и методы доступны в дочернем классе; если имеет место совпадение имен методов, то говорят, что они перекрываются.

Поведение методов при наследовании, без преувеличения, является краеугольным камнем объектно-ориентированного программирования. В зависимости от того, какие действия происходят при вызове, методы делятся на три группы. В первую группу отнесем статические методы, во вторую — виртуальные (`virtual`) и динамические (`dynamic`) и, наконец, в третью — появившиеся только в Delphi 4 — перегружаемые (`overload`) методы.

Методы первой группы полностью перекрываются в классах-потомках при их переопределении. При этом можно полностью изменить объявление метода. Методы второй группы при наследовании должны сохранять наименование и тип. Перегружаемые методы дополняют механизм наследования возможностью использовать нужный вариант метода (собственный или родительский) в зависимости от условий применения. Подробно все эти методы обсуждаются ниже.

Язык C++ допускает так называемое множественное наследование. В этом случае новый класс может наследовать часть своих элементов от одного родительского класса, а часть — от другого. Это, наряду с удобствами, зачастую приводит к проблемам. В Object Pascal понятие множественного наследования отсутствует. Если вы хотите, чтобы новый класс объединял свойства нескольких, породите классы-предки один от другого или включите в один класс несколько полей, соответствующих другим желаемым классам.

Полиморфизм

Рассмотрим внимательно следующий пример. Пусть у нас имеются некое обобщенное поле для хранения данных — класс `TField` и три его потомка — для хранения строк, целых и вещественных чисел:

```
type
 TField = class
  function GetData:string; virtual; abstract;
 end;

 TStringField = class(TField)
  FData : string;
  function GetData: string; override;
 end;

 TIntegerField = class(TField)
  FData : Integer;
  function GetData: string;override;
 end;

 TExtendedField = class(TField)
  FData : Extended;
  function GetData: string;override;
 end;
...
 function TStringField.GetData;
  begin
   Result := FData;
  end;

 function TIntegerField.GetData;
  begin
   Result := IntToStr(FData);
  end;

 function TExtendedField.GetData;
  begin
   Result:= FloatToStrF(FData, ffFixed, 7, 2);
  end;
....
procedure ShowData(AField : TField);
begin
  Form1.Label1.Caption := AField.GetData;
end;
```

В этом примере классы содержат разнотипные поля данных `FData` и только-то и "умеют", что сообщить о значении этих данных текстовой строкой (при помощи метода `GetData`). Внешняя по отношению к ним процедура `ShowData` получает объект в виде параметра и показывает эту строку.

Правила контроля *соответствия типов* (typecasting) языка Object Pascal гласят, что *объекту как указателю на экземпляр объектного типа может быть*

присвоен адрес любого экземпляра любого из дочерних типов. В процедуре `ShowData` параметр описан как `TField` — это значит, что в нее можно передавать объекты классов и `TStringField`, и `TIntegerField`, и `TExtendedField`, и любого другого потомка класса `TField`.

Но какой (точнее, чей) метод `GetData` при этом будет вызван? Тот, который соответствует классу *фактически* переданного объекта. Этот принцип называется *полиморфизмом*, и он, пожалуй, представляет собой наиболее важный козырь ООП.

Допустим, вы имеете дело с некоторой совокупностью явлений или процессов. Чтобы смоделировать их средствами ООП, нужно выделить их самые общие, типовые черты. Те из них, которые не изменяют своего содержания, должны быть реализованы в виде статических методов. Те же, которые изменяются при переходе от общего к частному, лучше облечь в форму виртуальных методов.

Основные, "родовые" черты (методы) нужно описать в классе-предке и затем перекрывать их в классах-потомках. В нашем примере программисту, пишущему процедуру вроде `ShowData`, важно лишь, что любой объект, переданный в нее, является потомком `TField` и он умеет сообщить о значении своих данных (выполнив метод `GetData`). Если, к примеру, такую процедуру скомпилировать и поместить в динамическую библиотеку, то эту библиотеку можно будет раз и навсегда использовать без изменений, хотя будут появляться и новые, неизвестные в момент ее создания классы-потомки `TField`.

Наглядный пример использования полиморфизма дает среда Delphi. В ней имеется класс `TComponent`, на уровне которого сосредоточены определенные "правила" взаимодействия компонентов со средой разработки и с другими компонентами. Следуя этим правилам, можно порождать от класса `TComponent` свои компоненты, настраивая Delphi на решение специальных задач.

Теперь, надеемся, стало более или менее ясно, какие преимущества ООП позволили ему стать основным способом разработки серьезного программного обеспечения. Те, кто начинал программировать еще для Windows 3.0, наверняка помнят, сколько усилий требовалось при написании совершенно тривиального кода. Сейчас для того же самого в Delphi достаточно буквально пары щелчков мышью. На самом деле именно сложность программирования для Windows стала катализатором внедрения ООП.

Методы

Из предыдущего материала читатели узнали, что функционирование объектов обеспечивается различными типами методов, которые различаются особенностями реализации механизма наследования. Теперь настало время рассмотреть эти методы более подробно.

Абстрактными называются методы, которые определены в классе, но не содержат никаких действий, никогда не вызываются и обязательно должны быть переопределены в потомках класса. Абстрактными могут быть только виртуальные и динамические методы. В Object Pascal такие методы объявляются с помощью одноименной директивы. Она указывается при описании метода:

```
procedure NeverCallMe; virtual; abstract;
```

При этом никакого кода для этого метода писать не нужно. Вызов метода `NeverCallMe` приведет к созданию исключительной ситуации `EAbstractError` *(исключительным ситуациям посвящена гл. 4)*.

Пример с классом `TField` из *разд. "Полиморфизм" этой главы* поясняет, для чего нужно использование абстрактных методов. В данном случае класс `TField` не используется сам по себе; его основное предназначение — быть родоначальником иерархии конкретных классов-"полей" и дать возможность абстрагироваться от частностей. Хотя параметр процедуры `ShowData` и описан как `TField`, но, если передать в нее объект этого класса, произойдет исключительная ситуация вызова абстрактного метода.

Статические методы, а также любые поля в объектах-потомках ведут себя одинаково: вы можете без ограничений перекрывать старые имена и при этом изменять тип методов. Код нового статического метода полностью перекрывает (заменяет собой) код старого метода:

```
type
T1stObj = class
  i : Extended;
  procedure SetData(AValue: Extended);
  end;
T2ndObj = class(T1stObj)
  i : Integer;
  procedure SetData(AValue: Integer);
 end;
....
procedure T1stObj.SetData;
begin
i := 1.0;
end;
procedure T2ndObj.SetData;
begin
i := 1;
inherited SetData(0.99);
end;
```

В этом примере *разные* методы с именем `SetData` присваивают значения *разным* полям с именем `i`. Перекрытое (одноименное) поле предка недоступно в потомке; поэтому, конечно, два одноименных поля с именем `i` — это нонсенс; так сделано только для примера.

Примечание

В практике программирования принято присваивать всем идентификаторам в программе (в том числе полям объектов) осмысленные названия. Это существенно облегчит работу с исходным кодом не только другим разработчикам, но и вам.

В отличие от поля, внутри других методов перекрытый метод доступен при указании зарезервированного слова `inherited`. По умолчанию все методы объектов являются статическими — их адрес определяется еще на стадии компиляции проекта, поэтому они вызываются быстрее всего. Может быть, читателю еще не ясно, для чего упоминается этот факт. Просто запомните его, он понадобится при сравнении статических и виртуальных методов.

Принципиально отличаются от статических *виртуальные* и *динамические* методы. Они должны быть объявлены путем добавления соответствующей директивы `virtual` или `dynamic`. Обе эти категории существовали и в прежних версиях языка Pascal. С точки зрения наследования методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип.

Если задуматься над рассмотренным выше примером, становится ясно, что у компилятора нет возможности определить класс объекта, фактически переданного в процедуру `ShowData`. Нужен механизм, позволяющий определить это прямо во время выполнения. Такой механизм называется *поздним связыванием* (late binding).

Естественно, что этот механизм должен быть каким-то образом связан с передаваемым объектом. Для этого используются *таблица виртуальных методов* (Virtual Method Table, VMT) и *таблица динамических методов* (Dynamic Method Table, DMT).

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес.

Такая таблица есть для каждого класса (объектного типа). В ней хранятся адреса *всех* виртуальных методов класса, независимо от того, унаследованы ли они от предка или перекрыты в данном классе. Отсюда и достоинства, и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большое количество памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса *только тех* динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, класс `TObject`, где имеется стандартный обработчик вызова динамических методов. Экономия памяти налицо.

Для перекрытия и виртуальных, и динамических методов служит директива `override`, с помощью которой (и только с ней!) можно переопределять оба этих типа методов. Приведем пример:

```
type
  TFirstClass = class
  FMyField1: Integer;
  FMyField2: Longint;
  procedure StatMethod;
  procedure VirtMethod1; virtual;
  procedure VirtMethod2; virtual;
  procedure DynaMethod1; dynamic;
  procedure DynaMethod2; dynamic;
  end;

  TSecondClass = class(TMyObject)
  procedure StatMethod;
  procedure VirtMethod1; override;
  procedure DynaMethod1; override;
end;
var
  Obj1: TFirstClass;
  Obj2: TSecondClass;
```

Первый из методов в примере создается заново, остальные два — перекрываются. Попытка применить директиву `override` к статическому методу вызовет ошибку компиляции.

Примечание

Будьте внимательны: попытка перекрытия с директивой не `override`, а `virtual` или `dynamic` приведет на самом деле к созданию нового одноименного метода.

Перегрузка методов

Есть еще одна, совершенно особенная разновидность методов — *перегружаемые*.

Эту категорию методов нельзя назвать антагонистом двух предыдущих: и статические, и виртуальные, и динамические методы могут быть перегружаемыми. *Перегрузка методов нужна, чтобы произвести одинаковые или похожие действия с разнотипными данными.*

Рассмотрим немного измененный пример, иллюстрирующий статические методы:

```
type
T1stObj = class
  FExtData : Extended;
  procedure SetData(AValue: Extended);
  end;
T2ndObj = class(T1stObj)
  FIntData : Integer;
  procedure SetData(AValue: Integer);
 end;
...
var T1: T1stObj;
    T2 : T2ndObj;
```

В этом случае попытка вызова из объекта `T2` методов

```
...
T2.SetData(1.0);
T2.SetData(1);
...
```

вызовет ошибку компиляции на первой из двух строк. Для компилятора внутри `T2` статический метод с параметром типа `extended` перекрыт, и он его "не признает". Где же выход из сложившегося положения? Переименовать один из методов, например создать `SetIntegerData` и `SetExtendedData`? Можно, но если методов не два, а, скажем, сто, моментально возникнет путаница. Сделать методы виртуальными? Нельзя, поскольку тип и количество параметров в одноименных виртуальных методах должны в точности совпадать. Теперь для этого существуют перегружаемые методы, объявляемые при помощи директивы `overload`:

```
type
T1stObj = class
  FExtData : Extended;
  procedure SetData(AValue: Extended);overload;
  end;
```

```
T2ndObj = class(T1stObj)
  FIntData : Integer;
  procedure SetData(AValue: Integer); overload;
 end;
```

Объявив метод `SetData` перегружаемым, в программе можно использовать обе его реализации одновременно. Это возможно потому, что компилятор определяет тип передаваемого параметра (целый или с плавающей точкой) и в зависимости от этого подставит вызов соответствующего метода: для целочисленных данных — метод объекта `T2ndObj`, для данных с плавающей точкой — метод объекта `T1stObj`.

Можно перегрузить и виртуальный (динамический) метод. Надо только в этом случае добавить директиву `reintroduce`:

```
type
T1stObj = class
  FExtData : Extended;
  procedure SetData(AValue: Extended); overload; virtual;
  end;
T2ndObj = class(T1stObj)
  FIntData : Integer;
  procedure SetData(AValue: Integer); reintroduce; overload;
 end;
```

На перегрузку методов накладывается ограничение — нельзя перегружать методы, находящиеся в области видимости `published`, *т. е. те, которые будут использоваться в Инспекторе объектов.*

Области видимости

При описании нового класса важен разумный компромисс. С одной стороны, требуется скрыть от других методы и поля, представляющие собой внутреннее устройство класса (для этого и придуманы свойства). Маловажные детали на уровне пользователя объекта будут бесполезны и только помешают целостности восприятия.

С другой стороны, если слишком ограничить того, кто будет порождать классы-потомки, и не обеспечить ему достаточный набор инструментальных средств и свободу маневра, то он и не станет использовать ваш класс.

В модели объектов языка Object Pascal существует механизм доступа к составным частям объекта, определяющий области, где ими можно пользоваться *(области видимости)*. Поля и методы могут относиться к четырем группам (секциям), отличающимся областями видимости. Методы и свойства могут быть *общими* (секция `public`), *личными* (секция `private`), *защищен-*

ными (секция `protected`) и *опубликованными* (секция `published`). Есть еще и пятая группа, `automated`, она ранее использовалась для создания объектов COM; теперь она присутствует в языке только для обратной совместимости с программами на Delphi версий 3—5.

Области видимости, определяемые первыми тремя директивами, таковы.

- Поля, свойства и методы секции `public` не имеют ограничений на видимость. Они доступны из других функций и методов объектов как в данном модуле, так и во всех прочих, ссылающихся на него.
- Поля, свойства и методы, находящиеся в секции `private`, доступны только в методах класса и в функциях, содержащихся в том же модуле, что и описываемый класс. Такая директива позволяет полностью скрыть детали внутренней реализации класса. Свойства и методы из секции `private` можно изменять, и это не будет сказываться на программах, работающих с объектами этого класса. Единственный способ для кого-то другого обратиться к ним — переписать заново созданный вами модуль (если, конечно, доступны исходные тексты).
- Поля, свойства и методы секции `protected` также доступны только внутри модуля с описываемым классом. Но — и это главное — они доступны в классах, являющихся потомками данного класса, в том числе и в других модулях. Такие элементы особенно необходимы для разработчиков новых компонентов — потомков уже существующих. Оставляя свободу модернизации класса, они все же скрывают детали реализации от того, кто только пользуется объектами этого класса.

Рассмотрим пример, иллюстрирующий три варианта областей видимости.

Листинг 1.1. Пример задания областей видимости методов

```
unit First;
interface


type
TFirstObj = class
private
 procedure Method1;
protected
 procedure Method2;
public
 procedure Method3;
end;

procedure TestProc1;
```

```
unit Second;
interface
uses First;

type
TSecondObj = class(TFirstObj)
 procedure Method4;
end;



procedure TestProc2;
```

```
implementation

uses dialogs;
var AFirstObj: TFirstObj;

procedure TestProc1;
begin
 AFirstObj := TFirstObj.Create;
 AFirstObj.Method1; {допустимо}
 AFirstObj.Method2; {допустимо}
 AFirstObj.Method3; {допустимо}
 AFirstObj.Free;
end;

procedure TFirstObj.Method1;
begin
 ShowMessage('1');
end;
procedure TFirstObj.Method2;
begin
 ShowMessage('2');
 Method1;
end;
procedure TFirstObj.Method3;
begin
 ShowMessage('3');
 Method2;
end;

end.
```

```
implementation

var AFirstObj : TFirstObj;
   ASecondObj: TSecondObj;

procedure TSecondObj.Method4;
begin
  Method1; {недопустимо -
произойдет ошибка компиляции}
  Method2; {допустимо}
  Method3; {допустимо}
end;

procedure TestProc2;
begin
 AFirstObj := TFirstObj.Create;
 AFirstObj.Method1; {недопустимо}
 AFirstObj.Method2; {недопустимо}
 AFirstObj.Method3; {допустимо}
 AFirstObj.Free;

 ASecondObj := TSecondObj.Create;
 ASecondObj.Method1; {недопустимо}
 ASecondObj.Method2; {допустимо}
 ASecondObj.Method3; {допустимо}
 ASecondObj.Free;
end;

end.
```

Если к этому примеру добавить модуль `Third` и попробовать вызвать методы классов `TFirstObj` и `TSecondObj` оттуда, то к числу недоступных будет отнесен и `Method2` — он доступен только в том модуле, в котором описан.

Наконец, область видимости, определяемая четвертой директивой — `published`, имеет особое значение для интерфейса визуального проектирования Delphi. В этой секции должны быть собраны те свойства объекта, которые будут видны не только во время исполнения приложения, но и из среды разработки. Публиковать можно свойства большинства типов, за исключением старого типа `real` (теперь он называется `real48`), свойств типа "массив" и некоторых других. Все свойства компонентов, доступные через

Инспектор объектов, являются их опубликованными свойствами. Во время выполнения такие свойства общедоступны, как и public.

Три области видимости — private, protected, public — как бы упорядочены по возрастанию видимости методов. В классах-потомках можно повысить видимость методов и свойств, но не понизить ее. При описании дочернего класса можно переносить методы и свойства из одной сферы видимости в другую, не переписывая их заново и даже не описывая — достаточно упомянуть о нем в другом месте:

```
type
 TFirstObj = class
 private
  FNumber: Integer;
 protected
  property Number: Integer read: FNumber;
 end;
...
 TSecondObj = class(TFirstObj)
 published
  property Number;
 end;
```

Если какое-либо свойство объекта из состава VCL принадлежит к области public, вернуть его в private невозможно. Напротив, обратная процедура широко практикуется в Delphi. У многих компонентов (например, TEdit) есть предок (в данном случае TCustomEdit), который отличается только отсутствием опубликованных свойств. Так что, если вы хотите создать новый редактирующий компонент, порождайте его на базе TCustomEdit и публикуйте только те свойства, которые считаете нужными. Разумеется, если вы поместили свойство в область private, "достать" его оттуда в потомках возможности уже нет.

Объект изнутри

Теперь, когда мы разобрались с основными определениями и механизмами ООП, настало время более подробно изучить, что представляет собой объект и как он работает. Ясно, что каждый экземпляр класса содержит отдельную копию всех его полей. Ясно, что где-то в его недрах есть указатели на таблицу виртуальных методов и таблицу динамических методов. А что еще там имеется? И как происходит вызов методов? Вернемся к примеру из *разд. "Полиморфизм" данной главы*:

```
type
  TFirstClass = class
```

```
  FMyField1: Integer;
  FMyField2: Longint;
  procedure StatMethod;
  procedure VirtMethod1; virtual;
  procedure VirtMethod2; virtual;
  procedure DynaMethod1; dynamic;
  procedure DynaMethod2; dynamic;
  end;

  TSecondClass = class(TMyObject)
  procedure StatMethod;
  procedure VirtMethod1; override;
  procedure DynaMethod1; override;
end;
  Obj1: TFirstClass;
  Obj2: TSecondClass;
```

На рис. 1.1 показано, как будет выглядеть внутренняя структура рассмотренных в нем объектов.

Первое поле каждого экземпляра того или иного объекта содержит указатель на его класс. Класс как структура состоит из двух частей. Начиная с адреса, на который ссылается указатель на класс, располагается таблица виртуальных методов. Напомним, что она содержит адреса всех виртуальных методов класса, включая унаследованные от предков. Длина таблиц VMT объектов `Obj1` и `Obj2` одинакова — по два элемента (8 байт). Перед таблицей виртуальных методов расположена специальная структура, содержащая дополнительную служебную информацию. В ней содержатся данные, полностью характеризующие класс: его имя, размер экземпляра, указатели на класс-предок, имя класса и т. д. На рис. 1.1 она показана одним блоком, а ее содержимое расшифровано ниже.

Одно из полей структуры содержит адрес таблицы динамических методов класса (DMT). Таблица имеет следующий формат — в начале слово, содержащее количество элементов таблицы; затем — слова, соответствующие индексам методов. Нумерация индексов начинается с −1 и идет по убывающей. После индексов идут собственно адреса динамических методов. Обратите внимание, что DMT объекта `Obj1` состоит из двух элементов, `Obj2` — из одного, соответствующего перекрытому методу `DynaMethod1`. В случае вызова `Obj2.DynaMethod2` индекс не будет найден в таблице DMT `Obj2`, и произойдет обращение к DMT `Obj1`. Именно так экономится память при использовании динамических методов.

В языке Object Pascal определены два оператора — `is` и `as`, неявно обращающиеся к таблице динамических методов. Оператор `is` предназначен для проверки совместимости по присваиванию экземпляра объекта с заданным классом.

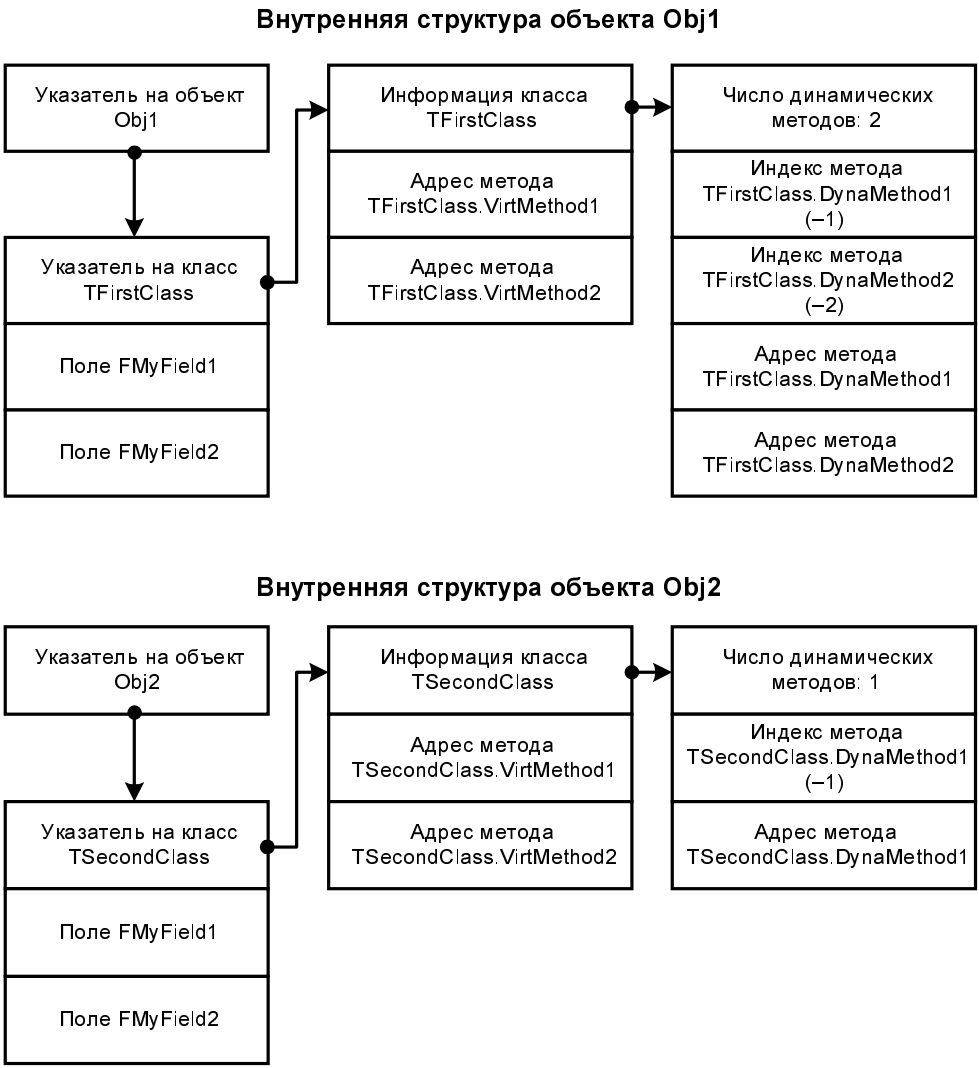


Рис. 1.1. Внутренняя структура объектов `Obj1` и `Obj2`

Выражение вида:

```
AnObject is TObjectType
```

принимает значение `True`, только если объект `AnObject` совместим по присваиванию с классом `TObjectType`, т. е. является объектом этого класса или одного из классов, порожденных от него. Кстати, определенная проверка происходит еще при компиляции: если формально объект и класс несовместимы, то компилятор выдаст ошибку в этом операторе.

Оператор `as` введен в язык специально для приведения объектных типов. С его помощью можно рассматривать экземпляр объекта как принадлежащий к другому совместимому типу:

```
with ASomeObject as TAnotherType do...
```

От стандартного способа приведения типов с помощью конструкции `TAnotherType(ASomeObject)` использование оператора `as` отличается наличием проверки на совместимость типов во время выполнения (как в операторе `is`): попытка приведения к несовместимому типу приводит к возникновению исключительной ситуации `EInvalidCast` *(см. гл. 4)*. После применения оператора `as` сам объект остается неизменным, но вызываются те его методы, которые соответствуют присваиваемому классу.

Очень полезным может быть оператор `as` в методах-обработчиках событий. Для обеспечения совместимости в 99% случаев источник события `Sender` имеет тип `TObject`, хотя в тех же 99% случаев им является форма или другие компоненты. Поэтому, чтобы иметь возможность пользоваться их свойствами, применяют оператор `as`:

```
(Sender as TControl).Caption := "Thanks!";
```

Вся информация, описывающая класс, создается и размещается в памяти на этапе компиляции. Возникает резонный вопрос: а нельзя ли получить доступ к ней, не создавая экземпляр объекта? Да, можно. Доступ к информации класса вне методов этого класса можно получить, описав соответствующий указатель, который называется *указателем на класс*, или *указателем на объектный тип* (class reference). Он описывается при помощи зарезервированных слов `class of`. Например, указатель на класс `TObject` описан в модуле SYSTEM.PAS и называется `TClass`:

```
type
  TObject = class;
  TClass = class of TObject;
```

Аналогичные указатели уже описаны и для других важных классов. Вы можете использовать в своей программе `TComponentClass`, `TControlClass` и т. п.

Указатели на классы тоже подчиняются правилам приведения объектных типов. Указатель на класс-предок может ссылаться и на любые дочерние классы; обратное невозможно:

```
type
 TFirst = class
...
 end;
```

```
 TSecond = class(TFirst)
...
 end;
 TFirstClass = class of TFirst;
 TSecondClass = class of TSecond;

var
 AFirst : TFirstClass;
 ASecond : TSecondClass;
begin
 AFirst := TSecond; {допустимо}
 ASecond := TFirst; {недопустимо}
end.
```

С указателем на класс тесно связано понятие *методов класса*. Такие методы можно вызывать без создания экземпляра объекта — с указанием имени класса, в котором они описаны. Перед описанием метода класса нужно поставить зарезервированное слово `class`:

```
type
  TMyObject = class(TObject)
    class function GetSize: string;
  end;
var
  MyObject: TMyObject;
  AString: string;
begin
  AString := TMyObject.GetSize;
  MyObject := TMyObject.Create;
  AString := MyObject.GetSize;
end.
```

Разумеется, методы класса не могут использовать значения, содержащиеся в полях класса: ведь экземпляра-то не существует. Возникает вопрос: для чего нужны такие методы?

Важнейшие методы класса определены в самом `TObject`: они как раз и позволяют, не углубляясь во внутреннюю структуру класса, извлечь оттуда практически всю необходимую информацию.

Резюме

В этой главе рассмотрены основы объектно-ориентированного программирования в Delphi. Объект обладает свойствами и методами, которые позво-

ляют изменять значения свойств. Знание основ ООП необходимо для изучения всех глав не только этой части, но и всех последующих. Ведь компоненты Delphi — это объекты, размещенные в специальной библиотеке VCL. А ни одна глава этой книги не обходится без описания возможностей тех или иных компонентов.

Рассмотренные в данной главе возможности объектов используются при создании исходного кода приложений Delphi.

ГЛАВА 2

Библиотека визуальных компонентов VCL и ее базовые классы

Все классы библиотеки визуальных компонентов произошли от группы базовых классов, которые лежат в основе иерархии VCL. Самый общий предок компонентов — это класс `TObject`, инкапсулирующий простейший объект. Как известно *(см. гл. 1)*, каждый объект наследует свойства и методы родительского класса. К объекту можно добавить новые свойства и методы, но нельзя удалить унаследованные. Объект-наследник в свою очередь может стать родительским для нового класса, который унаследует возможности всех своих предков.

Поэтому иерархия базовых классов VCL продумана чрезвычайно тщательно — ведь на их основе создано все множество компонентов, используемых в Delphi. Особое место среди базовых классов, помимо `TObject`, занимают `TComponent` (от него происходят все компоненты) и `TControl` (от него происходят все элементы управления).

В этой главе рассматривается иерархия базовых классов и их возможности. Представленные здесь сведения помогут разобраться с основными механизмами функционирования компонентов. Настоящая глава послужит справочным материалом для тех, кто создает собственные объекты и элементы управления.

Иерархия базовых классов

В основе всего многообразия классов и компонентов, используемых в Delphi, лежат всего лишь пять базовых классов (рис. 2.1). Они обеспечивают выполнение основных функций любого объекта — будь это стандартный компонент VCL или специализированный объект, выполняющий некоторые операции в приложении.

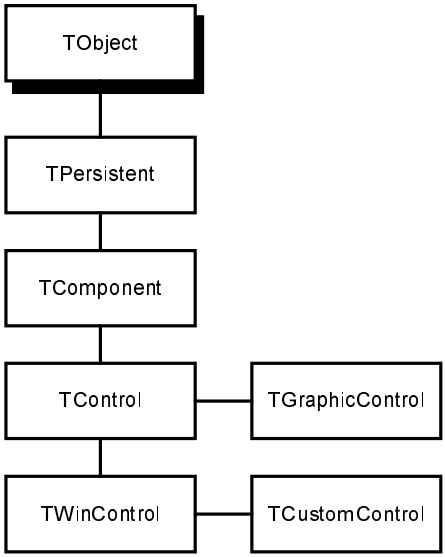


Рис. 2.1. Иерархия базовых классов VCL

Благодаря механизму наследования свойств и методов, потомки базовых классов умеют "общаться" друг с другом; работают в среде разработки, взаимодействуя с Палитрой компонентов и Инспектором объектов; распознаются операционной системой как элементы управления и окна.

В основе иерархии классов лежит класс `TObject`. Он обеспечивает выполнение важнейших функций "жизнедеятельности" любого объекта. Благодаря ему, каждый класс получает в наследство механизмы создания экземпляра объекта и его уничтожения.

Обычно разработчик даже не задумывается о том, как объект будет создан и что необходимо сделать для его корректного уничтожения. Компоненты VCL создаются и освобождают занимаемые ресурсы автоматически. Иногда разработчику приходится создавать и удалять объекты самостоятельно. Но даже в этом случае ему достаточно вызвать соответствующие конструктор и деструктор:

```
var SomeList: TStrings;
...
 SomeList := TStrings.Create;
...
 SomeList.Free;
```

За кажущейся простотой этих операций скрывается довольно сложная реализация указанных процессов. Практически весь исходный код класса

`TObject` написан на ассемблере для обеспечения наибольшей эффективности операций, которые будут выполняться в каждом его потомке.

Кроме этого, класс `TObject` обеспечивает создание и хранение информации об экземпляре объекта и обслуживание очереди сообщений.

Класс `TPersistent` происходит непосредственно от класса `TObject`. Он обеспечивает своих потомков возможностью взаимодействовать с другими объектами и процессами на уровне данных. Его методы позволяют передавать данные в потоки, а также обеспечивают взаимодействие объекта с Инспектором объектов.

Класс `TComponent` является важнейшим для всех компонентов. Непосредственно от него можно создавать любые невизуальные компоненты. Механизмы, реализованные в классе `TComponent`, обеспечивают взаимодействие компонента со средой разработки, главным образом с Палитрой компонентов и Инспектором объектов. Благодаря возможностям этого класса, компоненты начинают работать на форме проекта уже на этапе разработки.

Класс `TControl` происходит от класса `TComponent`. Его основное назначение — обеспечить функционирование визуальных компонентов. Каждый визуальный компонент, произошедший от `TControl`, наделяется основными признаками элемента управления. Благодаря этому, каждый визуальный компонент умеет работать с GUI (Graphic User Interface — графический интерфейс пользователя ОС) и отображать себя на экране.

Класс `TWinControl` расширяет возможности разработчиков по созданию элементов управления. Он наследуется от класса `TControl` и обеспечивает создание оконных элементов управления.

На основе класса `TWinControl` создан еще один дополнительный класс — `TCustomControl`. Он обеспечивает создаваемые на его основе компоненты возможностями по использованию канвы — специального объекта, предназначенного для отображения графики *(подробнее о канве см. гл. 11)*.

Класс `TCustomControl` является общим предком для целой группы классов, обеспечивающих создание различных нестандартных типов оконных (получающих фокус) элементов управления Windows: редакторов, списков и т. д.

Для создания неоконных (не получающих фокус) элементов управления используется класс `TGraphicControl`, являющийся потомком класса `TControl`.

В целом иерархия базовых классов обеспечивает полноценную работу разработчиков в Delphi, позволяя проектировать любые типы приложений.

Ниже мы остановимся на основных свойствах и методах базовых классов, выделяя только те, которые могут пригодиться в реальной работе. Часть из них доступна в Инспекторе объектов, часть может быть использована в программном коде.

Класс *TObject*

Класс `TObject` является родоначальником всей иерархии использующихся в Delphi классов VCL. Он реализует функции, которые обязательно будет выполнять любой объект, который может быть создан в среде разработки. Учитывая гигантское разнообразие возможных областей применения объектов в процессе создания приложений, можно сказать, что круг общих для всех классов операций весьма невелик.

В первую очередь — это создание экземпляра объекта и его уничтожение. Любой объект выполняет эти две операции в обязательном порядке.

Процесс создания объекта включает выделение области адресного пространства, установку указателя на экземпляр объекта, задание начальных значений свойств и выполнение установочных действий, связанных с назначением объекта. В общем случае две последние операции могут не выполняться.

Указатель на экземпляр объекта передается в переменную объектного типа, которая в дальнейшем будет идентифицировать объект в программном коде приложения. В приведенном выше фрагменте кода переменная объектного типа `SomeList` объявлена как экземпляр типа `TStrings`. При создании экземпляра этого типа конструктор `Create` возвращает в переменную `SomeList` указатель на выделенную для нового объекта область памяти. Для этого применяется метод `NewInstance`, который вызывается в конструкторе автоматически:

```
class function NewInstance: TObject; virtual;
```

Объект класса `TObject` обеспечивает выполнение этого процесса для любого порожденного от него объекта. А уже внутри конструктора, который унаследован от класса `TObject`, можно предусмотреть инициализацию переменных и выполнение дополнительных операций.

Объявление конструктора выглядит следующим образом:

```
constructor Create;
```

В конструкторах потомков это объявление может перекрываться, но при необходимости вызвать конструктор предка используется оператор `inherited`:

```
constructor TSomeObject.Create;
begin
 inherited Create;
...
end;
```

Для уничтожения экземпляра объекта в классе `TObject` предназначены методы `Destroy` и `Free`:

```
destructor Destroy; virtual;
procedure Free;
```

Как видно из объявления, настоящим деструктором является метод `Destroy`. Он обеспечивает освобождение всех занимаемых экземпляром объекта ресурсов. Обычно при создании новых классов деструктор всегда перекрывается для того, чтобы корректно завершить работу с данными.

Примечание

Обратите внимание, что обычно унаследованный конструктор вызывается в первую очередь. Это необходимо для того, чтобы выполнить все нужные операции при создании объекта до инициализации его собственных свойств. При уничтожении объекта обычно сначала выполняются завершающие операции и только в самом конце вызывается унаследованный деструктор, чтобы выполнить собственно уничтожение объекта.

При уничтожении объектов рекомендуется вместо деструктора использовать метод `Free`, который просто вызывает деструктор, но перед этим проверяет, чтобы указатель на экземпляр объекта был не пустым (не был равен `Nil`). Это позволяет избежать серьезных ошибок.

Если объект является владельцем других объектов (например, форма владеет всеми размещенными на ней компонентами), то его метод `Free` автоматически вызовет эти же методы для всех объектов. Поэтому при закрытии формы разработчик избавлен от необходимости заботиться об уничтожении всех компонентов.

Для освобождения занимаемой объектом памяти деструктор автоматически вызывает метод `FreeInstance`:

```
procedure FreeInstance; virtual;
```

Каждый объект должен содержать некоторую информацию о себе, которая используется приложением и средой разработки. Поэтому класс `TObject` содержит ряд методов, обеспечивающих представление этой информации в потомках.

Метод

```
class function ClassInfo: Pointer;
```

возвращает указатель на таблицу информации времени выполнения (RTTI). Эта информация используется в среде разработки и в приложении.

Функция

```
class function ClassName: ShortString;
```

возвращает имя типа объекта, которое может быть использовано для идентификации. Например, один метод-обработчик щелчка на кнопке может работать с несколькими типами компонентов кнопок:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
 if Sender is TBitBtn
  then TBitBtn(Sender).Enabled := False;
 if Sender is TSpeedButton
  then TSpeedButton(Sender).Down := True;
end;
```

Метод

```
class function ClassNameIs(const Name: string): Boolean;
```

позволяет определить, является ли данный объект того типа, имя которого передано в параметре `Name`. В случае положительного ответа функция возвращает `True`.

Как известно, программирование для Windows основано на событиях. Каждое приложение и каждый программный объект должны уметь реагировать на сообщение о событиях и, в свою очередь, рассылать сообщения. В выполнении этих операций заключается третья общая для всех объектов функция.

Метод

```
procedure Dispatch(var Message); virtual;
```

осуществляет обработку сообщений, поступающих объекту. Он определяет, сможет ли объект обработать сообщение при помощи собственных обработчиков событий. В случае отсутствия таких методов сообщение передается аналогичному методу `Dispatch` класса-предка (если он есть).

Класс `TObject` имеет предопределенный обработчик событий:

```
procedure DefaultHandler(var Message); virtual;
```

Кроме рассмотренных здесь методов, класс `TObject` имеет еще несколько методов, которые в основном применяются для взаимодействия объекта со средой разработки.

В целом класс `TObject` может служить для создания на его основе некоторых простых классов для использования в приложениях.

Класс *TPersistent*

"Persistent" в переводе с английского означает "устойчивый", "постоянный". Что же такого постоянного в одноименном классе? Ответ таков: виртуальный метод

```
procedure Assign(Source: TPersistent);
```

Этот важнейший метод осуществляет копирование содержимого одного объекта (`Source`) в другой (`Self`, т. е. в объект, вызвавший метод `Assign`). При этом объект-получатель остается самим собой, чего нельзя достигнуть, используя простое присваивание переменных объектного типа:

```
FirstObject := SecondObject;
```

Ведь в этом случае указатель на одну область адресного пространства, содержащую экземпляр класса (объект), замещается указателем на другую область адресного пространства, содержащую другой объект.

Метод `Assign` позволяет продублировать объект — присвоить одному объекту значения всех свойств другого. При этом объекты не обязательно должны быть одного и того же класса; более того, они не обязательно должны находиться в отношениях "родитель-потомок". Данный метод тем и хорош, что позволяет полиморфное присвоение. Конструкция

```
Clipboard.Assign(Picture);
```

позволяет скопировать содержимое картинки `Picture` в папку обмена Windows (объект `Clipboard`). Какова здесь логика? Известно, что в папку обмена можно поместить растровую картинку, текст, метафайл, мультимедийные данные и т. п. Метод `Assign` класса `TClipboard` переписан таким образом, чтобы обеспечить присвоение (т. е. реальное перемещение в папку обмена) всех этих данных.

```
procedure TClipboard.Assign(Source: TPersistent);
begin
  if Source is TPicture
    then AssignPicture(TPicture(Source))
  else
    if Source is TGraphic
      then AssignGraphic(TGraphic(Source))
    else inherited Assign(Source);
end;
```

Для обеспечения взаимодействия потомков класса `TPersistent` со средой разработки предназначен метод

```
function GetNamePath: string; dynamic;
```

Он возвращает имя объекта для передачи его в Инспектор объектов.

Для взаимодействия с потоками при загрузке и сохранении компонентов предназначен следующий метод:

```
procedure DefineProperties(Filer: TFiler); virtual;
```

Класс `TPersistent` никогда не используется напрямую, от него порождаются потомки, которые должны уметь передавать другим объектам значения своих свойств, но не являться при этом компонентами.

Класс *TComponent*

Класс `TComponent` является предком всех компонентов VCL. Он используется в качестве основы для создания невизуальных компонентов и реализует основные механизмы, которые обеспечивают функционирование любого компонента. В нем появляются первые свойства, которые отображаются в Инспекторе объектов. Это свойство

```
property Name: TComponentName;
```

Оно содержит имя экземпляра компонента, которое используется для идентификации компонента в приложении.

Примечание

Тип `TComponentName` представляет собой обычную строку:

```
type TComponentName = type string;
```

Свойство

```
property Tag: Longint;
```

является вспомогательным и не влияет на работу компонента. Оно отдано на откуп разработчику, который может присваивать ему значения по своему усмотрению. Например, это свойство можно использовать для дополнительной, более удобной идентификации компонентов.

Для компонентов существует своя иерархия, поэтому в классе введен механизм учета и управления компонентами, для которых данный компонент является владельцем. Свойства и методы, которые отвечают за управление, приведены в табл. 2.1.

Таблица 2.1. *Свойства и методы для управления списком компонентов*

Свойство (метод)	Описание
`property Components[Index: Integer]: TComponent;`	Содержит индексированный список указателей всех компонентов, для которых данный компонент является владельцем (`Owner`)
`property ComponentCount: Integer;`	Число подчиненных компонентов
`property Owner: TComponent;`	Указывается, какой компонент является владельцем данного
`property ComponentIndex: Integer;`	Индекс данного компонента в списке владельца

***Таблица 2.1** (окончание)*

Свойство (метод)	Описание
`procedure InsertComponent(AComponent: TComponent);`	Вставляет компонент `AComponent` в список
`procedure RemoveComponent(AComponent: TComponent);`	Удаляет компонент `AComponent` из списка
`procedure FindComponent(AName: string): TComponent;`	Осуществляет поиск компонента по имени `AName`
`procedure DestroyComponents;`	Предназначен для уничтожения всех компонентов, подчиненных данному

Очень важное свойство

```
type TComponentState = set of (csLoading, csReading, csWriting,
csDestroying, csDesigning, csAncestor, csUpdating, csFixups,
csFreeNotification, csInline, csDesignInstance);
property ComponentState: TComponentState;
```

дает представление о текущем состоянии компонента. В табл. 2.2 описаны возможные состояния компонента. Состояние может измениться в результате получения компонентом некоторого сообщения, действий разработчика, выполнения акции и т. д. Это свойство активно используется средой разработки.

***Таблица 2.2.** Возможные состояния компонента*

Состояние	Описание
`csLoading`	Устанавливается при загрузке компонента из потока
`csReading`	Устанавливается при чтении значений свойств из потока
`csWriting`	Устанавливается при записи значений свойств в поток
`csDestroying`	Устанавливается при уничтожении компонента
`csDesigning`	Состояние разработки. Устанавливается при работе с формой во время разработки
`csAncestor`	Устанавливается при переносе компонента на форму. Для перехода в это состояние должно быть уже установлено состояние `csDesigning`
`csUpdating`	Устанавливается при изменении значений свойств и отображения результата на форме-владельце. Для перехода в это состояние должно быть уже установлено состояние `csAncestor`

Таблица 2.2 *(окончание)*

Состояние	Описание
`csFixups`	Устанавливается, если компонент связан с компонентом другой формы, которая еще не загружена в среду разработки
`csFreeNotification`	Если это состояние устанавливается, другие компоненты, связанные с данным, уведомляются о его уничтожении
`csInline`	Определяет компонент верхнего уровня в иерархии. Используется для обозначения корневого объекта в разворачивающихся свойствах
`csDesignInstance`	Определяет корневой компонент на этапе разработки

Для обеспечения работы механизма действий *(см. гл. 8)* предназначен метод

```
function ExecuteAction(Action: TBasicAction): Boolean; dynamic;
```

Он вызывается автоматически при необходимости выполнить акцию, предназначенную для данного компонента.

На уровне класса `TComponent` обеспечена поддержка COM-интерфейсов `IUnknown` и `IDispatch`.

Через свойство

```
property ComObject: IUnknown;
```

вы можете обеспечить применение методов этих интерфейсов.

Таким образом, класс `TComponent` имеет все для использования в качестве предка, для создания собственных невизуальных компонентов.

Базовые классы элементов управления

Вслед за классом `TComponent` в иерархии базовых классов (см. рис. 2.1) располагается группа из трех классов, которые обеспечивают создание различных визуальных компонентов. *Визуальные компоненты* — это разнообразные стандартные для Windows и специальные (созданные разработчиками Inprise) элементы управления.

Понятно, что визуальные компоненты должны уметь отобразить себя на экране монитора и реагировать на целый ряд новых событий (реакция на мышь и клавиатуру, движение курсора и т. д.). Для этого в них встроен специальный механизм, обеспечивающий взаимодействие компонентов с графической подсистемой ОС (GUI).

Существует несколько типов элементов управления, которые существенно отличаются по своим возможностям и поведению. Каждому типу соответствует собственный класс иерархии.

Класс `TWinControl` обеспечивает использование в Delphi оконных элементов управления. Главное отличие оконного элемента управления от любых других — наличие дескриптора окна `hWnd`. *Дескриптор окна* — это специальный идентификатор, который операционная система присваивает всем объектам, которые должны обладать свойствами окна. Если элемент управления имеет дескриптор окна, то он должен уметь выполнять следующие операции:

- получать и передавать фокус управления во время выполнения приложения;
- воспринимать управляющие воздействия от мыши и клавиатуры;
- уметь размещать на себе другие элементы управления.

Оконными элементами управления являются не только формы, но и практически все стандартные элементы управления Windows: и списки, и редакторы имеют дескриптор окна.

Итак, все визуальные компоненты происходят от класса `TWinControl`. Однако нестандартные элементы управления имеют еще одного общего предка. Это класс `TCustomControl`. Он существенно облегчает использование элементов управления, т. к. позволяет управлять отрисовкой компонента путем использования специального класса `TCanvas` — так называемой канвы *(см. гл. 11)* вместо обращения к системным функциям GUI.

Для обеспечения создания обычных (не оконных) элементов управления непосредственно от класса `TControl` порожден класс `TGraphicControl`. Его потомки не могут получать фокус, но используют для визуализации канву.

Класс *TControl*

Класс `TControl` является базовым для всех визуальных компонентов и инкапсулирует механизмы отображения компонента на экране. В нем используется множество новых свойств и методов. Недаром в Delphi в Инспекторе объектов появилась категоризация методов и свойств (рис. 2.2). Большинство из них вводятся как раз в классах `TControl` и `TWinControl`.

Рассмотрим только важнейшие свойства и методы по категориям.

Группа свойств *Visual*. Местоположение и размер элемента управления

Для определения местоположения и размеров визуального компонента введены два опубликованных свойства для задания координат левого верхнего угла:

Рис. 2.2. Категории свойств визуального компонента.
Для представления их в таком виде нужно отметить флажок **By Category** в пункте меню **Arrange** всплывающего меню Инспектора объектов

```
property Top: Integer;
property Left: Integer;
```

и два опубликованных свойства для определения размеров:

```
property Height: Integer;
property Width: Integer;
```

Значения свойств задаются в пикселах. Для определения местоположения используется система координат рабочей области владельца данного компонента. Начало отсчета находится в левом верхнем углу. Оси направлены вправо и вниз. Под рабочей областью понимается та часть площади владельца (формы, панели), которая предназначена для размещения дочерних элементов. Эти свойства можно использовать как на этапе разработки, так и во время выполнения.

Свойство

```
property ClientOrigin: TPoint;
```

содержит координаты левого верхнего угла элемента управления в системе координат экрана. Координаты любой точки можно пересчитать в экранные при помощи метода

```
function ClientToScreen(const Point: TPoint): TPoint;
```

и наоборот:

```
function ScreenToClient(const Point: TPoint): TPoint;
```

Для приведения компонента в соответствие текущим значениям указанных выше свойств используется метод

```
procedure AdjustSize; dynamic;
```

Параметры рабочей области компонента определяются следующими свойствами:

- `property ClientHeight: Integer;`

  определяет высоту рабочей области в пикселах.

- `property ClientWidth: Integer;`

  определяет ширину рабочей области в пикселах.

- `property ClientRect: TRect;`

  значение которого есть не что иное, как (`0,0,.ClientWidth, ClientHeight`). Кому-то будет удобнее пользоваться этим свойством.

Если разработчику неизвестны текущие параметры рабочей области, то он может воспользоваться следующими методами.

Функция

```
function GetClientOrigin: TPoint; virtual;
```

возвращает координаты левого верхнего угла рабочей области.

Функция

```
function GetClientRect: TRect; virtual;
```

возвращает размеры прямоугольника рабочей области.

Выравнивание элемента управления

Для выравнивания компонента в рабочей области его владельца (обычно это форма) применяется свойство

```
property Align: TAlign;
```

Тип `TAlign` объявлен следующим образом:

```
type TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);
```

При значении `alNone` выравнивание отсутствует. При следующих четырех значениях выравнивание осуществляется по соответствующей стороне. Значение `alClient` приводит к тому, что элемент управления изменяет свои размеры до размеров рабочей области владельца.

Свойство

```
property Anchors: TAnchors;
type TAnchors = set of TAnchorKind;
type TAnchorKind = (akTop, akLeft, akRight, akBottom);
```

обеспечивает фиксацию элемента управления по сторонам владельца. "Якорь" можно установить по одной, двум, трем или четырем сторонам. При зада-

нии якоря по любой стороне расстояние между стороной и элементом управления сохраняется неизменным. Комбинируя якоря для сторон, можно добиться различных вариантов поведения компонента при изменении размеров владельца.

Если по вертикали или горизонтали якорь не установлен вообще, то при изменении размеров владельца компонент остается на том же месте с учетом изменившегося размера.

Если по вертикали или горизонтали установлены оба якоря, то при изменении размеров владельца размер элемента управления изменяется таким образом, чтобы расстояния до сторон владельца остались неизменными.

Свойство

```
property AutoSize: Boolean;
```

обеспечивает изменение размеров компонента в соответствии с размерами его содержимого (текста, изображения, списка, иерархического дерева и т. д.).

Внешний вид элемента управления

Для определения цвета компонента используется свойство

```
property Color: TColor;
```

При нахождении указателя мыши над компонентом его изображение может изменяться в соответствии со значением свойства

```
property Cursor: TCursor;
```

Для текста компонента шрифт обычно задается свойством

```
property Font: TFont;
```

Сложный класс `TFont`, задающий все характеристики шрифта, подробно рассматривается в *гл. 10*.

Свойство

```
property DesktopFont: Boolean;
```

определяет возможность использования шрифта для отображения, который применяется ОС для представления текста в значках.

Сам текст задается свойством

```
type TCaption = string;
property Text: TCaption;
```

Длину текста можно определить при помощи функции

```
function GetTextLen: Integer;
```

Она возвращает число символов в тексте.

Элемент управления можно сделать видимым или невидимым. Для этого применяется свойство

```
property Visible: Boolean;
```

Этих же результатов можно достичь методами `Show` (компонент видим) и `Hide` (компонент невидим).

Опубликованное свойство

```
property Hint: string;
```

содержит текст *ярлыка* — однострочной подсказки, которая появляется в маленькой рамке при остановке мыши на элементе управления.

Для управления ярлыком используется свойство

```
property ShowHint: Boolean;
```

При значении `True` ярлык начинает работать, при значении `False` ярлык выключается.

Для каждого элемента управления можно создать собственное всплывающее меню. Ссылка на экземпляр класса `TPopupMenu`, инкапсулирующего такое меню, хранится в свойстве

```
property PopupMenu: TPopupMenu;
```

Текущее состояние элемента управления определяется свойством `ControlState`:

```
type TControlState = set of (csLButtonDown, csClicked, csPalette,
csReadingState, csAlignmentNeeded, csFocusing, csCreating, csPaintCopy,
csCustomPaint, csDestroyingHandle, csDocking,);
property ControlState: TControlState;
```

Описание возможных состояний элемента управления представлено в табл. 2.3.

Таблица 2.3. *Возможные состояния элемента управления*

Состояние	Описание
`csLButtonDown`	Левая кнопка мыши нажата, но еще не отпущена. Используется для реализации события `OnMouseDown`
`csClicked`	Левая кнопка мыши нажата, но еще не отпущена. Используется для реализации события `OnClick`
`csPalette`	Состояние соответствует режиму изменения палитры. Это реакция на сообщение `WM_PALETTCHANGED`
`csReadingState`	Осуществляется чтение значений свойств из потока (см. табл. 5.1)
`csAlignmentNeeded`	Осуществляется выравнивание компонента

Таблица 2.3 *(окончание)*

Состояние	Описание
csFocusing	Элемент управления получает фокус
csCreating	Элемент управления и его дочерние элементы создаются
csPaintCopy	Отрисовывается копия элемента управления
csCustomPaint	Элемент управления выполняет нестандартные операции отрисовки, заданные разработчиком
csDestroyingHandle	Указатель на объект элемента управления уничтожается
csDocking	Элемент управления находится в режиме присоединения

В зависимости от совокупности установленных свойств, элемент управления может соответствовать одному из возможных стилей, который задается свойством

```
type TControlStyle = set of (csAcceptsControls, csCaptureMouse,
csDesignInteractive, csClickEvents, csFramed, csSetCaption, csOpaque,
csDoubleClicks, csFixedWidth, csFixedHeight, csNoDesignVisible,
csReplicatable, csNoStdEvents, csDisplayDragImage, csReflector,
csActionClient, csMenuEvents);
property ControlStyle: TControlStyle;
```

Доступность элемента управления в целом определяется свойством

```
property Enabled: Boolean;
```

При значении `True` элемент управления полностью работоспособен. При значении `False` элемент управления неактивен и отображается серым цветом.

Для получения контекста устройства `HDC` элемента управления используется метод

```
function GetDeviceContext(var WindowHandle: HWnd): HDC; virtual;
```

Набор свойств и методов класса `TWinControl` обеспечивает функционирование механизма перетаскивания (Drag-and-Drop) и механизма присоединения (Drag-and-Dock).

Связь с родительским элементом управления

Механизм связывания визуального компонента с родительским компонентом (владельцем) позволяет автоматически задавать для нового элемента управления некоторые свойства, отвечающие за его внешний вид *(см. выше)*. В результате все дочерние элементы управления для одного родительского (формы, панели) будут выглядеть одинаково оформленными.

Родительский компонент задается свойством

```
property Parent: TWinControl;
```

Для каждого дочернего элемента управления можно задать значения нескольких свойств:

```
property ParentBiDiMode: Boolean;
property ParentColor: Boolean;
property ParentFont: Boolean;
property ParentShowHint: Boolean;
```

Каждое из них управляет одной характеристикой визуализации элемента.

Метод

```
function HasParent: Boolean; override;
```

используется для того, чтобы определить, имеется ли у компонента владелец вообще.

В классе `TControl` впервые появляются методы-обработчики событий, которые обеспечивают передачу в элемент действия мыши, присоединение и перетаскивание.

Класс *TWinControl*

Класс `TWinControl` обеспечивает создание и использование оконных элементов управления *(см. выше)*. Напомним, что оконный элемент управления имеет системный дескриптор окна `hWnd`. Однако оконными элементами являются не только формы и диалоги, но и большинство стандартных элементов управления.

Новые механизмы, инкапсулированные в классе, обеспечивают выполнение характерных для оконных элементов функций: прием и передачу фокуса, отклик на действия мышью и ввод с клавиатуры и т. д. Рассмотрим основные свойства и методы класса.

Дескриптор окна содержится в свойстве

```
property Handle: HWND;
```

При создании оконного элемента управления вызывается метод

```
procedure CreateParams(var Params: TCreateParams); virtual;
```

который заполняет структуру `TCreateParams` необходимыми значениями:

```
type

  TCreateParams = record
    Caption: PChar;
    Style: DWORD;
```

```
    ExStyle: DWORD;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;

  end;
```

Для создания дескриптора окна для элемента управления используется метод

```
procedure CreateHandle; virtual;
```

Операционная система создает дескриптор окна только вместе с самим окном. Поэтому метод `CreateHandle` только создает окно, а для присваивания свойству `Handle` значения дескриптора окна вызывает метод `CreateWnd`.

Для передачи фокуса между элементами управления на одной форме часто используется клавиша <Tab>. Порядок перемещения фокуса между элементами определяется свойством

```
type TTabOrder = -1..32767;
property TabOrder: TTabOrder;
```

В первую очередь фокус передается компоненту с минимальным значением свойства. Далее — по возрастанию значения. При переносе компонента на форму это значение задается автоматически в соответствии с числом компонентов на форме.

Компонент можно заставить не откликаться на клавишу <Tab>. Для этого свойству

```
property TabStop: Boolean;
```

необходимо присвоить значение `False`.

Для передачи фокуса прямо элементу управления применяется метод

```
procedure SetFocus; virtual;
```

Чтобы узнать, имеет ли элемент управления фокус, в настоящее время используется метод

```
function Focused: Boolean; dynamic;
```

Все оконные элементы имеют рамку по контуру (впрочем, она может быть не видна). Ее параметры задаются группой свойств:

- `property BevelEdges: TBevelEdges;`

  задает, какие стороны входят в рамку;

- `property BevelInner: TBevelCut;`
  `property BevelOuter: TBevelCut;`

  задают внешний вид рамки;

- `property BevelKind: TBevelKind;`

  определяет стиль рамки;

- `property BevelWidth: TBevelWidth;`

  задает размер рамки.

Свойство

```
property Brush: TBrush;
```

определяет параметры кисти (цвет и заполнение), которой рисуется фон элемента.

Оконный элемент может содержать другие компоненты. Для управления ими применяется индексированный список указателей, представляющих свойство

```
property Controls[Index: Integer]: TControl;
```

Общее число дочерних элементов управления содержится в свойстве

```
property ControlCount: Integer;
```

Внешний вид оконного элемента определяется свойством

```
property Ctl3D: Boolean
```

При значении `True` элемент управления имеет трехмерный вид. Иначе элемент выглядит плоским.

Для вызова темы контекстной помощи для конкретного элемента управления предназначено свойство

```
type THelpContext = -MaxLongInt..MaxLongInt;
property HelpContext: THelpContext;
```

Значение свойства должно соответствовать номеру темы в файле помощи.

В классе `TWinControl` добавлена возможность использования редакторов способа ввода (Input Method Editor, IME). Такие редакторы позволяют приспособить стандартную раскладку клавиатуры для символьных языков для ввода нестандартных символов (иероглифов и т. д.). Редакторы IME представляют собой специально устанавливаемое в операционной системе программное обеспечение (ПО). Имя такого редактора задается в свойстве `ImeName`. Режим работы редактора определяется свойством `ImeMode`.

В классе `TWinControl` добавлено еще несколько методов-обработчиков событий, обеспечивающих реакцию на ввод с клавиатуры, получение и потерю фокуса.

Класс *TCustomControl*

Класс `TCustomControl` предназначен для создания на его основе нестандартных оконных элементов управления. Процесс визуализации в нем упрощен за счет использования специального класса `TCanvas`, инкапсулирующего канву *(см. гл. 11)*.

Доступ к канве осуществляется через свойство

```
property Canvas: TCanvas;
```

Отрисовка элемента управления осуществляется методом

```
procedure PaintWindow(DC: HDC); override;
```

после получения сообщения `WM_PAINT`.

Возможности этого класса унаследовали классы `TPanel`, `TGroupBox`, `TStringGrid` и т. д.

Класс *TGraphicControl*

Класс `TGraphicControl` предназначен для создания на его основе визуальных компонентов, не получающих фокус в процессе выполнения приложения. Так как непосредственным предком класса является класс `TControl`, то потомки `TGraphicControl` умеют реагировать на управляющие воздействия мышью.

Наглядный пример элемента управления, которому не нужно получать фокус, — это компонент `TLabel`, предназначенный для отображения текста, или компонент `TImage`, предназначенный для визуализации изображений.

Для визуализации элементов управления на основе этого класса используется канва, инкапсулированная в классе `TCanvas`.

Доступ к канве осуществляется через свойство

```
property Canvas: TCanvas;
```

Отрисовка элемента управления осуществляется методом

```
procedure PaintWindow(DC: HDC); override;
```

после получения сообщения `WM_PAINT`.

Резюме

В настоящей главе рассмотрены важнейшие свойства и методы базовых классов, лежащих в основе VCL. Зная возможности этих классов, вы тем самым будете знать назначение многих свойств и методов в компонентах VCL. Поэтому материал данной главы будет полезен при изучении любых вопросов, рассматриваемых в этой книге в дальнейшем.

ГЛАВА 3

Обработка исключительных ситуаций

Любому, кто писал более или менее сложные программы, интуитивно ясно, что такое обработка исключительных ситуаций (ИС). Всякое взаимодействие с операционной системой на предмет получения ресурсов — места на диске, в памяти, открытие файла — может завершиться неудачно. Любое вычисление может закончиться делением на ноль или переполнением. Дополнительный фактор возникновения исключительных ситуаций содержится в данных, к которым могут обращаться программы. Особенно это актуально в приложениях баз данных.

Платой за надежную работу программы в таких условиях служит введение многочисленных проверок, способных предотвратить некорректные действия в случае возникновения нештатной ситуации. Хорошо, если в конце очередной конструкции `if..then` можно просто поставить оператор `Exit`. Обычно же для корректного выхода из ситуации нужно отменить целую последовательность действий, предшествующих неудачному. Все это сильно запутывает программу, маскируя четкую структуру главного алгоритма.

При разработке приложений в Delphi программист имеет возможность использовать несколько механизмов, обеспечивающих обработку исключительных ситуаций. Это и специальные операторы языка Object Pascal, и классы, предназначенные для программирования реакции на ошибки.

Поэтому эта глава посвящена... нет, не тому, как писать безошибочно; а тому, как защищать приложения от воздействия неизбежно возникающих ошибок.

Исключительная ситуация как класс

Что же такое исключительная ситуация? Интуитивно понятно, что это — некое нештатное событие, могущее повлиять на дальнейшее выполнение программы. Если вы ранее писали в среде Turbo Pascal или подобной, то вы

наверняка пытались избежать таких ситуаций, вводя многочисленные проверки данных и кодов возврата функций. От этого громоздкого кода можно раз и навсегда избавиться, взяв на вооружение механизм, реализованный в Delphi.

Компилятор Delphi генерирует код, который перехватывает любое такое нештатное событие, сохраняет необходимые данные о состоянии программы, и выдает разработчику... Что можно выдать в объектно-ориентированном языке программирования? Конечно же, объект. *С точки зрения Object Pascal исключительная ситуация — это объект.*

Вы можете получить и обработать этот объект, предусмотрев в программе специальную языковую конструкцию (`try..except`). Если такая конструкция не предусмотрена, все равно исключение будет обработано — в недрах VCL есть соответствующие обработчики, окружающие все потенциально опасные места.

Чем же различаются между собой исключительные ситуации? Как отличить одну исключительную ситуацию от другой? Поскольку это объекты, они отличаются классом (объектным типом). В модуле SYSUTILS.PAS описан объектный тип `Exception`. Он является предком для всех других объектов — исключительных ситуаций. Вот он:

```
Exception = class(TObject)
  private
    FMessage: string;
    FHelpContext: Integer;
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Integer); overload;
    constructor CreateRes(ResStringRec: PResStringRec); overload;
    constructor CreateResFmt(Ident: Integer; const Args: array of const);
      overload;
    constructor CreateResFmt(ResStringRec: PResStringRec; const Args:
                             array of const); overload;
    constructor CreateHelp(const Msg: string; AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string; const Args:
                                                 array of const;
      AHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer; AHelpContext: Integer);
      overload;
    constructor CreateResHelp(ResStringRec: PResStringRec;
      AHelpContext: Integer); overload;
    constructor CreateResFmtHelp(ResStringRec: PResStringRec;
          const Args: array of const; AHelpContext: Integer); overload;
```

```
    constructor CreateResFmtHelp(Ident: Integer;
          const Args: array of const; AHelpContext: Integer); overload;
    property HelpContext: Integer read FHelpContext write FHelpContext;
    property Message: string read FMessage write FMessage;
  end;

  ExceptClass = class of Exception;
```

Как видно из приведенного описания класса `Exception`, у него имеется двенадцать (!) конструкторов, позволяющих задействовать при создании объекта текстовые строки из ресурсов приложения (имя включает строку `Res`), форматирование текста (включает `Fmt`), связь с контекстом справочной системы (включает `Help`).

Конструкторы, в названии которых встречается подстрока `Fmt`, могут вставлять в формируемый текст сообщения об ошибке значения параметров, как это делает стандартная функция `Format`:

```
If MemSize > Limit then
 raise EOutOfMemory.CreateFmt('Cannot allocate more than %d
bytes',[Limit]);
```

Если в названии присутствует подстрока `Res`, это означает, что текст сообщения будет загружаться из ресурсов приложения. Это особенно полезно при создании локализованных версий программных продуктов, когда нужно сменить язык всех сообщений, ничего не компилируя заново.

И наконец, если в названии фигурирует подстрока `Help`, то такой конструктор инициализирует свойство `HelpContext` создаваемого объекта. Естественно, система помощи должна быть создана и в ней должна иметься статья, связанная с этим контекстом. Теперь пользователь может затребовать помощь для данной ситуации, скажем, нажав клавишу <F1> в момент показа сообщения об ИС.

Тип `Exception` порождает многочисленные дочерние типы, соответствующие часто встречающимся случаям ошибок ввода/вывода, распределения памяти и т. п. Дерево исключительных ситуаций Delphi 7 приведено на рис. 3.1.

Заметим, что тип `Exception` и его потомки представляют собой исключение из правила, предписывающего все объектные типы именовать с буквы T. Потомки `Exception` начинаются с E, например `EZeroDivide`.

Для экономии места потомки нескольких важных объектов не показаны. Ниже приведены табл. 3.1—3.3, содержащие описания этих групп исключительных ситуаций.

Вы можете самостоятельно инициировать исключительную ситуацию при выполнении тех или иных действий. Но, хотя синтаксис конструктора объекта `Exception` похож на конструкторы всех других объектов, создается он по-особенному.

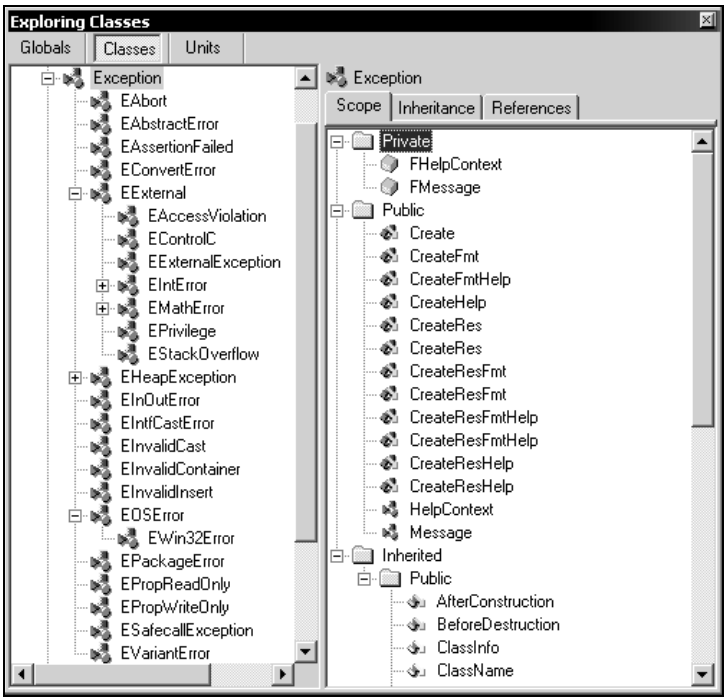


Рис. 3.1. Дерево объектов исключительных ситуаций Delphi 7

Таблица 3.1. *Исключительные ситуации при работе с памятью (порождены от* `EHeapException`*)*

Тип	Условие возникновения
`EOutOfMemory`	Недостаточно места в куче (памяти)
`EOutOfResources`	Нехватка системных ресурсов
`EInvalidPointer`	Недопустимый указатель (обычно `nil`)

Таблица 3.2. *Исключительные ситуации целочисленной математики (порождены от* `EIntError`*)*

Тип	Условие возникновения
`EDivByZero`	Попытка деления на ноль (целое число)
`ERangeError`	Число или выражение выходит за допустимый диапазон
`EIntOverflow`	Целочисленное переполнение

Таблица 3.3. *Исключительные ситуации математики с плавающей точкой (порождены от* `EMathError`*)*

Тип	Условие возникновения
`EInvalidOp`	Неверная операция
`EZeroDivide`	Попытка деления на ноль
`EOverflow`	Переполнение с плавающей точкой
`EUnderflow`	Исчезновение порядка
`EInvalidArgument`	Неверный аргумент математических функций

Для этого используется оператор `raise`, за которым в качестве параметра должен идти экземпляр объекта типа `Exception`. Обычно сразу за оператором следует конструктор класса ИС:

```
raise EMathError.Create('');
```

но можно и разделить создание и возбуждение исключительной ситуации:

```
var E: EMathError;
begin
  E := EMathError.Create('');
  raise E;
end;
```

Оператор `raise` передает созданную исключительную ситуацию ближайшему блоку `try..except` *(см. ниже)*.

```
if C = 0 then
raise EDivByZero.Create('Деление на ноль')
else
A := B/C;
```

Самостоятельная инициализация ИС может пригодиться при программировании реакции приложения на ввод данных, для контроля значений переменных и т. д. В таких случаях желательно создавать собственные классы ИС, специально приспособленные для ваших нужд. Также полезно использовать специально спроектированные исключительные ситуации при создании собственных объектов и компонентов. Так, многие важнейшие классы VCL — списки, потоки, графические объекты — сигнализируют о своих (или ваших?) проблемах созданием соответствующей ИС — `EListError`, `EInvalidGraphic`, `EPrinter` и т. д.

Самый важный отличительный признак объекта `Exception` — это все же класс, к которому он принадлежит. Именно факт принадлежности возник-

шей ИС к тому или иному классу говорит о том, что случилось. Если же нужно детализировать проблему, можно присвоить значение свойству `Message`. Если и этого мало, можно добавить в объект новые поля. Так, в ИС `EInOutError` (ошибка ввода/вывода) есть поле `ErrorCode`, значение которого соответствует произошедшей ошибке — запрету записи, отсутствию или повреждению файла и т. д.

```
try
.FileOpen('c:\myfile.txt', fmOpenWrite);
except
 on E: EInOutError do
 case E.ErrorCode of
  ERROR_FILE_NOT_FOUND {=2}: ShowMessage('Файл не найден !');
  ERROR_ACCESS_DENIED {=5}:  ShowMessage('Доступ запрещен!');
  ERROR_DISK_FULL {=112}:    ShowMessage('Диск переполнен!');
 end;
end;
```

Впрочем, ИС `EInOutError` возникают только тогда, когда установлена опция компилятора `{$IOCHECKS ON}` (или иначе `{$I+}`). В противном случае проверку переменной `IOResult` (известной еще по Turbo Pascal) нужно делать самому.

Еще более "продвинутый" пример — ИС `EDBEngineError`. Она имеет свойства `ErrorCount` и свойство-массив `Errors`: одна операция с базой данных может породить сразу несколько ошибок.

Защитные конструкции языка Object Pascal

Для работы с объектами исключительных ситуаций существуют специальные конструкции языка Object Pascal — блоки `try..except` и `try..finally`. Они контролируют выполнение операторов, помещенных внутри блока до ключевого слова `except` или `finally`. В случае возникновения исключительной ситуации штатное выполнение вашей программы немедленно прекращается, и управление передается операторам, идущим за указанными ключевыми словами. Если в вашей процедуре эти блоки отсутствуют, управление все равно будет передано ближайшему блоку, внутри которого возникла ситуация. А уж внутри VCL их предостаточно.

Хотя синтаксис двух видов блоков похож, но они *принципиально* отличаются назначением и решаемыми задачами. Поэтому вопрос, какой из них выбрать, не стоит: чтение нижеследующего материала, надеемся, убедит вас в пользе обоих.

Блок *try..except*

Для реакции на конкретный тип ситуации применяется блок `try..except`. Синтаксис его следующий:

```
try
 <Оператор>
 <Оператор>
...
 except
  on EException1 do  < Оператор обработки ИС типа EException1 >;
  on EException2 do  < Оператор >;
 ...
  else { }
  <Оператор> {обработчик прочих ИС}
end;
```

Выполнение блока начинается с секции `try`. При отсутствии исключительных ситуаций только она и выполняется. Секция `except` получает управление в случае возникновения ИС. После обработки происходит выход из защищенного блока, и управление обратно в секцию `try` не передается; выполняются операторы, стоящие после `end`.

Если вы хотите обработать любую ИС одинаково, независимо от ее класса, вы можете писать код прямо между операторами `except` и `end`. Но если обработка отличается, здесь можно применять набор директив `on..do`, определяющих реакцию приложения на определенную ситуацию. Каждая директива связывает ситуацию (`on...`), заданную своим именем класса, с группой операторов (`do...`).

```
U := 220.0;
R := 0;
try
  I := U / R;
except
  on EZeroDivide do MessageBox('Короткое замыкание!');
end;
```

В этом примере замена `if..then` на `try..except`, может быть, не дала очевидной экономии кода. Однако если при решении, допустим, вычислительной задачи проверять на возможное деление на ноль приходится не один, а множество раз, то выигрыш от нового подхода неоспорим — достаточно одного блока `try..except` на все вычисления.

При возникновении ИС директивы просматриваются последовательно, в порядке их описания. Каждый тип исключительной ситуации, описанный

после ключевого слова `on`, обрабатывается именно этим блоком: только то, что предусмотрено в нем, и будет являться реакцией на данную ситуацию.

Если при этом обработчик родительского класса стоит перед дочерним, последний никогда не получит управления.

```
try
 i:=1;j:=0;
 k:=i div j;
...
except
 on EIntError do ShowMessage('IntError');
 on EDivByZero do ShowMessage('DivByZero');
 end;
```

В этом примере, хотя в действительности будет иметь место деление на ноль (`EDivByZero`), вы увидите сообщение, соответствующее родительскому классу `EIntError`. Но стоит поменять две конструкции `on..do` местами, и все придет в норму.

Если возникла ситуация, не определенная ни в одной из директив, выполняются те операторы, которые стоят после `else`. Если и их нет, то ИС считается не обработанной и будет передана на следующий уровень обработки. Этим следующим уровнем может быть другой оператор `try..except`, который содержит в себе данный блок.

Примечание

Мы детально обсудили, что и как помещают в блок `try..except`. Но в ряде случаев можно... не размещать там ничего. Пустой блок применяют, когда хотят просто проигнорировать возникновение некоторых ситуаций. Скажем, у вас в программе предусмотрена некая обработка данных, которая может завершиться неудачно, но это не повлияет на дальнейшую работу, и пользователь может об этом не знать. В этой ситуации вполне уместно изолировать ее в пустом блоке `try..except`. Важно только не поместить туда больше кода, чем нужно — иначе "с водой можно выплеснуть и ребенка".

Если вы не предусмотрели блоков обработки ИС в своем коде, это не должно привести к аварийному завершению всего приложения. Все места в VCL, где управление передается коду разработчика (в том числе, конечно, все обработчики событий всех компонентов), заключены в такие блоки. Но, увы, в Borland не знают о конкретных проблемах вашей программы, и максимум, что они могут сделать для вас, — это проинформировать о типе и месте возникновения ИС. Стандартная обработка подразумевает вывод на экран панели текстового сообщения (из свойства `Exception.Message`) с указанием типа ошибки. Можно получить и развернутую информацию с именем модуля и адреса, где она имела место (рис. 3.2).

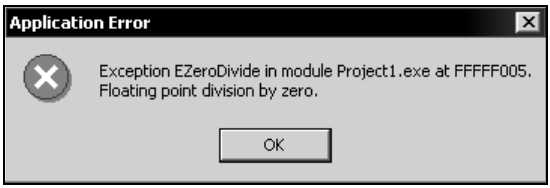


Рис. 3.2. Типовое окно сообщения об ошибке

Для этого нужно вызвать процедуру

```
procedure ShowException(ExceptObject: TObject; ExceptAddr: Pointer);
```

имеющуюся в модуле SYSUTILS.PAS.

Если предусмотренной вами обработки ИС недостаточно, то можно продолжить ее дальше программно при помощи оператора `raise`.

Этот оператор уже встречался нам при описании создания пользовательских ИС. Там за ним следовал вызов конструктора ИС. Здесь же конструктор опущен: возбуждаться будет уже существующий объект ИС, приведший нас в блок:

```
...
sl:= TStringList.Create;
 try
   sl.LoadFromFile(AFileName);
except
   sl.Free;
   raise;
 end;
...
```

В этом примере в случае возникновения исключительной ситуации созданный список строк должен быть уничтожен. Сама же обработка предоставляется "вышестоящим инстанциям".

Блок *try..finally*

Параллельно с блоком `try..except` в языке существует и `try..finally`. Он соответствует случаю, когда необходимо возвратить выделенные программе ресурсы даже в случае аварийной ситуации. Синтаксис блока `try..finally` таков:

```
try
  <Оператор>
  <Оператор>
...
```

```
 finally
  <Оператор>
 ...
end;
```

Смысл этой конструкции можно описать одним предложением: операторы, стоящие после `finally`, выполняются *всегда*.

Следующие за `try` операторы исполняются в обычном порядке. Если за это время не возникло никаких ИС, далее следуют те операторы, которые стоят после `finally`. В случае, если между `try` и `finally` произошла ИС, управление немедленно передается на операторы после `finally`, которые называются *кодом очистки*. Допустим, вы поместили после `try` операторы, которые должны выделить вам ресурсы системы (дескрипторы блоков памяти, файлов, контекстов устройств и т. п.). Тогда операторы, освобождающие их, следует поместить после `finally`, и ресурсы будут освобождены в любом случае. Блок `try..finally`, как можно догадаться, еще называется *блоком защиты ресурсов*.

Важно обратить внимание на такой факт: данная конструкция ничего не делает с самим объектом — исключительной ситуацией. Задача `try..finally` — только прореагировать на факт нештатного поведения программы и проделать определенные действия. Сама же ИС продолжает "путешествие" и вопрос ее обработки остается на повестке дня.

Блоки защиты ресурсов и обработчики ИС, как и другие блоки, могут быть вложенными. В этом простейшем примере каждый вид ресурсов системы защищается в отдельном блоке:

```
try
 Allocate1stResource;
 try
  Allocate2ndResource;
  SolveProblem;
 finally
  Free2ndResource;
 end;
finally
 Free1stResource;
end;
```

Можно также вкладывать обработчики друг в друга, предусмотрев в каждом специфическую реакцию на ту или иную ошибку:

```
var i,j,k : Integer;
begin
```

```
i := Round(Random);
j := 1 - i;
try
  k := 1 div i;
  try
    k := 1 div j;
  except
    On EDivByZero do
     ShowMessage('Вариант 1: j=0');
  end;
except
  On EDivByZero do
     ShowMessage('Вариант 2: i=0');
end;
end;
```

Но все же идеально правильный случай — это сочетание блоков двух типов. В один из них помещается общее (освобождение ресурсов в `finally`), в другой — особенное (конкретная реакция внутри `except`).

Использование исключительных ситуаций

Если произошла ошибка и возбуждена исключительная ситуация, то она будет обрабатываться по такому алгоритму:

1. Если ситуация возникла внутри блока `try..except`, то там она и будет обработана. Если ИС "продвинута" дальше при помощи оператора `raise`, а также если она возникла в блоке `try..finally`, обработка продолжается.
2. Если программистом определен обработчик события `Application.OnException`, то он получит управление. Обработчик объявлен следующим образом:

   ```
   TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
   ```

3. Если программист никак не определил реакцию на ИС, то будет вызван стандартный метод `ShowException`, который сообщит о классе и месте возникновения исключительной ситуации.

Пункты 2 и 3 реализуются в методе `TApplication.HandleException`. Собственно, выглядят они следующим образом:

```
if not (ExceptObject is EAbort) then
 if Assigned(FOnException) then
  FOnException(Sender, Exception(ExceptObject))
```

```
 else
  ShowException(Exception(ExceptObject));
```

Обработчик `OnException` нужен, если требуется выполнять одно и то же действие в любой исключительной ситуации, возникшей в вашем приложении. К примеру, назвать себя, указать координаты для обращения или предупредить, что это еще бета-версия.

```
program Project1;

uses
  Forms,
  SysUtils, //добавлено вручную — там описан класс Exception
  Dialogs,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}
 type
  TExceptClass = class
  public
  procedure GlobalExceptionHandler(Sender: TObject; E:Exception);
  end;

  procedure TExceptClass.GlobalExceptionHandler(Sender: TObject;
    E:Exception);
  begin
   ShowMessage('Произошла исключительная ситуация ' + E.ClassName
   + ': ' + E.Message
   + #13#10'Свяжитесь с разработчиками по тел. 222-33-44');

  end;
begin
  with TExceptClass.Create do
  begin
   Application.OnException := GlobalExceptionHandler;
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.Run;
   Free;
  end;
end.
```

Здесь класс `TExceptClass` создается только для того, чтобы быть носителем метода `GlobalException`. Обработчик любого события — метод, и он должен относиться к какому-либо объекту. Поскольку он здесь нужен еще до инициализации форм приложения и других его составных частей, то и объект класса `TExceptClass` создается первым. Теперь пользователь знает, что бла-

годарить за неожиданности нужно по указанному в сообщении об ошибке телефону разработчиков.

Примечание

Есть и более простой способ присвоить обработчик событию `Application.OnException`. Для этого поместите на форму компонент типа `TApplicationEvents` (страница **Additional** Палитры компонентов), роль которого — предоставление "визуального" доступа к свойствам невизуального объекта `TApplication`. Среди его событий есть и `OnException`.

Но как "пощупать" переданный при исключительной ситуации объект? Обычная конструкция

```
on EExceptionType do...
```

указывает на класс объекта, но не на конкретный экземпляр. Если во время обработки требуется доступ к свойствам этого экземпляра, его нужно поименовать внутри `on..do`, указав перед именем класса некий идентификатор:

```
on EZD: EZeroDivide do EZD.Message := 'Деление на ноль!';
```

Здесь возникшее исключение выступает под именем `EZD`. Можно изменить его свойства и отправить дальше:

```
var  APtr : Pointer;
    Form1 : TForm;
try
 APtr := Form1;
 with TObject(APtr) as TBitmap do;
except
  on EZD: EInvalidCast do EZD.Message :=. EZD.Message + 'xa-xa!';
     Raise;{ теперь обработка будет сделана в другом месте }
end;
```

Но как поименовать исключительную ситуацию, не попавшую ни в одну из директив `on..do`? Или, может быть, в вашем обработчике вообще нет `on..do`, а поработать с объектом надо? Описанный выше путь здесь не подходит. Для этих случаев есть пара системных функций `ExceptObject` и `ExceptAddr`. К сожалению, эти функции инициализируются только внутри конструкции `try..except`; в `try..finally` работать с объектом — исключительной ситуацией не представляется возможным.

Протоколирование исключительных ситуаций

Часто нужно иметь подробный материал для анализа причин возникновения ИС. Разумно было бы записывать все данные о них в файл, чтобы потом прогнозировать ситуацию. Такой подход важен для программ, которые

так или иначе будут отчуждены от разработчика: в случае возникновения непредвиденной ситуации это позволит ответить на вопросы "кто виноват?" и "что делать?". В следующем примере предложен вариант реализации протоколирования ИС.

```
const LogName : string = 'c:\appexc.log';
procedure LogException;
var fs: TFileStream; m : word;buf : array[0..511] of char;
begin
 if FileExists(LogName) then m := fmOpenReadWrite else m := fmCreate;
 fs := TFileStream.Create(LogName,m);
 fs.Seek(0,soFromEnd);
 StrPCopy(Buf,DateTimeToStr(Now)+'. ');
 ExceptionErrorMessage
(ExceptObject,ExceptAddr,@buf[StrLen(buf)],SizeOf(Buf)-StrLen(buf));
 StrCat(Buf,#13#10);
 fs.WriteBuffer(Buf, StrLen(buf));
 fs.Free;
end;

procedure TForm1.Button1Click(Sender: TObject);
var x,y,z: real;
begin
try
 try
 x:=1.0;y:=0.0;
 z := x/y;
 except
  LogException;
  raise;
 end;
except
 on E:EIntError do ShowMessage('IntError');
 on E:EMathError do ShowMessage('MathError');
end;
end;
```

Здесь задачу записи информации об ИС решает процедура `LogException`. Она открывает файловый поток и пишет туда информацию, отформатированную при помощи уже упоминавшейся функции `ExceptionErrorMessage`. В качестве ее параметров выступают значения функций `ExceptObject` и `ExceptAddr`. К сформированной строке добавляется время возникновения ИС. Для каждого защищаемого блока кода создаются две вложенные конструкции `try..except`. Первая, внутренняя — для вас; в ней ИС протоколиру-

ется и продвигается дальше. Внешняя — для пользователя; именно в ней проводится анализ типа ИС и готовится сообщение.

В Object Pascal существует и расширенный вариант употребления оператора raise:

```
raise <экземпляр объекта типа Exception> [at <адрес>]
```

Естественно, объектный тип должен быть порожден от Exception. То, что в таком типе ничего не переопределено, не столь важно — главное, что в обработчике ИС можно отследить именно этот тип.

```
ELoginError = class(Exception);
  If LoginAttemptsNo > MaxAttempts then raise ELoginError.Create('Ошибка
  регистрации пользователя');
```

Конструкция at <адрес> используется для того, чтобы изменить адрес, к которому привязывается возникшая ИС, в пределах одного блока обработки ИС.

Коды ошибок в исключительных ситуациях

Если ваше приложение уже готовится к продаже, если вы планируете его техническую поддержку, то пора задуматься о присвоении числовых кодов ошибкам, возникающим в нем. Сообщение типа "Exception EZeroDivide in module MyNiceProgram at addr $0781BAB0" годится для разработчика, пользователя же оно повергнет в полный ступор. Если он позвонит в вашу службу техподдержки, то, скорее всего, не сможет ничего объяснить. Гораздо грамотнее дать ему уже "разжеванную" информацию и, в том числе, числовой код.

Один из путей решения этой проблемы — размещение сообщений об ошибках в ресурсах программы. Если же вы еще делаете и несколько национальных версий программы на разных языках, то этот путь — единственный.

"Классический" способ поместить текст в файл ресурсов — 3-этапный:

1. Создается исходный файл ресурсов с расширением rc, в который помещаются необходимые строки с нужными номерами.
2. Файл обрабатывается компилятором ресурсов brcc32.exe (находится в папке bin в структуре папок Delphi). На выходе образуется одноименный файл с расширением res.
3. Файл включается в программу указанием директивы $R, например {$R mystrings.res}.

Чтобы совместно использовать константы-номера ошибок в файле ресурсов и в коде на Delphi, вынесем их в отдельный включаемый файл с расширением inc:

```
const
 IOError = 1000;
 FileOpenError =  IOError + 1;
 FileSaveError =  IOError + 2;
 InternetError = 2000;
 NoConnectionError = InternetError + 1;
 ConnectionAbortedError = InternetError + 2;
```

Взглянув на файл, вы увидите, что ошибки в нем сгруппированы по категориям. Советуем вам поступить так же, разделив константы категорий промежутком в 1000 или даже 10 000.

Сам файл ресурсов может выглядеть так:

```
#include "strids.inc"
STRINGTABLE
 {
 FileOpenError, "File Open Error"
 FileSaveError, "File Save Error"
 NoConnectionError, "No Connection"
 ConnectionAbortedError, "Connection Aborted"
 }
```

"Вытащить" строку из ресурсов можно несколькими способами, но самый простой из них — просто по числовому идентификатору, переданному в функцию `LoadStr` (модуль SysUtils). Код

```
ShowMessage(LoadStr(NoConnectionError));
```

покажет сообщение "`No Connection`".

Если же строка используется при возбуждении ИС, то место идентификатору — в перекрываемом конструкторе `Exception.CreateRes`, один из вариантов которого работает подобно функции `LoadStr`:

```
if FileOpen('c:\myfile.txt", fmOpenRead) = INVALID_HANDLE_VALUE then
 raise EMyException.CreateRes(FileOpenError);
```

Таким образом, решена половина проблемы: возможным исключительным ситуациям присвоены номера, им в соответствие поставлен текст. Теперь о второй половине — как в обработчике ИС этот номер использовать.

Ясно, что нужно объявить свой класс ИС, включающий в себя свойство-код ошибки.

```
EExceptionWithCode = class(Exception)
 private
  FErrCode : Integer;
```

```
 public
  constructor CreateResCode(ResStringRec: PResStringRec);
  property ErrCode: Integer read FErrCode write FErrCode;
 end;
```

Тогда любой обработчик сможет к нему обратиться:

```
if E is EExceptionWithCode then
  ShowMessage('Error code: '+ IntToStr(EExceptionWithCode(E).ErrCode) +
  #13#10
  + 'Error text: ' + E.Message);
```

Присвоить свойству `ErrCode` значение можно двумя способами:

1. Добавить к классу ИС еще один конструктор, содержащий код в качестве дополнительного параметра:

```
constructor EExceptionWithCode.CreateResCode(Ident: Integer);
begin
 FErrCode := Ident;
 inherited CreateRes(Ident);
end;
```

2. Присвоить значение свойства в промежутке между созданием объекта ИС и его возбуждением:

```
var E: EExceptionWithCode;
begin
 E := EExceptionWithCode.CreateRes(NoConnectionError);
 E.ErrCode := NoConnectionError;
 Raise E;
end;
```

Вот, казалось бы, последний штрих. Но как быть тем, кто заранее не заготовил файл ресурсов, а работает со строками, описанными в PAS-файлах? Если вы используете оператор `resourcestring`, то помочь вам можно.

Начнем с рассмотрения ключевого слова `resourcestring`. Вслед за ним описываются текстовые константы. Но, в отличие от ключевого слова `const`, эти константы размещаются не в сегменте данных программы, а в ресурсах, и подгружаются оттуда по мере необходимости. Каждая такая константа воспринимается и обрабатывается как обычная строка. Но за каждой из них на самом деле стоит такая структура:

```
PResStringRec = ^TResStringRec;
  TResStringRec = packed record
    Module: ^Cardinal;
    Identifier: Integer;
  end;
```

Если вы еще раз посмотрите на список конструкторов объекта `Exception`, вы увидите, что те из них, которые работают с ресурсами, имеют перегружаемую версию с параметром типа `pResStringRec`. Вы угадали правильно: они — для строк из `resourcestring`. А взглянув на приведенную выше структуру, вы увидите в ней поле `Identifier`. Это то, что нам надо.

Чтобы у программиста, пользующегося `resourcestring`, голова не болела об уникальных идентификаторах ресурсных строк, среда Delphi берет на себя заботу об этом. Номера назначаются компилятором, начиная от 65 535 (`SmallInt(-1)`) и ниже (если рассматривать номер как тип `SmallInt`, то выше): 65 534, 65 533 и т. п. Сначала в этом списке идут несколько сотен resourcestring-констант, описанных в VCL (из модулей, чье имя заканчивается на `const` или `consts`: `SysConst`, `DBConsts` и т. п.). Затем очередь доходит до пользовательских констант (рис. 3.3).

С одной стороны, отсутствие лишних забот — это большой плюс; с другой стороны, разработчик не может задать строкам те номера, какие хочет.

Все остальное почти ничем не отличается от работы с "самодельными" ресурсами. Так выглядит перегружаемая версия конструктора нашего объекта `EExceptionWithCode`:

```
constructor EExceptionWithCode.CreateResCode(ResStringRec:
PResStringRec);
begin
 FErrCode := ResStringRec^.Identifier;
 inherited CreateRes(ResStringRec);
end;
```

А так — возбуждение самой ИС:

```
resourcestring
 sError1 = 'Error 1';
..
Raise EExceptionWithCode.CreateResCode(PResStringRec(@sError1));
```

Результат обработки показан на рис. 3.3.

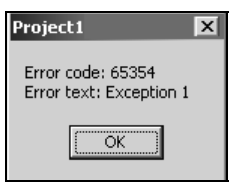


Рис. 3.3. Результат обработки ИС типа `EExceptionWithCode`

Исключительная ситуация *EAbort*

Если вы внимательно просмотрели код системной процедуры `HandleException`, то увидели там упоминание класса `EAbort`. ИС `EAbort` служит единственным — и очень важным — исключением из правил обработки. Она называется "тихой" (Silent) и отличается тем, что для нее обработка по умолчанию не предусматривает вывода сообщений на экран. Естественно, все сказанное касается и порожденных от нее дочерних объектных классов.

Применение `EAbort` оправдано во многих случаях. Вот один из примеров. Пусть разрабатывается некоторая прикладная программа или некоторое семейство объектов, не связанное с VCL. Если в них возникает ИС, то нужно как-то известить об этом пользователя. А между тем прямой вызов для этого функции `ShowMessage` или даже `MessageBox` не всегда оправдан. Для маленькой и компактной динамической библиотеки не нужно тащить за собой громаду VCL. С другой стороны, в большом и разнородном проекте нельзя давать каждому объекту или подпрограмме самой общаться с пользователем. Если их разрабатывают разные люди, такой проект может превратиться в вавилонскую башню. Тут и поможет `EAbort`. Эта исключительная ситуация не создается системой — ее должен создавать и обслуживать программист.

Применение `EAbort` — реальная альтернатива многочисленным конструкциям `if..then` и тем более (упаси боже!) `goto`. Эта ИС не должна подменять собой другие, вроде ошибки выделения памяти или чтения из файла. Она нужна, если вы сами видите, что сложились определенные условия и пора менять логику работы программы.

```
If LogicalCondition then Raise EAbort.Create('Condition 1');
```

Если не нужно определять сообщение, можно создать `EAbort` и проще — вызвав процедуру `Abort` (без параметров), содержащуюся в модуле SYSUTILS.PAS.

Функция *Assert*

Эта процедура и сопутствующая ей ИС `EAssertionFailed` специально перенесены в Object Pascal из языка C для удобства отладки. Синтаксис ее прост:

```
procedure Assert(expr : Boolean [; const msg: string]);
```

При вызове функции проверяется, чему равно значение переданного в нее булевого выражения `expr`. Если оно равно `True`, то ровным счетом ничего не происходит. Если же оно равно `False`, создается ИС `EAssertionFailed`. Все это было бы довольно тривиально с точки зрения уже изученного, если бы не два обстоятельства:

1. Предопределенный обработчик `EAssertionFailed` устроен таким образом, что выдает не шестнадцатеричный адрес ошибки, а имя файла с исходным текстом и номер строки, где произошла ИС, как показано на рис. 3.4.

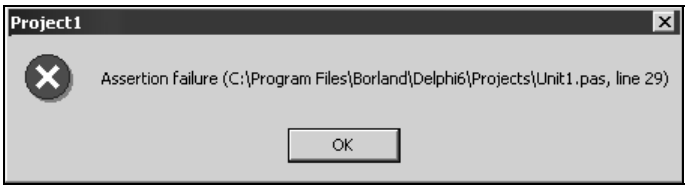


Рис. 3.4. Окно сообщения обработчика исключительной ситуации `EAssertionFailed`

2. При помощи специальной директивы компилятора `{$ASSERTIONS ON/OFF}` (или, что то же самое, `{$C+}/{$C–}`) возникновение этих ИС можно централизованно запретить. То есть в отлаживаемом коде в режиме `{$C+}` можно расставить вызов `Assert` во всех сомнительных и проверяемых местах. Когда же придет время генерации конечного варианта кода, переключением директивы на `{$C–}` весь отладочный вывод запрещается.

Резюме

Любое созданное в Delphi приложение должно обеспечивать обработку возможных исключительных ситуаций. Пусть вначале вам покажется ненужным создавать дополнительный код обработки ошибок для простейших программ, в которых вероятность ошибки минимальна. Зато впоследствии приобретенные навыки позволят вам существенно повысить надежность реальных приложений.

В Delphi для обработки исключительных ситуаций используются специальные конструкции языка Object Pascal и классы на основе базового класса исключительных ситуаций `Exception`.

ГЛАВА 4

Кроссплатформенное программирование для Linux

Времена безраздельного господства операционных систем Windows для домашних компьютеров и корпоративных рабочих станций подходят к концу. Все большее число рядовых компьютеров работает под управлением других операционных систем. Среди них по праву выделяется операционная система Linux, сочетающая в себе открытость и хорошие возможности настройки.

В этих условиях, когда бывает необходимо разрабатывать программное обеспечение с одними функциями сразу для нескольких операционных систем, программистам была бы весьма полезна среда разработки, позволяющая делать это по возможности с наименьшими затратами. Оставим в стороне споры о причинах и следствиях, о пользе или вреде такого развития ситуации и займемся техническим вопросом, связанным с тематикой данной книги — как взаимодействует Delphi 7 и Linux.

В этом месте читатель может заподозрить авторов в некомпетентности — ведь существует вполне самостоятельный программный продукт Kylix, который и предназначен для разработки программ для Linux. Да, Delphi и Kylix очень схожи, но каждый из них работает в своей операционной системе и о переносе программ не может быть и речи.

Однако Delphi 7 действительно позволяет писать программы для Linux.

Дело в том, что теперь разработчик, использующий Delphi 7, может создавать приложения, исходный код которых будет компилироваться без каких-либо дополнительных усилий не только в Delphi для Windows, но и в Kylix для Linux.

Для этого необходимо только лишь выбрать в Delphi соответствующий тип проекта и затем написать приложение. При этом разработчику будут доступны многие компоненты Палитры компонентов и соответственно возможности визуального программирования, чем всегда славилась Delphi.

Казалось бы, в переносимости кода широко распространенного и стандартизированного языка программирования нет ничего необычного и новаторского. Ведь можно же писать программы на ANSI C и компилировать их где угодно, так почему такая возможность для Pascal должна вызывать какие-либо эмоции? Несомненное преимущество кроссплатформенного программирования в Delphi заключается в том, что для совместного использования доступны не только обычные конструкции и операторы языка программирования, но и множество высокоуровневых компонентов для визуального программирования.

Кроссплатформенная разработка приложений в Delphi стала возможной благодаря созданию специального варианта библиотеки VCL, которая называется Component Library for Cross Platform (CLX). В основе CLX лежит иерархия специально созданных базовых классов, обеспечивающих работоспособность визуальных компонентов — потомков сразу в двух операционных системах. Конечно, набор компонентов CLX не столь богат по сравнению с нынешним разнообразием "супермаркета" VCL, однако вполне сравним с Палитрой компонентов Delphi или Delphi 2. А в те далекие времена при помощи Delphi разработчики создавали полный спектр программных продуктов и не слишком жаловались на скудость выбора компонентов.

Конечно же, серьезное кроссплатформенное программирование, включающее, например, взаимодействие с памятью, обработку процессов с учетом их приоритетов и т. д., потребует скрупулезной и вдумчивой работы. Но это неизбежно — совмещение возможностей двух операционных систем в одной программе — дело нелегкое, и проблема здесь не столько в недостатках среды разработки, сколько в сложности самой задачи. Попробуйте, например, написать кроссплатформенную программу, имеющую функции работы с файлами. Даже в этой простой и необходимой задаче вас ждут достаточно серьезные проблемы — файловые системы Windows и Linux трудно назвать идентичными.

В такой ситуации тем более ценно, что Delphi берет на себя все заботы по созданию интерфейса кроссплатформенной программы.

В этой главе рассматриваются следующие вопросы:

- состав стандартного проекта CLX и кроссплатформенные элементы Репозитория;
- CLX — библиотека компонентов кроссплатформенного программирования;
- иерархия классов CLX, общие свойства и методы компонентов, их отличия от компонентов VCL;
- особенности кроссплатформенного программирования Windows — Linux;
- дополнительные возможности кроссплатформенных приложений.

Проект CLX

Создание кроссплатформенного приложения в Delphi требует выполнения абсолютно стандартных действий. Достаточно создать новый проект, выбрав для этого в Репозитории пункт **CLX Application**.

На первый взгляд новый проект ничем не отличается от обычного, но это не так. Да и среда разработки тоже претерпела некоторые изменения. В Палитре компонентов теперь представлены компоненты из библиотеки CLX, той самой, которую использует Kylix. Да и немногочисленные изменения в составе проекта также связаны с необходимостью совмещения с Kylix.

Проект CLX отличается от обычного типом файла, содержащего информацию о форме. Если в обычном проекте файл формы имеет расширение dfm, то в проекте CLX это файл с расширением xfm, одинаково понятный и для Delphi, и для Kylix, так так и те и другие файлы являются обычными текстовыми файлами и сведения о форме представлены в них в текстовом виде. Примерно то же самое вы видите, просматривая форму в текстовом представлении в окне Редактора Delphi (команда **View as Text** из всплывающего меню формы).

Обратите внимание, что форма и модуль CLX связываются при помощи директивы `{$R *.xfm}`.

Кроме этого, в проектах Delphi и Kylix различаются расширения файла опций проекта (в Delphi — dof, в Kylix — kof). Однако это не принципиальная проблема и при отсутствии такого файла среда разработки создаст новый с настройками по умолчанию. Таким образом, всегда можно придумать как минимум несколько способов перенести текстовое содержимое файла настроек проекта.

Теперь рассмотрим сгенерированный Delphi исходный код проекта — это даст нам несколько интересных наблюдений. По синтаксису и основным элементам исходный код не отличается от стандартного. Файл проекта содержит список модулей и секцию `begin..end`. Файл модуля также обычен, за исключением списка используемых модулей в секции `uses`.

Выше мы говорили о библиотеке компонентов CLX и ее роли в кроссплатформенной разработке. И модули с непривычными названиями `QControls`, `QForms` и др. как раз содержат базовые классы библиотеки CLX.

В остальном проект CLX подобен стандартному проекту Delphi и в нем можно использовать весь инструментарий среды разработки и приемы визуального программирования.

В подтверждение сказанного при помощи нескольких изменений в проекте VCL мы можем легко преобразовать проект VCL в CLX и обратно. Для этого понадобится любой текстовый редактор. И конечно проект не должен содержать компонентов, которые не входят в состав библиотеки CLX.

1. В файле проекта в секции `uses` изменим ссылку на модуль `Forms` на `QForms`.
2. В файлах модулей заменим ссылки на модули VCL на модули CLX. Например, секция `uses` может выглядеть так:

   ```
   uses SysUtils, Types, Classes, QGraphics, QControls, QForms;
   ```
3. В файлах модулей заменим директиву `{$R *.dfm}` на `{$R *.xfm}`.
4. В файлах форм заменим расширение dfm на xfm.

Сохранив сделанные изменения и открыв проект в среде разработки, вы убедитесь, что Delphi приняла его за проект CLX, изменила соответствующим образом Палитру компонентов и с готовностью компилирует его.

Что же касается непосредственно кода кроссплатформенного приложения для Linux, то основные принципы его создания рассматриваются ниже в этой главе.

При необходимости разработчик может добавлять к проекту новые шаблоны, используя для этого Репозиторий. При этом в Репозитории доступны только те шаблоны, которые можно использовать в проекте CLX (например, форма, модуль или модуль данных). Отсутствуют шаблоны, использование которых в Linux невозможно или поддержку которых не обеспечивает библиотека CLX (например, шаблон однодокументного приложения или шаблоны печатных форм Quick Report).

Объектная концепция кроссплатформенного программирования

Программирование в Delphi подразумевает использование тех или иных классов, будь то формы, невизуальные компоненты или списки. Концепция кроссплатформенного программирования в рамках одной среды разработки имеет в виду наличие общего ядра, обеспечивающего функционирование зависимой от операционной системы программной надстройки. В Delphi таким ядром стала библиотека времени выполнения (RunTime Library — RTL), с использованием классов которой созданы библиотеки компонентов VCL и CLX.

В соответствии с этим для обеспечения кроссплатформенной разработки в исходные коды базовых классов Delphi были внесены изменения. Общим ядром библиотек компонентов VCL и CLX является иерархия классов `TObject` — `TPersistent` — `TComponent`, которые входят в состав RTL. Это позволяет среде разработки легко интегрировать кроссплатформенные проекты и работать со стандартными проектами для Windows.

Расхождения двух ветвей начинаются с класса `TControl`, который обеспечивает функциональность всех визуальных компонентов. Начиная с этого

класса и далее, библиотеки компонентов имеют собственные исходные коды. Многие модули CLX имеют названия, аналогичные модулям VCL, но с добавлением первой буквы Q, например QControls.pas.

В библиотеке VCL классы `TControl` и `TWinControl` являются предками всех компонентов, которые должны уметь отображать себя на экране при помощи графических средств операционной системы. В библиотеке CLX аналогичную задачу выполняют классы `TControl` и `TWidgetControl`. Они обеспечивают работоспособность компонентов-потомков в качестве экранных объектов widget.

Большинство свойств и методов классов `TWinControl` и `TWidgetControl` совпадают. Однако есть и различия, вызванные особенностями графических интерфейсов операционных систем Windows и Linux.

Более подробные сведения о функциональности перечисленных классов VCL и RTL содержатся в *гл. 2*.

Для обеспечения работоспособности кроссплатформенных классов CLX применяется динамическая библиотека Qt. Для использования ее методов в классах CLX в составе Delphi имеется заголовочный файл Qt.pas. При создании объекта CLX конструктором `Create` в исходном коде или переносом компонента на форму автоматически создается специальный объект — widget. Widget связан с объектом CLX, обеспечивает взаимодействие объекта CLX с операционной системой и уничтожается вместе с ним.

Однако widget может быть создан и напрямую. Такая ситуация может возникнуть, к примеру, при создании собственных компонентов. Для этого применяется функция `QWidget_Create` динамической библиотеки Qt. В этом случае widget не привязан к объекту CLX и, соответственно, не уничтожается вместе с ним.

Библиотека компонентов CLX

Безусловно, библиотека компонентов CLX более бедна по сравнению с VCL. Тем не менее, ее компоненты позволяют создавать полноценные приложения. В целом состав компонентов CLX напоминает Палитру компонентов ранних версий Delphi. Библиотека CLX загружается в Палитру компонентов при открытии существующего или создании нового проекта CLX.

Все компоненты CLX, имеющие аналоги в VCL, а таких большинство, имеют те же имена, что и компоненты VCL. Так как при переносе компонентов из Палитры компонентов на форму соответствующие модули подключаются в проект автоматически, никаких проблем с двойным наименованием не возникает.

Исходные модули библиотеки CLX содержаться в папке \Delphi7\Source \Clx.

Первые три страницы Палитры компонентов (**Standard**, **Additional**, **Common Controls**), а также страница **Dialogs** содержат визуальные и невизуальные компоненты для конструирования пользовательского интерфейса приложения.

Примечание

Из-за некоторых различий стандартов пользовательского интерфейса Windows и Linux часть визуальных компонентов CLX имеют несвойственные для Windows дополнительные функции. Отличия в стандартных свойствах и методах описываются ниже.

Большинство компонентов на этих страницах хорошо знакомы разработчикам (правда, некоторые из них перекочевали из других страниц VCL — например `TTimer` и `TSpinEdit`). Однако существуют некоторые новинки. Это компоненты `TLCDNumber`, `TTextViewer` и `TTextBrowser`. Их краткая аннотация представлена в табл. 4.1.

Таблица 4.1. *Уникальные визуальные компоненты CLX*

Компонент	Страница Палитры компонентов	Описание
`TLCDNumber`	**Additional**	Компонент отображает совокупность символов (букв и цифр), которые можно представить в режиме цифрового дисплея. Соответственно, не все буквы можно показать в этом компоненте. Например, буквы J, Q, Z и т. д. Строка символов содержится в свойстве `Value`
`TTextViewer`	**Common Controls**	Компонент является аналогом компонента VCL `TRichEdit`. Предназначен для редактирования текстов
`TTextBrowser`	**Common Controls**	Компонент развивает возможности компонента `TTextViewer`, предоставляя функции гипертекстовой разметки

Дополнительные возможности по созданию кроссплатформенных приложений баз данных дают компоненты на страницах **Data Access**, **DataControls**, **DBExpress**, **InterBase**. Безусловно, механизмы доступа к данным, используемые такими приложениями, в значительной степени зависят от операционной системы. Поэтому выбор способов доступа к данным сравнительно невелик.

Также библиотека CLX содержит довольно большое число компонентов, позволяющих создавать кроссплатформенные приложения для Internet. Это

и привычные традиционные компоненты (страницы **Internet**, **InternetExpress**, **WebServices**) и новые из набора Internet Direct (**Indy**).

Сходства и различия визуальных компонентов CLX и VCL

Большинство свойств и методов компонентов VCL и CLX идентичны. А существующие различия вызваны необходимостью использования специальных объектов — widget и особенностями представления визуальных элементов в Linux.

Базовые классы CLX — `TControl` и `TWidgetControl` для обеспечения прорисовки обращаются к динамической библиотеке Qt через заголовочный файл Qt.pas.

Таким образом, разработчик избавлен от необходимости работы с графическим интерфейсом Linux на низком уровне.

Для компонента CLX существует свойство

```
property Handle: QWidgetH;
```

которое является указателем на связанный объект widget и позволяет вызывать его методы напрямую.

Если экземпляр widget не создан, метод

```
procedure CreateHandle; virtual;
```

не только создает и инициализирует widget, но и устанавливает указатель `Handle`, создает объекты-перехватчики *(см. ниже)* и задает настройки по умолчанию для этого визуального компонента. При необходимости в классах-потомках метод `CreateHandle` перекрывается и в него добавляется новая функциональность.

Уничтожение созданного widget осуществляется методом

```
procedure DestroyHandle;
```

который уничтожает все дочерние widget и объекты-перехватчики, а также обнуляет свойства `Handle` и `Hooks`.

При необходимости для простого создания и инициализации widget можно использовать метод

```
procedure CreateWidget; virtual;
```

который сделает это, вызвав внешнюю функцию `QWidget_Create`, и метод

```
procedure InitWidget; virtual;
```

который определяет визуальные параметры widget.

Также в классах CLX доступен указатель на родительский widget за счет использования свойства

```
property ParentWidget: QWidgetH;
```

Если это свойство не определено, можно использовать свойство

```
property ChildHandle: QWidgetH;
```

родительского класса, например, таким образом:

```
if Not Assigned(ParentWidget) then
 if Assigned(Parent) then
    Result := Parent.ChildHandle;
```

В классах CLX иначе реализована обработка событий. В Linux все события делятся на два вида — системные и события widget. Системные события обрабатываются процедурой — аналогом процедуры `WndProc` для компонентов VCL.

События, генерируемые widget, перехватываются и обрабатываются специальными объектами, взаимодействующими с объектом widget. Затем они передаются связанному объекту CLX, который вызывает необходимые обработчики событий.

Объекты-перехватчики создаются при вызове метода

```
procedure HookEvents; virtual;
```

а непосредственно для создания перехватчиков используется библиотечная функция `QWidget_hook_create`. Метод `HookEvents` вызывается автоматически при создании widget.

Доступ к объекту-перехватчику возможен при помощи свойства

```
property Hooks: QWidget_hookH;
```

которое объявлено в секции `protected` и может быть использовано только при создании новых компонентов.

Классы CLX имеют очень интересное и важное свойство

```
property Style: TWidgetStyle;
```

которое позволяет управлять внешним видом и процессом отрисовки компонента.

Свойство

```
type TDefaultStyle = (dsWindows, dsMotif, dsMotifPlus, dsCDE, dsQtSGI,
dsPlatinum, dsSystemDefault);
property DefaultStyle: TDefaultStyle;
```

класса `TWidgetStyle` определяет стиль визуального компонента, задающий его внешний вид по умолчанию. Естественно, операционная система должна поддерживать выбранный стиль.

Кроме того, класс `TWidgetStyle` определяет некоторые наиболее общие параметры визуальных компонентов и обладает огромным числом обработчиков событий, которые вызываются при отрисовке всех возможных компонентов и экранных элементов.

Таким образом, свойство `Style` является прекрасным инструментом для создания собственных компонентов с нестандартной функциональностью.

Для использования в Linux модернизирована система контекстной помощи для компонентов CLX. Теперь статья подсказки для визуального компонента может быть вызвана двумя способами.

Традиционно, путем определения уникального номера статьи в свойстве

```
property HelpContext: THelpContext;
```

и дополнительно, путем определения ключевого слова подсказки в свойстве

```
property HelpKeyword: String;
```

Способ вызова помощи определяется свойством

```
type THelpType = (htKeyword, htContext);
property HelpType: THelpType;
```

Примечание

Свойства контекстной подсказки являются новыми в Delphi 7 и имеются у компонентов CLX и VCL.

Кроме того, отдельные компоненты CLX имеют дополнительные свойства и методы, определяющие их дополнительную функциональность в Linux.

В то же время некоторые привычные для программирования в Windows свойства компонентов отсутствуют в компонентах CLX. Это свойства обрамления компонента (`BevelEdges`, `BevelInner`, `BevelKind`, `BevelOuter`); возможность двунаправленной печати текстов (свойство `BiDiMode`); свойства для обратной совместимости с Windows 3.*x* (`Ctl3D` и `ParentCtl3D`); механизм присоединения и свойства Drag-and-Drop, хотя сам механизм Drag-and-Drop остался (свойства `DockSite`, `DragKind`, `DragCursor`).

Особенности программирования для Linux

Операционные системы Windows и Linux имеют достаточно серьезных различий, чтобы сделать кроссплатформенную разработку делом сложным и кропотливым. В первую очередь необходимо хорошо знать обе операционные системы и иметь опыт работы с ними.

Очевидно, что создание исходного кода для кроссплатформенных приложений — это трудоемкий процесс, который усложняется по мере использова-

ния специфических возможностей операционных систем. Простейшим путем в данном случае будет применение только стандартных свойств и методов компонентов CLX. Но, к сожалению, такой путь возможен для сравнительно несложных приложений.

К примеру, большинство приложений имеют функции для работы с файлами. Файловые системы Windows и Linux отличаются настолько, что кроссплатформенная реализация любых мало-мальски сложных операций с файлами требует серьезного внимания и усилий.

Linux чувствительна к регистру символов в именах файлов и путях, поэтому не стоит использовать в исходном коде имена файлов напрямую, а при необходимости делать это нужно аккуратно. Во многих компонентах для решения проблемы заглавных и строчных букв можно использовать свойство

```
property CaseSensitive: Boolean;
```

После присвоения свойству значения `True` компонент производит все строковые операции с учетом регистра символов.

Для формирования полного пути файла используйте константы из модуля SysUtils. Это `PathDelim` (символ разделителя каталогов в пути файла), `DriveDelim` (символ логического диска), `PathSep` (символ разделителя между несколькими путями файлов в одной строке).

При работе с текстовыми файлами необходимо помнить о различии управляющих символов в Windows и Linux. Для обозначения конца строки в Windows используются символы CR/LF, а в Linux — только символ LF. В Windows окончание текста определяется символом Ctrl-Z, а в Linux — просто концом файла.

Так как в Linux отсутствует системный реестр, то для сохранения настроек приложения используйте класс `TMemIniFile`, обеспечивающий сохранение переменных среды в INI-файле.

При создании кроссплатформенных приложений желательно использовать только свойства и методы классов CLX. В библиотеке CLX также доступны для применения такие важные для написания бизнес-логики приложения классы, как `TList`, `TStringList`, `TCollection`, `TAction` и др.

Если это ограничение является слишком жестким, и в программе требуется использовать функции системных API, применяйте директивы условного перехода:

```
{$IFDEF MSWINDOWS}
  {код для Windows}
{$ENDIF}

{$IFDEF LINUX}
  {код для Linux}
{$ENDIF}
```

Для определения исходного кода Windows применяйте константу `MSWINDOWS`, и не используйте константу `WIN32`, т. к. все еще есть код `WIN16` и никто не может поручиться, что вам не понадобится константа `WIN64` для соответствующего кода.

Приложения баз данных для Linux

Главной составной частью любого приложения баз данных является механизм доступа к данным. Для традиционных приложений баз данных, создаваемых в Delphi, выбор способов доступа к данным достаточно широк. Однако про кроссплатформенные приложения этого сказать нельзя. По существу, разработчик может выбрать только набор компонентов dbExpress. Или же, подобно старой рекламе автомобилей "форд", "Вы можете выбрать автомобиль любого цвета, если этот цвет черный", вам следует выбрать компоненты InterBase Express, если вы используете этот сервер для ваших данных в операционной системе Linux.

К сожалению, компоненты dbExpress ограничены по своим функциональным возможностям, обеспечивая однонаправленное перемещение курсора и просмотр данных в режиме "только для чтения".

Преимуществом этого способа доступа к данным является простота и отсутствие многомегабайтных вспомогательных библиотек. В частности, для каждого из четырех поддерживаемых dbExpress серверов баз данных необходима лишь одна динамическая библиотека Windows и только один разделяемый объект (shared object) Linux.

Подробное описание компонентов и механизма доступа dbExpress *см. в гл. 22.*

Internet-приложения для Linux

Для Internet-приложений вполне обычной является ситуация, когда клиентская часть должна работать на компьютерах с различными операционными системами, например Windows и Linux. В этом случае кроссплатформенное программирование клиентской части становится весьма привлекательным способом уменьшения затрат на процесс разработки.

В составе библиотеки CLX имеется достаточно большой набор компонентов для разработки Internet-приложений. Однако в Linux можно использовать только сервер Apache или CGI. Это накладывает существенные ограничения на вновь создаваемые кроссплатформенные приложения и требует серьезных усилий при переделке приложений Windows, использующих ISAPI или NSAPI.

Резюме

Кроссплатформенное программирование стало доступно в Delphi 7 благодаря использованию библиотеки компонентов CLX. Имея общее с библиотекой компонентов VCL ядро базовых компонентов, библиотека CLX обеспечивает совместимость приложений Delphi для Windows и Kylix для Linux.

При неизбежных для кроссплатформенного программирования трудностях реализации сложного кода, использующего системные вызовы и технологии удаленного доступа, в Delphi решена задача быстрого визуального проектирования пользовательского интерфейса и создания бизнес-логики приложения. Для этого применяется набор стандартных компонентов, имеющих практически идентичную функциональность и схожий программный интерфейс.

✧ ЧАСТЬ II ✧

Интерфейс и логика приложения

Глава 5. Элементы управления Win32

Глава 6. Элементы управления Windows XP

Глава 7. Списки и коллекции

Глава 8. Действия (Actions) и связанные с ними компоненты

Глава 9. Файлы и устройства ввода/вывода

Глава 10. Использование графики

ГЛАВА 5

Элементы управления Win32

Элементы управления составляют суть пользовательского интерфейса Windows. Всеми программами нужно управлять более или менее единообразно, поэтому в составе ОС имеется набор типовых кнопок, редактирующих элементов, списков выбора и т. п., которыми вы можете "украсить" свои разработки. Перечень этот постоянно пополняется. Во-первых, не дремлет фирма Microsoft. С новыми версиями ее продуктов (главным образом с MS Internet Explorer) поставляются новые элементы управления; содержатся они в библиотеке ComCtl32.dll. Во-вторых, на ниве их создания подвизаются многочисленные сторонние фирмы, оформляющие свои элементы управления в виде элементов ActiveX (файлов OCX). И, в-третьих, достаточное количество элементов написано прямо в Delphi — как в фирме Borland, так и независимыми разработчиками.

Элементам управления, пришедшим из состава Windows, начиная с Delphi 3, посвящается отдельная страница в Палитре компонентов под названием **Win32**. Их количество (и возможности!) постоянно растет. В этой главе будут рассмотрены основные и новые для Delphi 7 элементы.

Что такое библиотека ComCtl32

Изменив внешний вид "окон" в Windows 95, менеджеры Microsoft задумались о том, чтобы дать независимым разработчикам средства для создания приложений, внешне похожих на системные утилиты и использующих единые типовые элементы управления. Например, все эти элементы автоматически поддерживают установленные в системе цветовое оформление, размер, шрифт и т. п.

Все элементы, проверенные и обкатанные в Windows, объединялись в библиотеке ComCtl32.dll, документировались и публиковались для использования разработчиками. С 1995 года сменилось много версий библиотеки, эле-

менты добавлялись и совершенствовались. Соответственно росла и страница **Win32** в Палитре инструментов Delphi. Все компоненты, представленные там, взяты из библиотеки ComCtl32.

Специально отметим, что эти элементы управления не являются ActiveX. Это — обычные специализированные разновидности окон Windows, установка свойств которых происходит через посылку специализированных сообщений. Полная документация по всем сообщениям и применяемым в них константам и структурам есть в MSDN. Для большинства сообщений предусмотрены специальные функции оболочки, которые описаны в модуле CommCtrl.pas. Сами "дельфийские" классы компонентов работают с их использованием; классы компонентов описаны в модуле ComCtrls.pas.

Примечание

Своими свойствами компоненты Delphi покрывают лишь 70—80% возможностей соответствующих элементов управления. Так что для решения некоторых специфических задач иногда приходится обращаться к функциям модуля ComCtrls.pas. Примеры этого имеются как в настоящей, так и в последующих главах данной книги.

Если вы распространяете свое приложение, содержащее компоненты со страницы **Win32** Палитры компонентов, не забудьте позаботиться о проблеме совместимости. Если версия библиотеки ComCtl32 на компьютере пользователя старее той, что использовалась вами при разработке, то в лучшем случае вы получите проблемы с правильной отрисовкой и поведением элементов управления, а в худшем — фатальную ошибку и неработающее приложение.

Проблема эта решается включением в дистрибутив и запуском на машине клиента приложения 50comupd.exe, которое находится на дистрибутивном диске Delphi 7 в папке \info\extras\comctl\ и обновляет библиотеку, или его можно скачать с сервера Microsoft по адресу **http://www.microsoft.com /msdownload/ieplatform/ie/comctrlx86.asp**. Ни в коем случае не пытайтесь просто ("руками") заменить ComCtl32.dll на более новую версию — это запрещено Microsoft и последствия могут вас серьезно огорчить.

Размер 50comupd.exe составляет около 500 Кбайт, так что если оно и "утяжелит" ваш дистрибутив, то ненамного. Есть и более масштабный способ решения этой проблемы — установить последнюю версию Internet Explorer, который включает в себя новую библиотеку ComCtl32. Если у вас дело упирается только в элементы управления, то этот способ избыточен. Но если вы используете в распространяемом приложении другие решения от Microsoft, привязанные к IE (скажем, криптографическую библиотеку CryptoAPI или стек протоколов Internet в динамической библиотеке WinInet.dll — на нем базируется большинство сетевых технологий в Delphi 7), то он может оказаться необходимым.

Проверка версии библиотеки элементов управления делается с использованием функции из модуля ComCtrls.pas:

```
function GetComCtlVersion: Integer;
```

Она возвращает номер установленной версии библиотеки в виде пары цифр. Каждая версия выходит обычно вместе со следующей версией обозревателя Internet. Вот константы, описанные также в модуле ComCtrls.pas:

```
const
ComCtlVersionIE3 = $00040046;
ComCtlVersionIE4 = $00040047;
ComCtlVersionIE401 = $00040048;
ComCtlVersionIE5 = $00050050;
ComCtlVersionIE501 = $00050051;
ComCtlVersionIE6 = $00060000;
```

В Windows 2000 и ME библиотека элементов управления имеет версию $00050051 (5.81). Наконец, в следующей ОС от Microsoft — Windows XP (ex-Whistler) — сделаны кардинальные изменения внешних возможностей интерфейса пользователя. И основой их станет 6-я версия библиотеки ComCtl32.

И о совместимости. Не нужно быть мудрецом, чтобы понять, что компоненты со страницы **Win32** работают только в среде Win32. Когда вы создаете Kylix-совместимое приложение (CLX Application), вместо страницы **Win32** в Палитре компонентов появляется страница **Common Controls**. Находящиеся там компоненты сделаны на основе библиотеки QT (**www.trolltech.com**) и лицензированы Borland. Во многом они повторяют методы и свойства элементов управления, соответствующих им в Windows, но далеко не всегда. Поэтому, если вы пишете переносимые приложения, забудьте о библиотеке ComCtl32 и странице Палитры компонентов **Win32** и пропустите эту главу.

Многостраничный блокнот — компоненты *TTabControl* и *TPageControl*

В Палитре компонентов имеется два элемента управления, обеспечивающих создание многостраничных блокнотов. Это компоненты `TTabControl` и `TPageControl`. Переключение между страницами осуществляется при помощи закладок. Закладки могут выглядеть как "настоящие" в бумажном блокноте, а могут быть похожи на стандартные кнопки Windows. Кстати, сама Палитра компонентов Delphi является примером использования такого элемента управления.

Компонент `TTabControl` отличается тем, что представляет собой как бы "виртуальный" блокнот. Это — единый объект с одной фактической страни-

цей. При переключении закладок осуществляется вызов метода-обработчика события

```
property OnChange: TNotifyEvent;
```

соответствующий код в котором может изменить набор видимых элементов управления и создать для пользователя иллюзию "переключения страниц".

Компонент `TPageControl` является контейнером для объектов `TTabSheet`, соответствующих отдельным страницам блокнота. Страницы в нем могут нести каждая свой набор дочерних компонентов; их можно переключать уже во время разработки.

Первый подход удобен, если на разных страницах у вас должны располагаться одни и те же компоненты, "начиненные" различными данными. Идеальный пример приводится самими разработчиками Delphi (папка Help\Samples\TabCntrl — обязательно посмотрите пример!). Здесь `TTabControl` используется для редактирования базы данных. Закладки для страниц создаются по одной для каждой записи в таблице. А на одной-единственной странице располагаются компоненты для отображения данных. При переключении закладок происходит навигация по таблице, содержимое полей меняется, и создается впечатление перехода на другую страницу.

Второй подход необходим, если у вас действительно разные страницы с различными наборами компонентов на них. Компонент `TPageControl` используют для создания редакторов свойств и настроек программы, а также для разного рода мастеров (Wizards).

Оба компонента в своей основе имеют общий элемент управления из библиотеки ComCtl32 (в документации Microsoft он называется `Tab Control`). Соответственно, в иерархии классов Delphi они оба произошли от класса `TCustomTabControl`, от которого унаследовали значительную часть свойств и методов. А вот механизмы работы отдельных страниц у каждого компонента свои. Поэтому сначала мы рассмотрим общие для двух компонентов свойства, а затем особенности использования страниц. Свойства и методы-обработчики класса-предка `TCustomTabControl` представлены в табл. 5.1 и 5.2 соответственно. Обратите внимание, что перечисленные свойства и методы в потомках объявляются как опубликованные (`published`).

Таблица 5.1. *Основные свойства, общие для `TTabControl` и `TPageControl`*

Объявление	Описание
`property TabIndex: Integer;`	Задает номер текущей страницы, начиная с 0
`property TabHeight: Smallint;`	Задает высоту закладок в пикселах. При значении 0 высота определяется автоматически так, чтобы вместить текст

Таблица 5.1 *(окончание)*

Объявление	Описание
`property TabWidth: Smallint;`	Задает ширину закладок. При значении 0 ширина определяется автоматически так, чтобы вместить текст
`type TTabStyle = (tsTabs, tsButtons, tsFlatButtons);` `property Style: TtabStyle;`	Определяет стиль закладок компонента: • `tsTabs` — стандартные закладки; • `tsButtons` — объемные кнопки; • `tsFlatButtons` — плоские кнопки
`type TTabPosition = (tpTop, tpBottom, tpLeft, tpRight);` `property TabPosition: TTabPosition;`	Определяет расположение закладок на компоненте. Расположение, отличное от `tpTop`, возможно только для стиля `tsTabs`
`property HotTrack: Boolean;`	При значении `True` названия страниц выделяются цветом при перемещении над ними указателя мыши
`property Images: TCustomImageList;`	Указывает на список картинок, появляющихся на закладках страниц
`property RaggedRight: Boolean;`	При значении `True` ширина закладок изменяется таким образом, чтобы они не занимали всю сторону блокнота
`property MultiLine: Boolean;`	При значении `True` закладки страниц могут располагаться в несколько рядов (если они не помещаются в один). При значении `False` в верхнем правом углу появляются кнопки, организующие прокрутку невидимых заголовков
`property ScrollOpposite: Boolean;`	При значении `True`, если закладки расположены в несколько рядов, при переходе к закладке следующего ряда все остальные ряды перемещаются на противоположную сторону блокнота. Действительно только при `MultiLine=True`

Таблица 5.2. *Основные методы-обработчики, общие для `TTabControl` и `TPageControl`*

Объявление	Описание
`type TTabChangingEvent = procedure (Sender: TObject; var AllowChange: Boolean) of object;` `property OnChanging: TTabChangingEvent;`	Вызывается непосредственно перед открытием новой страницы. Параметр `AllowChange`, установленный в значение `False`, запрещает открытие

Таблица 5.2 *(окончание)*

Объявление	Описание
`property OnChange: TNotifyEvent;`	Вызывается при открытии новой страницы
`property OnDrawTab: TDrawTabEvent;`	Вызывается при перерисовке страницы, только если свойство `OwnerDraw = True`
`property OnGetImageIndex: TTabGetImageEvent;`	Вызывается при отображении на закладке картинки

Как видно из таблицы, большинство свойств обеспечивают различные стили представления многостраничного блокнота. При настройке стиля обратите внимание, что свойство `RaggedRight` может не работать, т. к. вступает в противоречие со свойством `TabWidth`. При `TabWidth = 0` компонент изменяет ширину закладок в соответствии с длиной текста, в противном случае ширина закладок всегда равна значению свойства `TabWidth`.

Для того чтобы в закладках совместно с текстом показать картинки, используется свойство `Images`, в котором необходимо задать требуемый экземпляр компонента `TImageList` *(см. ниже)*.

Свойство `TabIndex`, задающее номер текущей страницы, позволяет переключать страницы программно. Для компонента `TTabControl` это единственный способ изменить текущую страницу на этапе разработки. При смене страниц сначала происходит событие `OnChanging` — в этот момент `TabIndex` еще содержит индекс старой страницы (и смену можно запретить), а затем `OnChange` — это свойство уже указывает на новую страницу.

В компоненте `TTabControl` число и заголовки страниц полностью зависят от свойства

```
property Tabs: TStrings;
```

В списке перечисляются заголовки страниц, для которых автоматически создаются закладки. Порядок следования страниц зависит от расположения текстов заголовков в свойстве `Tabs`.

При этом забота о правильном чередовании элементов управления при смене страниц полностью ложится на программиста. Для этого необходимо в методе-обработчике `OnChange` определить видимость элементов в зависимости от индекса текущей страницы:

```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
 with TabControl1 do
```

```
 begin
  Edit1.Visible := TabIndex = 0;
  Edit2.Visible := TabIndex = 1;
  Edit3.Visible := TabIndex = 2;
 end;
end;
```

Компонент `TPageControl`, в отличие от `TTabControl`, для обеспечения работы создает "настоящую" страницу — экземпляр класса `TTabSheet`. Список указателей на все созданные экземпляры страниц хранится в свойстве `Pages`, доступном только для чтения:

```
property Pages[Index: Integer]: TTabSheet;
```

Номер индекса соответствует порядковому номеру страницы. Для создания новой страницы используется команда **New Page** из всплывающего меню компонента, перенесенного на форму. Если же вы хотите создать страницу на этапе выполнения, создайте экземпляр `TTabSheet` самостоятельно и в свойстве `PageControl` укажите на родительский блокнот:

```
 pcMain: TPageControl;
 ts : TTabSheet;
...
ts := TTabSheet.Create(pcMain);
  with ts do
   begin
    PageControl := pcMain;
    ts.Caption :='New page';
   end;
```

Общее число страниц хранится в свойстве

```
property PageCount: Integer;
```

доступном только для чтения. Текущую страницу можно задать свойством:

```
property ActivePage: TTabSheet;
```

Если во время разработки (этой возможностью компонент `TPageControl` отличается от своего собрата) или во время выполнения переключиться на другую страницу, значение свойства `ActivePage` изменится.

Также для перехода на соседнюю страницу программными средствами можно использовать метод

```
procedure SelectNextPage(GoForward: Boolean);
```

в котором параметр `GoForward` при значении `True` задает переход на следующую страницу, иначе — на предыдущую.

Рассмотрев свойства блокнота, обратимся к его страницам и остановимся подробнее на возможностях класса `TTabSheet`. На владельца страницы указывает значение свойства

```
property PageControl: TPageControl;
```

Расположение страницы в блокноте задает свойство `PageIndex`:

```
property PageIndex: Integer;
```

Если в блокноте одновременно выделено несколько страниц, то положение данной страницы среди выделенных определяется свойством только для чтения

```
property TabIndex: Integer;
```

Страница может временно "исчезнуть" из блокнота, а затем опять появиться. Для этого применяется свойство

```
property TabVisible: Boolean;
```

Компонент *TToolBar*

Возможность создать панель инструментов появилась у разработчика давно, начиная с первых версий Delphi. Тогда она была реализована с помощью сочетания компонентов `TPanel` и `TSpeedButton`. Так можно было поступить и сейчас; но панель инструментов получила развитие с появлением стандартного элемента управления `TToolBar`, который объединяет расположенные на нем кнопки и другие элементы управления и централизованно управляет ими.

Примечание

Для других элементов управления, помещаемых на `TToolBar`, создается невидимая кнопка, обеспечивающая взаимодействие между ними и панелью. Но не со всеми из них "все гладко". Например, компонент `TSpinEdit` масштабируется и позиционируется неправильно. Вместо него следует применять пару `TEdit+TUpDown`.

Все кнопки (класс `TToolButton`) на панели инструментов имеют одинаковый размер, задаваемый свойствами:

```
property ButtonWidth: Integer;
property ButtonHeight: Integer;
```

Но эти свойства срабатывают только тогда, когда свойство

```
property ShowCaptions: Boolean;
```

имеет значение `False`. Оно отвечает за видимость надписей на кнопках компонента. И если эти надписи должны быть видимы, то размер кнопок автоматически подстраивается под размер кнопки с самым длинным текстом.

На каждой кнопке могут отображаться два ее атрибута — текст (заголовок кнопки, свойство `Caption`) и картинка. Показ текста можно запретить установкой свойства `ShowCaptions` в значение `False`.

Панель инструментов тяжело себе представить без украшающих ее картинок. У компонента `TToolBar` целых три свойства, ссылающихся на списки картинок:

```
property Images: TCustomImageList;
property DisabledImages: TCustomImageList;
property HotImages: TCustomImageList;
```

В обычном состоянии на кнопках отображаются картинки из набора, указанного свойством `Images`. Если кнопка неактивна (свойство `Enabled` обращено в `False`), надпись на кнопке отображается серым цветом и на ней показывается картинка из свойства `DisabledImages`.

Если свойство

```
property Flat: Boolean;
```

установлено в значение `True`, внешний вид панели инструментов становится более "модным" — плоским. В этом случае границы кнопок не видны, и все они выглядят как набор надписей и рисунков на единой плоской панели. Границы становятся видны, только когда над кнопкой находится указатель мыши. Можно при этом изменить и внешний вид кнопки. Если задано значение свойства `HotImages`, то на текущей кнопке обычная картинка из `Images` меняется на картинку из `HotImages`. Посмотрите, например, на панель Microsoft Internet Explorer 4 и старшей версии — там все картинки на кнопках серые, но кнопка, к которой подведен указатель мыши, становится цветной.

Примечание

Возможность сделать панель инструментов плоской появилась в версии 4.70 библиотеки ComCtl32.dll. Распространяя приложение, не забудьте поставить с дистрибутивами эту библиотеку нужной версии.

Текст и картинка на кнопке могут располагаться друг относительно друга двумя способами, в зависимости от значения свойства `List`. Если свойство `List` равно значению `False` (установка по умолчанию), то картинка располагается сверху, текст — снизу. В противном случае вы увидите текст справа от картинки.

Панели инструментов — это контейнеры, но и они часто располагаются на контейнерах — компонентах `TCoolBar` и `TControlBar`. Те, как правило, имеют свою фоновую картинку, и располагающийся сверху компонент `TToolBar` можно сделать прозрачным. За это отвечает свойство:

```
property Transparent: Boolean;
```

Особенно удобно использовать его, если установлен режим плоской панели — в этом случае прозрачен не только фон самой панели, но и всех кнопок на ней.

Перейдем к рассмотрению функциональных возможностей кнопок. Вы думаете, что "функциональные возможности" — это громко сказано? С одной стороны, да: кнопка — это то, на что нажимает пользователь, и не более того. Главное событие и для кнопки `TToolButton`, и для панели `TToolBar` — событие `OnClick`. Кроме него они могут отреагировать только на перемещение мыши и на процессы перетаскивания/присоединения (Drag-and-Drop, Drag-and-Dock; описанные ниже в этой главе).

С другой стороны, кнопки можно нажимать в разнообразных вариантах и сочетаниях. Ключ к выбору варианта — свойство `Style` объекта `TToolButton`:

```
type TToolButtonStyle = (tbsButton, tbsCheck, tbsDropDown, tbsSeparator,
tbsDivider);
property Style: TToolButtonStyle;
```

Стили `tbsSeparator` и `tbsDivider` предназначены для оформления панели и представляют собой пустое место и вертикальный разделитель соответственно. Обычная кнопка — это, понятное дело, стиль `tbsButton`.

Если вы хотите создать одну или несколько кнопок, "залипающих" после нажатия, выберите стиль `tbsCheck`. После щелчка такая кнопка остается в нажатом состоянии до следующего нажатия. Об ее состоянии говорит свойство:

```
property Down: Boolean;
```

Если нужна группа кнопок, из которых только одна может пребывать в нажатом состоянии, следует воспользоваться свойством:

```
property Grouped: Boolean;
```

Такая группа называется *группой с зависимым нажатием*. Если на панели инструментов есть ряд расположенных подряд кнопок с `Style=tbsCheck` и `Grouped=True`, то этот ряд будет обладать свойствами группы с зависимым нажатием. Если групп зависимых кнопок должно быть две и более, разделить их между собой можно кнопкой другого стиля (например, `tbsSeparator` или `tbsDivider`) или любым другим элементом управления (рис. 5.1).

В такой группе всегда должна быть нажата хотя бы одна кнопка; на этапе разработки установите ее свойство `Down` в значение `True`. Но если это вам не подходит, можно установить свойство

```
property AllowAllUp: Boolean;
```

в значение `True` — и можно отжимать все кнопки. Значение этого свойства всегда одинаково для всех кнопок в группе.

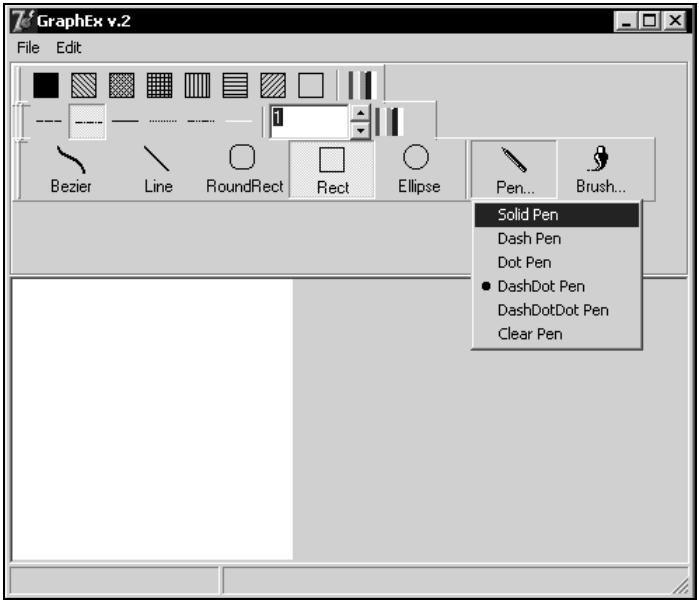


Рис. 5.1. Несколько групп кнопок с зависимым нажатием на панели инструментов

Если в какой-то ситуации одна или несколько кнопок должны стать недоступными, для этого можно установить свойство `Enabled` в значение `False`. Но у кнопок в группе есть еще и третье состояние — неопределенное:

```
property Indeterminate: Boolean;
```

Такие кнопки выделяются серым цветом, чтобы показать пользователю, что их выбирать не следует. Переход в состояние `Indeterminate=True` все еще позволяет кнопке обрабатывать событие `OnClick`, но при этом она переходит в отжатое состояние (`Down=False`). Но — только до следующего нажатия. После него кнопка выходит из состояния `Indeterminate`.

Свойство

```
property Marked: Boolean;
```

отображает поверхность кнопки синим цветом (точнее, цветом `clHighlight`), как у выделенных объектов. В отличие от предыдущего случая с `Indeterminate` кнопка остается в состоянии `Marked` независимо от нажатий вплоть до присвоения этому свойству значения `False`.

Ниже приведен фрагмент программы, с помощью которого можно выделить кнопки на панели при помощи мыши. Приведенные ниже обработчики событий нужно присвоить всем кнопкам панели и самой панели `TToolBar`:

```
var StartingPoint : TPoint;
    Selecting : boolean;
```

```
procedure TForm1.ToolBar1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 StartingPoint := (Sender as TControl).ClientToScreen(Point(X,Y));
 Selecting := True;
end;

procedure TForm1.ToolBar1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i: Integer;r,r0 : TRect;
begin
 if Selecting then
  begin
  r.TopLeft := StartingPoint;
  r.BottomRight := (Sender as TControl).ClientToScreen(Point(X,Y));
  with ToolBar1 do for i := 0 to ButtonCount-1 do
   begin
   r0 :=Buttons[i].ClientRect;
   OffsetRect(r0,Buttons[i].ClientOrigin.X,Buttons[i].ClientOrigin.Y);
   if IntersectRect(r0,r,r0) then
     Buttons[i].Marked := True;
   end;
  end;
 Selecting := False;
end;
```

Наличие обработчиков событий `OnMouseDown/OnMouseUp` не мешает нажатию кнопок — нажатие все равно вызывает событие `OnClick`.

Компонент `TToolBar` может стать полноценной заменой главного меню (взгляните хотя бы на приложения из состава MS Office 97 или 2000). К каждой из кнопок можно присоединить меню — и не одно, а целых два:

```
property DropdownMenu: TPopupMenu;
property PopupMenu: TPopupMenu;
```

Для того чтобы по нажатии левой кнопки мыши выпадало меню `DropdownMenu`, нужно установить один из стилей кнопок — `tbsButton` или `tbsDropdown`. В первом случае меню появится при нажатии в любой части кнопки. При этом событие `OnClick` не возникает; кнопка из-за этого становится "неполноценной" — она пригодна только для показа меню. Второй случай — стиль `tbsDropdown` — специально предназначен для удобства работы с выпадающими меню. Меню появляеется при щелчке на специальной области с изображением треугольника в правой части кнопки. А вот щелчок на остальной части кнопки, как обычно, вызовет событие `OnClick`.

Компонент *TImageList*

С ростом возможностей пользовательского интерфейса Windows все больше и больше элементов управления стали оснащаться значками и картинками. И вот для централизованного управления этими картинками появился элемент управления `TImageList`. Он представляет собой оболочку для создания и использования коллекции одинаковых по размеру и свойствам изображений. На этапе разработки ее "начиняют" картинками (с Delphi для этих целей поставляется целая подборка, находящаяся в каталоге \Images). Компонент `TImageList` обладает двумя свойствами — `Images` и `ImageIndex`. Первое указывает на список (компонент `TImageList`), второе — на конкретную картинку в этом списке.

Проще всего заполнить список при помощи встроенного редактора (рис. 5.2), выполнив двойной щелчок на компоненте или выбрав пункт **Image List Editor** в его контекстном меню.

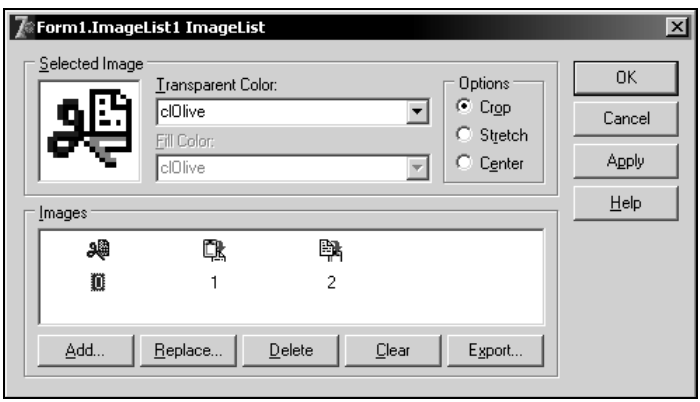


Рис. 5.2. Редактор списка изображений `TImageList`

Пользоваться редактором очень просто, но нужно обратить внимание на одну тонкость. Только что выбранное изображение можно отредактировать, изменив его положение относительно отведенного ему прямоугольника: **Crop** (размещение, начиная с точки (0, 0)), **Stretch** (масштабирование) или **Center** (центровка). Кроме того, можно изменить прозрачный цвет (**Transparent Color**). Точки с этим цветом при отрисовке не будут видны (прозрачны). Изображение можно выбрать либо из списка, либо мышью, щелкнув в нужном месте на увеличенной картинке в верхнем левом углу редактора. Если редактор уже записал изображение в список, редактирование этих свойств становится невозможным. Запись происходит, например, при закрытии редактора. Стало быть, размер картинок (свойства `Height` и `Width`) нужно установить заранее. Если компонент настроен на размер 16×16, а вы

пытаетесь наполнить его картинками 32×32, они будут сжаты и потеряют во внешнем виде.

Можно сильно упростить подбор картинок для `TImageList`. Если просмотреть ресурсы приложений из состава MS Office, да и многих других пакетов, то можно обнаружить, что картинки, которые встречаются на панелях инструментов, "склеены" между собой. Для просмотра ресурсов можно использовать, к примеру, приложение Resxplor, поставляемое в качестве примера с Delphi 7.

Такие картинки удобно использовать и в собственных программах. Кроме того, со времен Delphi 3 известна следующая ошибка разработчиков Microsoft: в разных версиях библиотеки ComCtl32.dll запись и чтение картинок при сохранении осуществлялась по-разному; если вы заполнили список во время разработки, скомпилировали приложение и запустили его на машине с другой версией библиотеки ComCtl32, вполне вероятно, что список окажется пустым.

Таким образом, с любой точки зрения правильнее явно читать картинки из ресурсов. Последовательность действий для этого следующая:

1. Создать исходный файл ресурсов, куда нужно включить и поименовать требуемые файлы с расширением bmp, к примеру:

   ```
   inout BITMAP "inout.bmp"
   tools BITMAP "tools.bmp"
   ```

   Сохранить этот файл с расширением rc, скажем, bitmap.rc.

2. Скомпилировать ресурсы при помощи утилиты brcc32.exe, поставляемой с Delphi:

   ```
   C:\Program Files\Borland\Delphi7\bin\brcc32 bitmap.rc
   ```

3. Появившийся файл bitmap.res нужно включить в состав проекта. Для этого используется директива `$R`:

   ```
   {$R bitmap.res}
   ```

4. Теперь картинка содержится в ресурсах и будет включена в состав исполняемого файла. Осталось загрузить ее в компонент `TImageList`. Для этого используется метод `ResourceLoad`:

   ```
   ImageList1.ResourceLoad(rtBitmap, 'bitmaps',TColor(0));
   ```

При этом произойдет автоматическая "нарезка" картинок в соответствии со свойствами `Width` и `Height`. Если размер большой картинки, к примеру, 256×16 пикселов, а ширина, заданная свойством `TImageList`, равна 16 пикселам, то в список будут включены 16 элементов размером 16×16. Поэтому еще во время разработки нужно правильно настроить размеры в компоненте `TImageList`, иначе после загрузки ресурса картинки будут разрезаны как попало.

Есть и другой метод загрузки — `FileLoad`:

```
function FileLoad(ResType: TResType; Name: string; MaskColor: TColor):
Boolean;
```

Аналогичным путем он позволяет загружать картинки из любого пригодного файла. Но загрузка из файла менее надежна — нет гарантии, что у пользователя вашего приложения нужный файл всегда находится на месте и он не изменен.

Описанный выше редактор списка картинок "умеет" делать их прозрачными еще во время разработки. Часто бывает необходимо сделать прозрачными картинки, загружаемые из файлов во время исполнения. Для этого нужно использовать их свойство `Transparent`:

```
Var bmp: TBitmap;
...
bmp.LoadFromFile('c:\test.bmp');
bmp.Transparent := True;
ImageList1.AddMasked(bmp, bmp.TransparentColor);
...
```

В методе `AddMasked` нужно вторым параметром указать "прозрачный" (фоновый) цвет, который в данном случае равен `bmp.TransparentColor`.

Как элемент управления **Win32**, компоннет `TImageList` имеет собственный дескриптор:

```
property Handle: HImageList;
```

Не следует путать этот дескриптор с дескрипторами растровых картинок, входящих в состав списка. В файле CommCtrl.pas приведены прототипы всех функций для работы с этим элементом управления, и для их вызова необходимо значение свойства `Handle`. Обратитесь к ним, если опубликованных свойств `TImageList` вам недостаточно.

Компоненты *TTreeView* и *TListView*

Эти компоненты известны каждому, кто хоть раз видел Windows 98 или Windows 2000. Именно на их базе создано ядро пользовательского интерфейса — оболочка Explorer, да и большинство других утилит Windows. Они включены в библиотеку ComCtl32.dll и доступны программистам.

Компонент `TTreeView` называют *деревом* (рис. 5.3).

Компонент `TTreeView` — правопреемник компонента `TOutline`, разработанного Borland еще для Delphi 1 и предназначен для отображения иерархической информации. Его "сердцем" является свойство

```
property Items: TTreeNodes;
```

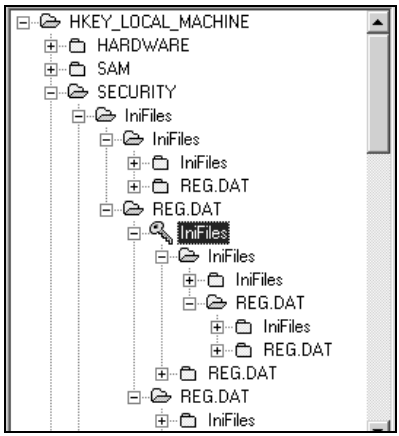


Рис. 5.3. Внешний вид компонента `TTreeView`

Данное свойство — это список всех вершин дерева, причем список, обладающий дополнительными полезными свойствами. Каждый из элементов списка — это объект типа `TTreeNode`. Свойства его сведены в табл. 5.3.

Таблица 5.3. *Список свойств объекта* `TTreeNode`

Объявление	Описание
`property HasChildren: Boolean;`	Равно `True`, если узел имеет дочерние узлы
`property Count: Integer;`	Счетчик числа дочерних узлов данного узла
`property Item[Index: Integer]: TTreeNode;`	Список дочерних узлов
`property Parent: TTreeNode;`	Ссылка на объект — родительский узел (верхнего уровня)
`property Level: Integer;`	Уровень, на котором находится узел. Для корневого узла это свойство равно 0; его потомки имеют значение `Level=1` и т. д.
`property Text: string;`	Текст узла
`property Data: Pointer;`	Данные, связанные с узлом
`property TreeView: TCustomTreeView;`	Ссылка на компонент `TTreeView`, в котором отображается данный узел
`property Handle: HWND;`	Дескриптор окна компонента `TTreeView`, в котором отображается данный узел
`property Owner: TTreeNodes;`	Ссылка на компонент `TTreeNodes`, которому принадлежит данный узел

Таблица 5.3 *(окончание)*

Объявление	Описание
`property Index: Longint;`	Индекс узла в списке своего родителя
`property IsVisible: Boolean;`	Равно `True`, если узел видим (все его родительские узлы развернуты)
`property ItemId: HTreeItem;`	Дескриптор узла (применяется при вызове некоторых методов)
`property AbsoluteIndex: Integer;`	Абсолютный индекс узла в списке корневого узла
`property ImageIndex: Integer;`	Индекс картинки, соответствующей невыбранному узлу в нормальном состоянии
`property SelectedIndex: Integer;`	Индекс картинки, соответствующей выбранному узлу
`property OverlayIndex: Integer;`	Индекс картинки, которая может накладываться поверх основной
`property StateIndex: Integer;`	Индекс дополнительной картинки, отражающей состояние узла
`property Selected: Boolean;`	Равно `True`, если данный узел выбран пользователем
`property Focused: Boolean;`	Равно `True`, если данный узел выбран пользователем для редактирования текста узла
`property Expanded: Boolean;`	Равно `True`, если данный узел развернут (показываются его дочерние узлы)

Очень важным является свойство `Data`. Вместе с каждым узлом можно хранить не только текст, но и любые данные. Необходимо только помнить, что при удалении узла они автоматически не освобождаются, и это придется сделать вручную.

Для добавления узлов в дерево используются десять методов объекта `TTreeNode` (табл. 5.4).

Таблица 5.4. *Методы, позволяющие добавлять узлы в объект* `TTreeNode`

Метод	Описание
`function Add(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется последним в тот же список, что и узел `Node`

Таблица 5.4 *(окончание)*

Метод	Описание
`function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и метод `Add`, но с узлом связываются данные из параметра `Ptr`
`function AddFirst(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется первым в тот же список, что и узел `Node`
`function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и метод `AddFirst`, но с узлом связываются данные из параметра `Ptr`
`function AddChild(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется последним в список дочерних узлов узла `Node`
`function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и метод `AddChild`, но с узлом связываются данные из параметра `Ptr`
`function AddChildFirst(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется первым в список дочерних узлов узла `Node`
`function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и метод `AddChildFirst`, но с узлом связываются данные из параметра `Ptr`
`function Insert(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется непосредственно перед узлом `Node`
`function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и метод `Insert`, но с узлом связываются данные из параметра `Ptr`

Во всех этих методах параметр `S` — это текст создаваемого узла. Место появления узла (первый или последний) также зависит от состояния свойства `TTreeView.SortType`:

```
type TSortType = (stNone, stData, stText, stBoth);
property SortType: TSortType;
```

Если узлы дерева как-либо сортируются, то новые узлы появляются сразу в соответствии с правилом сортировки. По умолчанию значение этого свойства равно `stNone`.

Добавляя к дереву сразу несколько узлов, следует воспользоваться парой методов `BeginUpdate` и `EndUpdate`:

```
TreeView1.Items.BeginUpdate;
ShowSubKeys(Root,1);
TreeView1.Items.EndUpdate;
```

Они позволяют отключать и снова включать перерисовку дерева на экране на момент добавления (удаления, перемещения) узлов и тем самым сэкономить подчас очень много времени.

Помимо добавления узлов в дерево программным способом можно сделать это и вручную во время разработки. При щелчке в Инспекторе объектов на свойстве `Items` запускается специальный редактор (рис. 5.4).

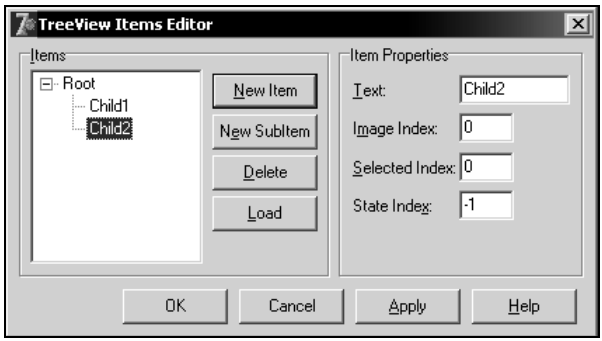


Рис. 5.4. Внешний вид редактора узлов компонента `TTreeView`

Внешний вид компонента `TTreeView` может быть весьма основательно настроен под нужды пользователя. Свойство `ShowButtons` отвечает за то, будут ли отображаться кнопки со знаком "+" и "−" перед узлами, имеющими "потомство" (дочерние узлы). Щелчок на этих кнопках позволяет сворачивать/разворачивать дерево дочерних узлов. В противном случае делать это нужно двойным щелчком на узле или установить свойство `AutoExpand` в значение `True` — тогда сворачивание и разворачивание будет происходить автоматически при выделении узлов. Свойство `ShowLines` определяет, будут ли родительские и дочерние узлы соединяться видимыми линиями. Аналогично, свойство `ShowRoot` определяет, будут ли на рисунке соединяться между собой линиями корневые узлы (если их несколько). При помощи свойства `HotTrack` можно динамически отслеживать положение текущего узла: если оно установлено в значение `True`, то текущий узел (не выделенный, а именно текущий — тот, над которым находится курсор мыши) подчеркивается синей линией.

Наконец, для оформления дерева можно использовать наборы картинок. Наборов два — в свойствах `Images` и `StateImages`. Напомним, что у каждого узла-объекта `TTreeNode` есть свойства `ImageIndex` и `StateIndex`, а вдобавок еще и `SelectedIndex`. Первая картинка предназначена для отображения типа узла, а вторая — его состояния. Можно сразу (при добавлении) указать но-

мера изображений для того или иного случая, а можно делать это динамически. Для этого существуют события:

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode)
of object;
property OnGetImageIndex: TTVExpandedEvent;
property OnGetSelectedIndex: TTVExpandedEvent;
```

Пример их использования дан в листинге 5.1 ниже — в момент возникновения этих событий следует переопределить свойство `ImageIndex` или `SelectedIndex` передаваемого в обработчик события объекта `TTreeNode`.

Свойства `StateIndex` и `StateImages` можно порекомендовать для имитации множественного выбора. Дело в том, что в отличие от `TListView`, в `TTreeView` невозможно одновременно выбрать несколько узлов. Вы можете отслеживать щелчки на узлах, и для выбранных устанавливать значение `StateIndex` в 1; под этим номером в `StateImages` поместите, например, галочку.

Изложенная информация будет неполной, если не рассказать, на какие события реагирует компонент `TTreeView`. Большинство из них происходит парами — до наступления какого-то изменения и после него. К примеру, возникновение события `OnChanging` означает начало перехода фокуса от одного узла к другому, а `OnChange` — его завершение.

Четверка событий

```
type TTVCollapsingEvent = procedure(Sender: TObject; Node: TTreeNode;
var AllowCollapse: Boolean) of object;
type TTVExpandingEvent = procedure(Sender: TObject; Node: TTreeNode;
var AllowExpansion: Boolean) of object;
property OnExpanding: TTVExpandingEvent;
property OnExpanded: TTVExpandedEvent;
property OnCollapsing: TTVCollapsingEvent;
property OnCollapsed: TTVExpandedEvent;
```

сопровождает процесс свертывания/развертывания узла, а пара

```
type TTVEditingEvent = procedure(Sender: TObject; Node: TTreeNode;
var AllowEdit: Boolean) of object;
property OnEditing: TTVEditingEvent;
type TTVEditedEvent = procedure(Sender: TObject; Node: TTreeNode;
var S: string) of object;
property OnEdited: TTVEditedEvent;
```

сопровождает редактирование его текста. Событие `OnDeletion` происходит при удалении узла. Иногда нужно сравнивать узлы между собой — если вы хотите сделать это по своим правилам, используйте событие `OnCompare`.

Наконец, те, кому и приведенных возможностей мало, могут сами рисовать на компоненте `TTreeView`. У него есть свойство `Canvas` и четыре события — `OnCustomDraw`, `OnCustomDrawItem`, `OnAdvancedCustomDraw`, `OnAdvancedCustomDrawItem`.

Перейдем теперь к компоненту `TListView`. Его еще называют *расширенным списком*. Действительно, в этот компонент заложено столько, что он перекрывает все мыслимые и немыслимые задачи по отображению упорядоченной однородной информации.

Начнем со свойства `ViewStyle`:

```
type TViewStyle = (vsIcon, vsSmallIcon, vsList, vsReport);
property ViewStyle: TViewStyle;
```

В зависимости от значения этого свойства кардинально меняется внешний вид компонента. Описание значений приведено в табл. 5.5.

Таблица 5.5. *Режимы отображения компонента* `TListView`

Значение	Внешний вид
`vsIcon`	Элементы списка появляются в виде больших значков с надписью под ними. Картинки для больших значков хранятся в свойстве `LargeImages`. Возможно их перетаскивание
`vsSmallIcon`	Элементы списка появляются в виде маленьких значков с надписью справа. Картинки для маленьких значков хранятся в свойстве `SmallImages`. Возможно их перетаскивание
`vsList`	Элементы списка появляются в колонке один под другим с надписью справа. Перетаскивание невозможно
`vsReport`	Элементы списка появляются в нескольких колонках один под другим. В первой содержится маленький значок и надпись, в остальных — определенная программистом информация. Если свойство `ShowColumnHeaders` установлено в значение `True`, колонки снабжаются заголовками

Как и для предыдущего компонента, элементы списка содержатся в свойстве `Items`. Это и есть собственно список; ничего необычного, кроме методов добавления/удаления, там нет. Каждый элемент списка (объект `TlistItem`) в свою очередь похож на компонент `TTreeNode`. Но у него есть и важное отличие — он может стать носителем большого количества дополнительной информации. Помимо свойства `Data` у него есть и свойство

```
property SubItems: TStrings;
```

При помощи этого свойства с каждым элементом списка может быть связан целый набор строк и объектов. Но как эти строки показать пользователю?

Именно они должны, по замыслу разработчиков этого элемента управления, отображаться в режиме отображения `vsReport`. Сначала следует создать необходимое количество заголовков колонок (заполнив свойство `Columns`), учитывая, что первая из них будет отведена под сам текст элемента списка (свойство `Caption`). Последующие же колонки будут отображать текст строк из свойства `Items.SubItems` (рис. 5.5).

Columns[0]	Columns[1]	...	Columns[m+1]
Items[0].Caption	Items[0].SubItems[0]	...	Items[0].SubItems[m]
Items[1].Caption	Items[1].SubItems[0]	...	Items[1].SubItems[m]
...	...	...	...
Items[n].Caption	Items[n].SubItems[0]	...	Items[n].SubItems[m]

Рис. 5.5. Так будет располагаться информация компонента `TListView` в режиме `vsReport`

Элементы в списке могут быть отсортированы — за это отвечает свойство `SortType`. Можно отсортировать элементы не только по названию (это возможно при значении `SortType`, равном `stText`), но и по данным (значения `stData` и `stBoth`), как это сделано в утилите Explorer. Для реализации такой сортировки нужно обработать события `OnColumnClick` и `OnCompare`:

```
var ColNum : Integer;
procedure TMainForm.ListView1ColumnClick(Sender: TObject; Column:
TListColumn);
begin
 ColNum := Column.Index;
 ListView1.AlphaSort;
end;

procedure TMainForm.ListView1Compare(Sender: TObject; Item1, Item2:
TListItem; Data: Integer; var Compare: Integer);
begin
if ColNum = 0 then // Заголовок
 Compare := CompareStr(Item1.Caption, Item2.Caption);
else
 Compare := CompareStr(Item1.SubItems[ColNum-1],
Item2.SubItems[ColNum-1]);
end;
```

Рассмотрим пример использования компонентов `TTreeView` и `TListView`. Где их совместное применение будет полезным? Выберем для этого отображение данных системного реестра. С одной стороны, ключи в реестре упорядочены иерархически. С другой, каждый из них может иметь несколько

разнотипных значений. Таким образом, мы почти пришли к тому же решению, что и разработчики из Microsoft, создавшие утилиту Registry Editor — слева дерево ключей, справа — расширенный список их содержимого.

Конечно, нет смысла дублировать их труд. Здесь мы ограничимся только просмотром реестра, затратив на это совсем немного усилий — четыре компонента и пару десятков строк кода. Так выглядит главная (и единственная) форма приложения Mini-Registry browser (рис. 5.6).

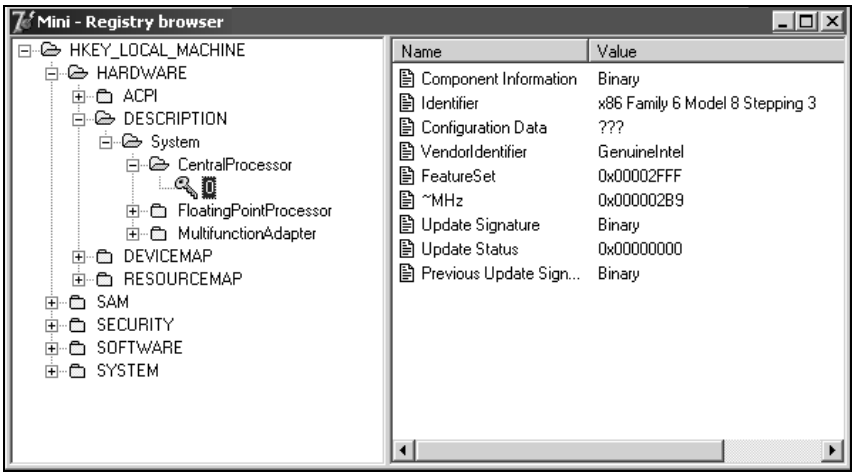


Рис. 5.6. Приложение Mini-Registry browser

А вот и весь его исходный код:

Листинг 5.1. Приложение Mini-Registry-browser, главный модуль

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Grids, Outline, ComCtrls, ImgList, ExtCtrls;

type
  TForm1 = class(TForm)
    TreeView1: TTreeView;
    ListView1: TListView;
    ImageList1: TImageList;
    Splitter1: TSplitter;
```

```
    procedure FormCreate(Sender: TObject);
    procedure TreeView1Change(Sender: TObject; Node: TTreeNode);
    procedure FormDestroy(Sender: TObject);
    procedure TreeView1Expanded(Sender: TObject; Node: TTreeNode);
    procedure TreeView1GetImageIndex(Sender: TObject; Node: TTreeNode);
  private
    { Private declarations }
  public
    { Public declarations }
   procedure ShowSubKeys(ParentNode: TTreeNode;depth: Integer);
   function GetFullNodeName(Node: TTreeNode):string;
  end;

var
  Form1: TForm1;

implementation
uses registry;
{$R *.DFM}
var  reg : TRegistry;

procedure TForm1.FormCreate(Sender: TObject);
var root : TTreeNode;
begin
Reg := TRegistry.Create;
ListView1.ViewStyle := vsReport;
with ListView1 do
begin
 with Columns.Add do
  begin
   Width := ListView1.Width div 3 - 2;
   Caption := 'Name';
  end;
 with Columns.Add do
  begin
   Width := ListView1.Width div 3 * 2 - 2;
   Caption := 'Value';
  end;
end;
 TreeView1.Items.Clear;
 Reg.RootKey := HKEY_LOCAL_MACHINE;
 Root := TreeView1.Items.Add(nil,'HKEY_LOCAL_MACHINE');
 TreeView1.Items.AddChild(root,'');
end;
```

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Reg.Free;
end;
function TForm1.GetFullNodeName(Node: TTreeNode):string;
var CurNode : TTreeNode;
begin
 Result:=''; CurNode := Node;
 while CurNode.Parent<>nil do
 begin
  Result:= '\'+CurNode.Text + Result;
  CurNode := CurNode.Parent;
 end;
end;

procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);
var s: string;
KeyInfo : TRegKeyInfo;
ValueNames : TStringList;
i : Integer;
DataType : TRegDataType;
begin
 ListView1.Items.Clear;
 s:= GetFullNodeName(Node);
 if not Reg.OpenKeyReadOnly(s) then Exit;
 Reg.GetKeyInfo(KeyInfo);
 if KeyInfo.NumValues<=0 then Exit;
 ValueNames := TStringList.Create;
 Reg.GetValueNames(ValueNames);
 for i := 0 to ValueNames.Count-1 do
  with ListView1.Items.Add do
   begin
   Caption := ValueNames[i];
   DataType := Reg.GetDataType(ValueNames[i]);
   Case DataType of
    rdString: s := Reg.ReadString(ValueNames[i]);
    rdInteger: s:= '0x'+IntToHex(Reg.ReadInteger(ValueNames[i]),8);
    rdBinary: s:='Binary';
   else s:= '???';
   end;
   SubItems.Add(s);
   ImageIndex :=1;
   end;
```

```
 ValueNames.Free;
end;

procedure TForm1.ShowSubKeys(ParentNode: TTreeNode;depth: Integer);
var ParentKey: string;
 KeyNames : TStringList;
 KeyInfo : TRegKeyInfo;
 CurNode : TTreeNode;
 i : Integer;
begin
 Cursor := crHourglass;
 TreeView1.Items.BeginUpdate;
 ParentKey := GetFullNodeName(ParentNode);
 if ParentKey<>'' then
  Reg.OpenKeyReadOnly(ParentKey)
 else
  Reg.OpenKeyReadOnly('\');
 Reg.GetKeyInfo(KeyInfo);
 if KeyInfo.NumSubKeys<=0 then Exit;
 KeyNames := TStringList.Create;
 Reg.GetKeyNames(KeyNames);
 While ParentNode.GetFirstChild<>nil do ParentNode.GetFirstChild.Delete;
 if (KeyNames.Count>0) then for i:=0 to KeyNames.Count-1 do
    begin
     Reg.OpenKeyReadOnly(ParentKey+'\'+KeyNames[i]);
     Reg.GetKeyInfo(KeyInfo);
     CurNode := TreeView1.Items.AddChild(ParentNode,KeyNames[i]);
     if KeyInfo.NumSubKeys>0 then
      begin
      TreeView1.Items.AddChild(CurNode,'');
      end;
    end;
 KeyNames.Free;
 TreeView1.Items.EndUpdate;
 Cursor := crDefault;
end;

procedure TForm1.TreeView1Expanded(Sender: TObject; Node: TTreeNode);
begin
 ShowSubKeys(Node,1);
end;

procedure TForm1.TreeView1GetImageIndex(Sender: TObject; Node: TTreeNode);
begin
```

```
with Node do
begin
 if Expanded then ImageIndex := 2
 else ImageIndex := 3;
end;
end;

end.
```

Для работы с системным реестром используется объект VCL `TRegistry`, удачно инкапсулирующий все предназначенные для этого функции Windows API. В обработчике события `OnCreate` главной формы создается объект `Reg`, а также к списку `ListView1` добавляются два заголовка (свойство `Columns`).

Пояснений требует принцип построения дерева ключей. Во-первых, это приложение отображает только один из системных ключей (а именно `HKEY_LOCAL_MACHINE`); при желании его можно заменить или добавить остальные. Во-вторых, попытка построить все "развесистое" дерево ключей сразу займет слишком много времени и наверняка не понравится пользователям. Вспомним, ведь утилита Registry Editor работает довольно быстро. Значит, придется строить дерево динамически — создавать и показывать дочерние узлы в момент развертывания родительского узла. Для этого используется событие `OnExpand` компонента `TreeView1`.

Остановимся на секунду. А какие узлы помечать кнопкой разворачивания (с пометкой "+"), ведь у родительского узла еще нет потомков? Выход из положения такой — в момент построения ключа проверить, есть ли у него дочерние. Если да, то к нему добавляется один (фиктивный) пустой ключ. Его единственная роль — дать системе поставить "+" против родительского узла.

Когда же пользователь щелкнул на кнопке, отмеченной знаком "+", и родительский узел разворачивается, фиктивный дочерний узел удаляется и вместо него создаются узлы настоящие, полученные путем сканирования реестра (см. метод `ShowSubKeys`).

Снабдим узлы картинками. Для этого в компонент `ImageList1` поместим картинки, соответствующие открытой и закрытой папкам. Напомним, что для отрисовки и смены картинок есть специальные события — `OnGetImageIndex` и `OnGetSelectedIndex`. В данном примере у двух этих событий один обработчик: развернутому узлу он сопоставляет картинку раскрытой папки, а свернутому — закрытой.

В заключение нужно сказать об очень важной особенности компонента `TListView`. Когда он отображает большой объем информации, обработка данных может затянуться очень и очень надолго и занять слишком много

памяти. Выход — перевести список в так называемый виртуальный режим. Он применяется для тех случаев, когда элементов в списке слишком много и хранить их там невозможно из соображений экономии времени или памяти. Выход из положения прост:

1. Переводим компонент в виртуальный режим установкой свойства `OwnerData` в значение `True`.
2. Сообщаем списку сколько в нем должно быть элементов установкой нужного значения `Items.Count`.
3. Чтобы предоставить нужные данные, программист должен предусмотреть обработку событий `OnData`, `OnDataFind`, `OnDataHint` и `OnDataStateChange`. Как минимум нужно описать обработчик события `OnData`.

```
TLVOwnerDataEvent = procedure(Sender: TObject; Item: TListItem)
of object;
```

Вам передается объект `TListItem`, и внутри обработчика события `OnData` необходимо динамически "оформить" его — полностью, от заголовка до картинок.

Возникает это событие перед каждой перерисовкой списка. Так что, если сбор данных для вашего списка занимает более или менее продолжительное время, лучше не связывать его с событием `OnData` — перерисовка сильно затянется. К тому же в виртуальном режиме сортировать список невозможно.

Borland прилагает к Delphi 7 прекрасный пример к вышесказанному — Virtual `ListView`. К нему и отсылаем заинтересованного читателя.

Примечание

Ответы на вопросы по компоненту `TListView` можно найти сразу в двух местах: "родном" файле справки d7vcl.hlp и файле справки Windows Win32.hlp. Во втором из них информация содержится в виде описания сообщений, посылаемых окну класса `ListView`, и соответствующих им макросов. Некоторые из них позволят вам расширить функциональные возможности компонента `TListView`. Эти макросы содержатся в файле CommCtrl.pas.

Календарь

Выбор даты — одна из часто используемых операций при вводе данных. Для облегчения этого действия разработчики Borland создали два новых элемента управления. Компонент `TMonthCalendar` инкапсулирует календарь, панель которого содержит типовую таблицу на один месяц. Компонент `TDateTimePicker` совмещает календарь с однострочным текстовым редактором, позволяя вводить даты путем выбора из календаря.

Компонент *TMonthCalendar*

Этот элемент управления представляет собой панель с календарем на один месяц (рис. 5.7). Он обладает богатыми возможностями по настройке. Основные свойства компонента, отвечающие за внешний вид и управление календарем, представлены в табл. 5.6. Их назначение достаточно прозрачно и не требует особенных комментариев.

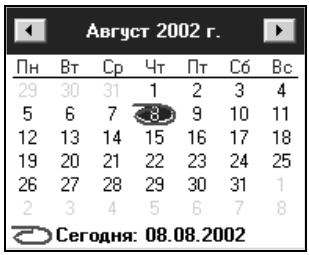


Рис. 5.7. Компонент `TMonthCalendar`

Сам календарь содержит в верхней части месяц и год, а расположенные слева и справа кнопки позволяют переходить к следующему и предыдущему месяцу. Красная окружность определяет текущую дату. Синий круг означает выбранную пользователем дату. При увеличении размеров в элементе управления отображается целое число календарей для месяцев, ближайших к текущему.

Таблица 5.6. *Основные свойства компонента* `TMonthCalendar`

Объявление	Описание
`property CalColors: TMonthCalColors;`	Определяет цвета основных элементов календаря
`property Date: TDate;`	Содержит выбранную дату
`property EndDate: TDate;`	Содержит последнюю из выбранных дат при `MultiSelect = True`. Иначе совпадает со свойством `Date`
`type TCalDayOfWeek = (dowMonday, dowTuesday, dowWednesday, dowThursday, dowFriday, dowSaturday, dowSunday, dowLocaleDefault); property FirstDayOfWeek: TCalDayOfWeek;`	Определяет день, с которого начинается неделя. Значение по умолчанию `dowLocaleDefault` соответствует установкам ОС
`property MaxDate: TDate;`	Максимальная доступная для просмотра дата

Таблица 5.6 *(окончание)*

Объявление	Описание
`property MaxSelectRange: Integer;`	Максимальная доступная для выбора дата
`property MinDate: TDate;`	Минимальная доступная для просмотра дата
`property MultiSelect: Boolean;`	При значении `True` позволяет выбирать несколько дат одновременно
`property ShowToday: Boolean;`	Включает или отключает показ текущей даты в нижней части календаря
`property ShowTodayCircle: Boolean;`	Включает или отключает выделение текущей даты красным кругом
`property WeekNumbers: Boolean;`	Включает или отключает показ номеров недель в левой части календаря

Результат выбора даты в календаре сохраняется в свойстве `Date`. При использовании возможности выбора нескольких значений одновременно в свойстве `EndDate` содержится последняя дата, а в свойстве `Date` — самая ранняя из выбранных.

Метод-обработчик

```
property OnGetMonthInfo: TOnGetMonthInfoEvent;
```

вызывается при смене месяца.

Компонент *TDateTimePicker*

Безусловно, календарь будет очень полезен пользователям. Однако было бы желательно не только выбирать даты, но и вводить их в элементы управления. Компонент `TDateTimePicker` совмещает календарь и однострочный текстовый редактор, причем календарь полностью совпадает с рассмотренным выше (оба компонента являются наследниками класса `TCommonCalendar`). Свойства и методы компонента представлены в табл. 5.7.

Таблица 5.7. *Основные свойства и методы компонента* `TDateTimePicker`

Объявление	Описание
`type TDTCalAlignment = (dtaLeft, dtaRight);` `property CalAlignment: TDTCalAlignment;`	Выравнивает панель календаря по левой или правой стороне компонента

Таблица 5.7 *(окончание)*

Объявление	Описание
`property Checked: Boolean;`	Возвращает `True`, если флажок в редакторе включен
`type TDTDateFormat = (dfShort, dfLong);` `property DateFormat: TDTDateFormat;`	Определяет формат представления даты
`type TDTDateMode = (dmComboBox, dmUpDown);` `property DateMode: TDTDateMode;`	Задает стиль компонента
`property DroppedDown: Boolean;`	Возвращает `True`, если панель календаря включена
`type TDateTimeKind = (dtkDate, dtkTime);` `property Kind: TDateTimeKind;`	Определяет возвращаемый результат — дату или время. Время можно вводить только в стиле `dmUpDown`
`property ParseInput: Boolean;`	Включает или отключает метод-обработчик `OnUserInput`
`property ShowCheckbox: Boolean;`	Управляет видимостью флажка
`type TTime = type TDateTime;` `property Time: TTime;`	Содержит установленное время
`property OnChange: TNotifyEvent;`	Вызывается при вводе даты или времени
`property OnCloseUp: TNotifyEvent;`	Вызывается при сворачивании панели календаря
`property OnDropDown: TNotifyEvent;`	Вызывается при разворачивании панели календаря

`type TDTParseInputEvent =`
`procedure(Sender: TObject; const UserString: string; var DateAndTime: TDateTime; var AllowChange: Boolean) of object;`
`property OnUserInput: TDTParseInputEvent;`
	Вызывается при прямом вводе значения в редактор.
	Параметр `UserString` содержит вводимое значение.
	Параметр `DateAndTime` содержит значение даты или времени.
	Параметр `AllowChange` управляет изменением значения

Компонент `TDateTimePicker` может обеспечивать ввод даты или времени.

Помимо календаря в элемент управления встроен флажок, который расположен в левой части редактора. Его видимостью можно управлять.

В зависимости от значения свойства `Kind` элемент управления настраивается на ввод даты или времени. Результат ввода даты сохраняется в свойстве `Date`. Дату можно выбирать из всплывающего календаря или путем перебора. Результат ввода времени сохраняется в свойстве `Time`.

Свойство `ParseInput` при значении `True` разрешает ручной ввод значения. В этом случае разработчик может использовать метод-обработчик

```
type TDTParseInputEvent = procedure(Sender: TObject; const UserString:
string; var DateAndTime: TDateTime; var AllowChange: Boolean) of object;
property OnUserInput: TDTParseInputEvent;
```

В нем можно предусмотреть необходимые действия, например проверку введенного значения:

```
procedure TForm1.DateTimePicker2UserInput(Sender: TObject;
  const UserString: String; var DateAndTime: TDateTime;
  var AllowChange: Boolean);
begin
 try
  DateAndTime := StrToDateTime(UserString);
 except
  on E: EConvertError do ShowMessage('Неверное значение');
 end;
end;
```

Обратите внимание, что здесь обязательно должно присутствовать присвоение результата ввода параметру `DateAndTime`, иначе элемент управления не получит новое значение.

Панель состояния *TStatusBar*

Этот вид элементов управления применяется уже достаточно давно. Его роль заключается в отображении различного рода справочной информации. Панель состояния инкапсулирована в компоненте `TStatusBar`.

Обычно панель состояния размещается в нижней части окна. Поэтому при переносе на форму свойство `Align` всегда имеет значение `alBottom`. Панель состояния можно разделить на произвольное число самостоятельных частей. Каждая часть описывается объектом `TStatusPanel`. Коллекция всех таких объектов находится в свойстве

```
property Panels: TStatusPanels;
```

Например, для того чтобы показать на панели состояния текущую дату и время, в методе-обработчике `OnTimer` компонента `TTimer` достаточно предусмотреть следующий код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 StatusBar1.Panels[0].Text := DateToStr(Now);
 StatusBar1.Panels[1].Text := TimeToStr(Now);
end;
```

Впрочем, панель состояния можно сделать сплошной. Для этого свойство `SimplePanel` должно иметь значение `True`. В данном случае текст панели должен содержаться в свойстве `SimpleText`.

Расширенный комбинированный список *TComboBoxEx*

Такой выпадающий список знаком пользователям со времен Windows 95 (например, список всех элементов оболочки Shell: папки My Computer, My Documents и т. п.) Соответствующий элемент управления появился в библиотеке ComCtl32 несколько позже, а в компонент он превратился только в Delphi 7.

Что отличает этот "продвинутый" выпадающий список от обычного `TComboBox`? С функциональной точки зрения основных отличий два: возможность добавлять картинки к элементам и выравнивать последние с разным отступом, имитируя иерархию.

Реализовано это следующим образом.

У компонента `TComboBoxEx`, помимо свойства `Items`, есть свойство

```
property ItemsEx: TComboExItems;
```

которое представляет собой коллекцию элементов типа `TComboExItem`. Щелкнув на этом свойстве в Инспекторе объектов, увидим типичный редактор коллекций, где каждый элемент обладает такими опубликованными свойствами:

- свойство `Caption` отвечает за заголовок элемента, каким он буден виден в списке;
- свойство `Data` — это нетипизированный указатель на прикрепляемые к элементу данные;
- отступ от левого края списка задается свойством `Indent`. В документации написано, что оно задается в пикселах. Это почти так: на самом деле одна единица значения свойства соответствует десятку пикселов;
- три номера картинок: обычный `ImageIndex`, номер для выбранного элемента `Selected ImageIndex` и `OverlayImageIndex`. Последнее свойство задает номер картинки, используемой как накладываемая маска для первых

двух. Она должна быть черно-белой: белые области прозрачны для исходной картинки, черные — нет. Все три индекса указывают на один и тот же список картинок, задаваемый свойством `Images` родительского компонента.

Дополнительные опции в расширенном выпадающем списке задаются свойством `StyleEx`. Это — множество из четырех флагов, установка которых сводится к разрешению или запрету перечисленных выше новых свойств.

Создание нового компонента на базе элементов управления из библиотеки ComCtl32

С каждой версией Internet Explorer Microsoft поставляет новую библиотеку ComCtl32 с новыми элементами управления. Программисты Borland пытаются поспеть за ними, но получается это не всегда. Так что полезно было бы и самому научиться создавать оболочку для новых и необходимых элементов управления, тем более, что это несложно. Рассмотрим это на примере.

Подходящей кандидатурой может служить редактор IP-адресов, появившийся в версии библиотеки 4.71 (Internet Explorer 4.0). Это элемент, упрощающий редактирование адресов для многих Internet-компонентов и приложений.

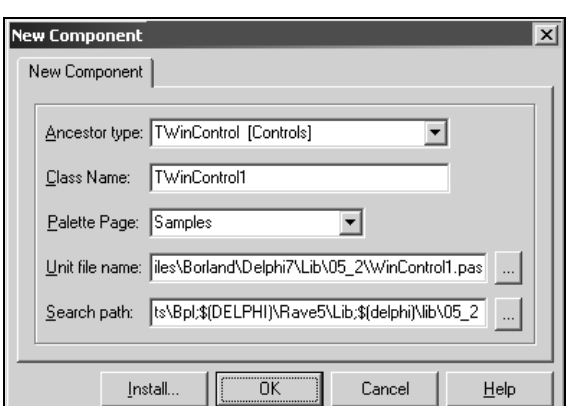


Рис. 5.8. Мастер создания новых компонентов Delphi 7

Мастер создания новых компонентов (рис. 5.8) создаст для нас шаблон. Поскольку элементы из состава библиотеки ComCtl32 есть не что иное, как окна со специфическими свойствами, наш компонент мы породим от `TWinControl`. IP-редактор представляет собой окно класса `WC_IPADDRESS`.

Название нового компонента выбрано `TCustomIPEdit`. Такая схема принята разработчиками Delphi для большинства компонентов VCL. Непосредственным предком, допустим, `TEdit` является компонент `TCustomEdit`.

Первым делом при создании компонента — особо не раздумывая — следует опубликовать типовые свойства и события, которые есть у большинства визуальных компонентов. Чтобы не занимать место в книге, позаимствуем их список у любого другого компонента из модуля ComCtrls.pas.

Далее приступим к описанию свойств, которыми будет отличаться наш компонент от других. Возможности, предоставляемые IP-редактором, описаны в справочной системе MSDN. Визуально он состоит из четырех полей, разделенных точками (рис. 5.9).

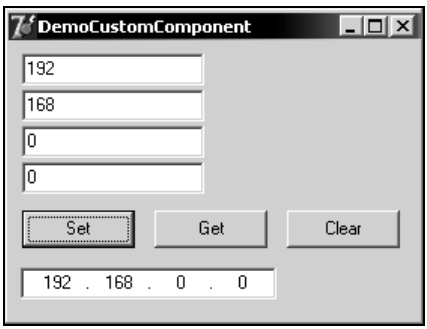


Рис. 5.9. Тестовое приложение, содержащее IP-редактор (внизу)

Для каждого из полей можно задать отдельно верхнюю и нижнюю границы допустимых значений. Это удобно, если вы планируете работать с адресами какой-либо конкретной IP-сети. По умолчанию границы равны 0—255.

Элемент управления обрабатывает шесть сообщений (см. документацию MSDN), которые сведены в табл. 5.8.

Таблица 5.8. *Сообщения, обрабатываемые элементом управления IP Address Control*

Сообщение	Назначение
`IPM_CLEARADDRESS`	Очистить поле адреса
`IPM_GETADDRESS`	Считать адрес
`IPM_ISBLANK`	Проверить, не пустое ли поле адреса
`IPM_SETADDRESS`	Установить адрес
`IPM_SETFOCUS`	Передать фокус заданному полю элемента управления
`IPM_SETRANGE`	Установить ограничения на значения в заданном поле

Кроме перечисленных, IP-редактор извещает приложение об изменениях, произведенных пользователем, путем посылки ему сообщения `WM_NOTIFY`.

Следует иметь в виду, что IP-редактор не является потомком обычного редактора (`TCustomEdit`) и не обрабатывает характерные для того сообщения `EM_XXXX`, так что название `TCustomIPEdit` отражает только внешнее сходство.

В создаваемом коде компонента первым делом нужно переписать конструктор `Create` и метод `CreateParams`. Последний метод вызывается перед созданием окна для установки его будущих параметров. Именно здесь нужно инициализировать библиотеку общих элементов управления ComCtl32 и породить новый класс окна.

```
constructor TIPEditor.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := [csCaptureMouse, csClickEvents, csDoubleClicks,
csOpaque];
  Color := clBtnFace;
  Width := 160;
  Height := 25;
  Align := alNone;
end;

 procedure TIPEditor.CreateParams(var Params: TCreateParams);
 begin
  InitCommonControl(ICC_INTERNET_CLASSES);
  inherited CreateParams(Params);
  CreateSubClass(Params, WC_IPADDRESS);
 end;
```

После создания свое значение получает дескриптор окна `Handle` (это свойство унаследовано от `TWinControl`). Все чтение/запись свойств элемента происходит путем обмена сообщениями с использованием этого дескриптора. Минимально необходимыми для работы являются свойства `IP` (задает IP-адрес в редакторе), `IPString` (отображает его в виде текстовой строки) и процедура `Clear` (очищает редактор).

Реализовано это следующим образом:

Листинг 5.2. Исходный код компонента `TCustomIPEdit`

```
unit uIPEdit;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TCustomIPEdit = class(TWinControl)
  private
    { Private declarations }
   FIPAddress: DWORD;
   FIPLimits: array [0..3] of word;
   FCurrentField : Integer;
   //procedure CMWantSpecialKey(var Msg: TCMWantSpecialKey);
     message CM_WANTSPECIALKEY;
   procedure WMGetDlgCode(var Message: TWMGetDlgCode);message
WM_GETDLGCODE;
   procedure CMDialogChar(var Message: TCMDialogChar);
   message CM_DIALOGCHAR;
   //procedure CMDialogKey(var Message: TCMDialogKey);
     message CM_DIALOGKEY;
   procedure CNNotify(var Message: TWMNotify); message CN_NOTIFY;
  protected
   { Protected declarations }
   function GetIP(Index: Integer): Byte;
   procedure SetIP(Index: Integer; Value: Byte);
   function GetMinIP(Index: Integer): Byte;
   procedure SetMinIP(Index: Integer; Value: Byte);
   function GetMaxIP(Index: Integer): Byte;
   procedure SetMaxIP(Index: Integer; Value: Byte);
   function GetIPString: string;
   procedure SetIPString(Value: string);
   function IsBlank: boolean;
   procedure SetCurrentField(Index: Integer);
   //
   procedure CreateParams(var Params: TCreateParams); override;
   procedure CreateWnd; override;
   //procedure KeyDown(var Key: Word; Shift: TShiftState);override;
   function IPDwordToString(dw: DWORD): string;
   function IPStringToDword(s: string): DWORD;
  public
    { Public declarations }
   constructor Create(AOwner: TComponent); override;
   property IP[Index: Integer]: byte read GetIP write SetIP;
   property MinIP[Index: Integer]: byte read GetMinIP write SetMinIP;
   property MaxIP[Index: Integer]: byte read GetMaxIP write SetMaxIP;
   property IPString : string read GetIPString write SetIPString;
```

```
    property CurrentField : Integer read FCurrentField write SetCurrentField;
    procedure Clear;

  end;

 TIPEdit = class(TCustomIPEdit)
 published
    property Align;
    property Anchors;
    property BorderWidth;
    property DragCursor;
    property DragKind;
    property DragMode;
    property Enabled;
    property Font;
    property Hint;
    property Constraints;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property TabStop;
    property Visible;
    property OnContextPopup;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDock;
    property OnStartDrag;
    { Published declarations }
    property IPString;
 end;

procedure Register;

implementation

uses Graphics, commctrl, comctrls;
```

```
constructor TCustomIPEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FIPAddress := 0;
  ControlStyle := [csCaptureMouse, csClickEvents, csDoubleClicks, csOpaque];
  Color := clBtnFace;
  Width := 160;
  Height := 25;
  Align := alNone;
  TabStop := True;
end;

procedure TCustomIPEdit.CreateParams(var Params: TCreateParams);
begin
  InitCommonControl(ICC_INTERNET_CLASSES);
  inherited CreateParams(Params);
  CreateSubClass(Params, WC_IPADDRESS);
  with Params do
  begin
    Style := WS_VISIBLE or WS_BORDER or WS_CHILD;
    if NewStyleControls and Ctl3D then
    begin
      Style := Style and not WS_BORDER;
      ExStyle := ExStyle or WS_EX_CLIENTEDGE;
    end;
  end;
end;

procedure TCustomIPEdit.CreateWnd;
var i: Integer;
begin
 inherited CreateWnd;
 Clear;
{ for i := 0 to 3 do
  begin
   MinIP[i] := 0;
   MaxIP[i] := $FF;
  end;}
 CurrentField := 0;
end;

procedure TCustomIPEdit.WMGetDlgCode(var Message: TWMGetDlgCode);
begin
  inherited;
```

```
  Message.Result := {Message.Result or} DLGC_WANTTAB;
end;

procedure TCustomIPEdit.CNNotify(var Message: TWMNotify);
begin
  with Message.NMHdr^ do begin
    case Code of
      IPN_FIELDCHANGED :
        begin
          FCurrentField := PNMIPAddress(Message.NMHdr)^.iField;
          {if Assigned(OnIpFieldChange) then
            with PNMIPAdress(Message.NMHdr)^ do begin
               OnIPFieldChange(Self, iField, iValue);}
        end;
    end;
  end;
end;

{procedure TCustomIPEdit.KeyDown(var Key: Word; Shift: TShiftState);
begin
 inherited KeyDown(Key, Shift);
 if Key = VK_TAB then
  if ssShift in Shift then
     CurrentField := (CurrentField - 1 + 4) mod 4
    else
     CurrentField := (CurrentField + 1) mod 4;
end;}

{procedure TCustomIPEdit.CMWantSpecialKey(var Msg: TCMWantSpecialKey);
begin
  inherited;
  //Msg.Result := Ord(Char(Msg.CharCode) = #9);
end;}

procedure TCustomIPEdit.CMDialogChar(var Message: TCMDialogChar);
begin
 with Message do
   if CharCode = VK_TAB then
    begin
     Message.Result := 0;
     if GetKeyState(VK_SHIFT)<>0 then
      begin
       if (CurrentField=0) then Exit;
       CurrentField := CurrentField - 1;
      end
```

```
    else
      begin
       if (CurrentField=3) then Exit;
       CurrentField := CurrentField + 1;
      end;
    Message.Result := 1;
    end //VK_TAB
   else
     inherited;
end;

{procedure TCustomIPEdit.CMDialogKey(var Message: TCMDialogKey);
begin
  if (Focused or Windows.IsChild(Handle, Windows.GetFocus)) and
    (Message.CharCode = VK_TAB) and (GetKeyState(VK_CONTROL) < 0) then
   begin
    if GetKeyState(VK_SHIFT)<>0 then
     CurrentField := (CurrentField - 1 + 4) mod 4
    else
     CurrentField := (CurrentField + 1) mod 4;
    Message.Result := 1;
   end
 else
  inherited;
end;}


 function TCustomIPEdit.GetIP(Index: Integer): Byte;
 begin
  SendMessage(Handle,IPM_GETADDRESS,0,longint(@FipAddress));
  case Index of
   1 : Result := FIRST_IPADDRESS(FipAddress);
   2 : Result := SECOND_IPADDRESS(FipAddress);
   3 : Result := THIRD_IPADDRESS(FipAddress);
   4 : Result := FOURTH_IPADDRESS(FipAddress);
  else Result := 0;
  end;
 end;

 procedure TCustomIPEdit.SetIP(Index: Integer; Value: Byte);
 begin
  case Index of
   1: FIPAddress := FIPAddress AND $FFFFFF or DWORD(Value) shl 24;
   2: FIPAddress := FIPAddress AND $FF00FFFF or DWORD(Value) shl 16;
```

```
  3: FIPAddress := FIPAddress AND $FFFF00FF or DWORD(Value) shl 8;
  4: FIPAddress := FIPAddress AND $FFFFFF00 or DWORD(Value);
  else Exit;
 end;
 SendMessage(Handle, IPM_SETADDRESS, 0, FIPAddress);
end;

function TCustomIPEdit.GetMinIP(Index: Integer): Byte;
begin
 if (Index<0) or (Index>3) then
  Result := 0
 else
  Result := LoByte(FIPLimits[Index]);
end;

procedure TCustomIPEdit.SetMinIP(Index: Integer; Value: Byte);
begin
 if (Index<0) or (Index>3) then Exit;
 FIPLimits[Index] := MAKEIPRANGE(HiByte(FIPLimits[Index]), Value);
 SendMessage(Handle, IPM_SETRANGE, Index, FIPLimits[Index]);
end;

function TCustomIPEdit.GetMaxIP(Index: Integer): Byte;
begin
 if (Index<0) or (Index>3) then
  Result := 0
 else
  Result := HiByte(FIPLimits[Index]);
end;

procedure TCustomIPEdit.SetMaxIP(Index: Integer; Value: Byte);
begin
 if (Index<0) or (Index>3) then Exit;
 FIPLimits[Index] := MAKEIPRANGE(Value, LoByte(FIPLimits[Index]));
 SendMessage(Handle, IPM_SETRANGE, Index, FIPLimits[Index]);
end;

procedure TCustomIPEdit.Clear;
begin
 SendMessage(Handle, IPM_CLEARADDRESS, 0, 0);
end;

function TCustomIPEdit.IsBlank: boolean;
begin
 Result:= SendMessage(Handle, IPM_ISBLANK, 0, 0) = 0;
end;
```

```
procedure TCustomIPEdit.SetCurrentField(Index: Integer);
begin
 if (Index<0) or (Index>3) then Exit;
 FCurrentField := Index;
 SendMessage(Handle, IPM_SETFOCUS, wParam(FCurrentField), 0);
end;

function TCustomIPEdit.IPDwordToString(dw: DWORD): string;
begin
  Result := Format('%d.%d.%d.%d',
  [FIRST_IPADDRESS(dw),
  SECOND_IPADDRESS(dw),
  THIRD_IPADDRESS(dw),
  FOURTH_IPADDRESS(dw)]);
end;

function TCustomIPEdit.IPStringToDword(s: string): DWORD;
var i,j : Integer;
 NewAddr, Part : DWORD;
begin
NewAddr := 0;
try
i := 0;
 repeat
  j := Pos('.', s);
  if j<=1 then
   if i<3 then
    Abort
   else
    Part := StrToInt(s)
  else
   Part := StrToInt(Copy(s, 1, j-1));
  if Part>255 then Abort;
  Delete(s, 1, j);
  NewAddr := (NewAddr shl 8) or Part;
  Inc(i);
 until i>3;
 Result := NewAddr;
 //Windows.MessageBox(0, pChar(IntToHex(FIPAddress, 8)), '', MB_Ok);
except
end;

end;
```

```
function TCustomIPEdit.GetIPString: string;
begin
 SendMessage(Handle,IPM_GETADDRESS,0, longint(@FIPAddress));
 Result := IpDwordToString(FIPAddress);
end;

procedure TCustomIPEdit.SetIPString(Value: string);
begin
 FIPAddress := IPStringToDword(Value);
 SendMessage(Handle, IPM_SETADDRESS, 0, FIPAddress);
end;

procedure Register;
begin
 RegisterComponents('Samples', [TIPEdit]);
end;

end.
```

Для удобства пользования полезно было бы добавить к компоненту `CustomIPEdit` задание диапазона для каждого из четырех составляющих и средства преобразования текстовой строки в двоичный адрес. Но это уже совсем другая история, к библиотеке ComCtl32 отношения не имеющая.

Резюме

Элементы управления — поистине неисчерпаемая тема. Надеемся, что читатель воспримет данную главу как руководство к последующему углубленному анализу того или иного элемента в свете стоящих перед ним задач.

С элементами управления вам так или иначе придется сталкиваться во всех последующих главах.

ГЛАВА 6

Элементы управления Windows XP

В Delphi 7 впервые появилась возможность настраивать пользовательский интерфейс приложений для использования в Windows XP. Для этого в состав ресурсов приложения должен включаться манифест Windows XP. Это дополнение призвано обеспечить корректное взаимодействие элементов управления приложения с системной библиотекой ComCtl32.dll версии 6, используемой в Windows XP. Собственно все особенности работы приложений под управлением Windows XP вызваны именно появлением новой версии этой библиотеки. Впрочем, об этом подробно рассказывается ниже.

В этой главе рассматриваются следующие вопросы:

- динамическая библиотека ComCtl32.dll v.6 и особенности пользовательского интерфейса Windows XP;
- что такое манифест Windows XP;
- компонент `TXPManifest`;
- как включить манифест Windows XP в ресурсы приложения;
- стили приложения и компоненты настройки цветовой палитры.

Пользовательский интерфейс Windows XP

При первом знакомстве с Windows XP сразу же обращаешь внимание на существенные изменения в оформлении пользовательского интерфейса этой операционной системы. Все нововведения основаны на использовании нового способа отрисовки элементов управления, который реализован в системной динамической библиотеке ComCtl32.dll версии 6. Эта библиотека входит в состав операционной системы Windows XP.

Примечание

Обратите внимание, что все более ранние операционные системы фирмы Microsoft используют ComCtl32.dll версии 5 с традиционным стилем оформления элементов управления. И, конечно, нельзя заставить Windows 2000 выглядеть как Windows XP простой заменой версии ComCtl32.dll. Операционная система должна уметь использовать новые возможности.

Теперь элементы управления в стиле Windows XP могут иметь собственную цветовую палитру, текстуры, стили заполнения и, самое главное, различные методы отрисовки своих составных частей. Например, элемент управления может рисовать собственное поле ввода данных обычным способом, а рамку — с использованием маски XOR.

Совокупность правил и методов поведения, оформления и отрисовки элементов управления называется *визуальным стилем* (visual style). Совместно с Windows XP поставляется один готовый стиль.

Если разработчик хочет, чтобы его приложение могло использовать новые возможности Windows XP, он должен включить в состав проекта специальный ресурс — *манифест* (manifest). Манифест представляет собой документ XML, содержащий необходимую для библиотеки ComCtl32.dll версии 6 информацию.

Такое приложение будет чувствовать себя комфортно в среде Windows XP. Но и при запуске в старых операционных системах оно просто будет использовать стандартный набор возможностей ComCtl32.dll версии 5 и выглядеть, как и любое другое приложение.

Кроме этого, Windows XP также имеет библиотеку ComCtl32.dll версии 5 и именно она используется по умолчанию.

Манифест Windows XP

Итак, начнем с манифеста. Он представляет собой документ в формате XML, содержащий всю информацию, необходимую для взаимодействия приложения и библиотеки ComCtl32.dll версии 6.

Примечание

Следует отметить, что манифесты широко используются во многих продуктах и технологиях, работающих в операционных системах Microsoft. С полной схемой XML манифеста вы можете ознакомиться в документации Microsoft MSDN.

Пример манифеста представлен в листинге 6.1.

Листинг 6.1. Манифест Windows XP

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
```

```
<assemblyIdentity
version="1.0.0.0"
processorArchitecture="X86"
name="CompanyName.ProductName.YourApp"
type="win32"
/>
<description>Your application description here.</description>
<dependency>
<dependentAssembly>
<assemblyIdentity
type="win32"
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="X86"
publicKeyToken="6595b64144ccf1df"
language="*"
/>
</dependentAssembly>
</dependency>
</assembly>
```

Обратите внимание, что в элементе

```
<dependency>
  <dependentAssembly>
    <assemblyIdentity>
```

описывается версия системной библиотеки, используемой приложением для отрисовки элементов управления.

При загрузке приложения операционная система Windows XP считывает манифест (если он есть) и получает информацию о том, что для выполнения приложения потребуется библиотека ComCtl32.dll версии 6.

Помимо этой информации манифест может содержать и другие необходимые сведения о приложении и условиях его работы. Например, общая информация о приложении и его версии представлены элементом `<AssemblyIdentity>`.

Вы можете добавить манифест в ваше приложение двумя способами:

- использовать компонент `TXPManifest`;
- добавить манифест в ресурсы приложения вручную.

Рассмотрим их.

Компонент *TXPManifest*

На странице **Win32** Палитры компонентов Delphi 7 имеется компонент `TXPManifest`. Будучи добавленным в проект, он обеспечивает компиляцию манифеста Windows XP в исполняемый файл приложения. В качестве основы используется стандартный манифест Delphi для Windows XP, содержащийся в файле ресурсов Delphi7\Lib\WindowsXP.res (листинг 6.2).

Листинг 6.2. Манифест Delphi для Windows XP

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
    type="win32"
    name="DelphiApplication"
    version="3.2.0.0"
    processorArchitecture="*"/>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        publicKeyToken="6595b64144ccf1df"
        language="*"
        processorArchitecture="*"/>
    </dependentAssembly>
  </dependency>
</assembly>
```

К сожалению, версия программного продукта (ProductVersion), а также любая другая информация о версии, содержащаяся в проекте (файлы DOF и RES) и настраиваемая в диалоге **Project Options** среды разработки Delphi, никак не влияет на содержимое манифеста. Поэтому при настройке манифеста в соответствии с потребностями приложения вам придется предварительно отредактировать файл WindowsXP.res или поправить манифест прямо в исполняемом файле. (Ввиду частых перекомпиляций проекта второй вариант представляется довольно обременительным.)

Включение манифеста Windows XP в ресурсы приложения

Так как использование стандартного компонента `TXPManifest` требует настройки исходного манифеста Delphi для каждого проекта, было бы неплохо изыскать более удобный способ. В качестве альтернативы вы можете под-

ключить манифест к файлу вашего проекта и по мере надобности редактировать его, не опасаясь, что ваша забывчивость может отразиться на версиях в манифестах других приложений.

Для начала необходимо создать исходный файл ресурса RC, включающий единственную строку:

```
1 24 "XP.manifest"
```

где `1` — номер ресурса версии библиотеки ComCtl32.dll, а `24` — номер ресурса манифеста (нумерация соответствует заголовочным файлам, распространяемых Microsoft); `"XP.manifest"` — имя файла с документом XML, содержащим манифест. Естественно, манифест нужно настроить в соответствии с потребностями вашего проекта.

Теперь нужно откомпилировать файл ресурса при помощи строчного компилятора ресурсов \Delphi7\Bin\brcc32.exe и разместить его в папке проекта.

И последняя операция — добавьте в исходный код файла проекта директиву подключения ресурса манифеста:

```
{$R xpmanifest.res}
```

В результате при компиляции проекта манифест из ресурса будет добавлен в исполняемый файл приложения.

Пример ресурсов манифеста имеется на дискете, прилагаемой к этой книге.

Визуальные стили и темы оформления

Теперь давайте более подробно разберемся с визуальными стилями и их влиянием на пользовательский интерфейс приложений.

Начиная с операционной системы Widows 95 пользователям был доступен пакет обновления Microsoft Plus!, который позволял использовать темы оформления рабочего стола Windows. *Темой* называется совокупность настроек цветов, шрифтов, курсоров и прочих ресурсов, необходимых для создания унифицированного пользовательского интерфейса.

Все параметры одной темы сохраняются в файле с расширением theme в виде секций и значений, подобно файлам INI. Существующие темы доступны для выбора в системном диалоге **Display Options**.

Визуальные стили, интегрированные в Windows XP, управляют внешним видом и поведением элементов управления. При этом визуальный стиль использует настройки параметров пользовательского интерфейса, заданные текущей темой. Для управления темами визуального стиля операционная система использует *менеджер тем*.

Визуальный стиль позволяет настраивать внешний вид элементов управления в целом и его составных частей. Правила и методы отрисовки сохраня-

ются в файле с расширением mst, который входит в состав визуального стиля.

Совместно с Windows XP поставляется только один визуальный стиль, и он составляет приятное и свежее впечатление о пользовательском интерфейсе операционной системы.

Визуальные стили в Delphi

В *гл. 8* мы детально поговорим о роли *действий* при разработке пользовательского интерфейса приложения и специальном компоненте для управления действиями — `TActionManager`. Немного забегая вперед скажем, что этот компонент является своего рода "командным пунктом", из которого должны управляться элементы управления приложения. Сейчас же нас интересует только одно свойство этого компонента

```
property Style: TActionBarStyle;
```

По умолчанию среда разработки Delphi предлагает к использованию два стиля:

- `Standard` — приложение использует системную библиотеку ComCtl32.dll версии 5;
- `Windows XP` — приложение использует системную библиотеку ComCtl32.dll версии 6 и единственный стандартный визуальный стиль Windows XP.

Эти стили применимы только к элементам управления, размещенным на панелях инструментов (`TActionToolBar`), созданных в компоненте `TActionManager`.

Однако не торопитесь возмущаться явной ограниченностью выбора. Вы можете создать собственный стиль самостоятельно. Правда, это потребует очень много услилий — ведь на основе базовых классов элементов управления вам потребуется создать собственные классы с нужным вам поведением и внешним видом.

Для этого необходимо создать класс нового визуального стиля на основе класса `TActionBarStyleEx`. Затем новый стиль регистрируется при помощи процедуры

```
procedure RegisterActnBarStyle(AStyle: TActionBarStyleEx);
```

После этого ваш стиль становится доступным для свойства `Style` компонента `TActionManager`.

Чтобы отменить стиль, используйте процедуру

```
procedure UnRegisterActnBarStyle(AStyle: TActionBarStyleEx);
```

Например, обе эти операции удобно выполнить при инициализации и деинициализации модуля, описывающего класс стиля:

Листинг 6.3. Вариант регистрации и отмены собственного визуального стиля

```
var MyStyle: TMyStyleActionBars;
...
initialization
  MyStyle := TMyStyleActionBars.Create;
  RegisterActnBarStyle(MyStyle);

finalization
  UnregisterActnBarStyle(MyStyle);
  MyStyle.Free;
end.
```

Для смены стиля приложения можно использовать глобальную переменную нового стиля (см. листинг 6.3). Ее достаточно присвоить свойству `Style`:

```
ActionManager1.Style := MyStyle;
```

При смене стиля все элементы управления, расположенные на панелях компонента `ActionManager1`, будут уничтожены и созданы заново с использованием настроек нового стиля.

Класс `TActionBarStyleEx` имеет всего несколько методов, которые необходимо перекрыть при создании собственного стиля. Все они возвращают классы объектов, используемых при создании пользовательского интерфейса. Рассмотрим их.

Функция

```
function GetStyleName: string;
```

возвращает имя стиля.

Функция

```
function GetColorMapClass(ActionBar: TCustomActionBar):
TCustomColorMapClass;
```

позволяет получить ссылку на класс компонента настройки цветовой палитры *(см. разд. "Компоненты настройки цветовой палитры" далее в этой главе)*, используемый панелью инструментов.

Следующие три метода дают информацию о классах различных типов элементов управления, используемых при проектировании пользовательского интерфейса с помощью компонента `TActionManager`.

Для того чтобы получить доступ к элементам управления, связанным со стилем, предназначен метод

```
function GetControlClass(ActionBar: TCustomActionBar; AnItem:
TActionClientItem): TCustomActionControlClass;
```

Он возвращает класс элемента управления из панели ActionBar, связанного с элементом управления AnItem. Именно эта функция вызывается при создании элементов управления в панелях инструментов компонента TActionManager.

Как уже говорилось выше, при присвоении свойству Style компонента TActionManager нового значения (экземпляра класса разработанного вами визуального стиля) уничтожаются все существующие элементы управления и затем создаются новые. И в процессе создания каждого визуального компонента вызывается функция GetControlClass нового стиля, а возвращенное ею значение используется для вызова конструктора соответствующего класса.

Аналогично, для получения класса, используемого в панели меню, применяется метод

```
function GetPopupClass(ActionBar: TCustomActionBar): TGetPopupClass;
```

и для классов кнопок панели инструментов применяется функция

```
function GetScrollBtnClass: TCustomToolScrollBtnClass;
```

А класс самой панели инструментов возвращает функция

```
function GetAddRemoveItemClass(ActionBar: TCustomActionBar):
TCustomAddRemoveItemClass;
```

Итак, после разработки и отладки собственных классов элементов управления с заданными свойствами вам потребуется создать потомка от класса TActionBarStyleEx и перекрыть все перечисленные выше функции так, чтобы они возвращали нужные классы для используемых типов элементов управления.

Theme API

Помимо описанного способа создания и управления визуальными стилями разработчик может использовать функции Theme API, разработанные Microsoft для этих целей.

Для того чтобы использовать Theme API, можно стандартным способом подключить к проекту динамическую библиотеку Theme.dll:

```
var ThemeDLL: HINST;
...
begin
 ThemeDLL := LoadLibrary('theme.dll');
 if ThemeDLL <> 0 then
 begin
 ...
 end;
end;
```

Затем можно использовать возможности этого программного интерфейса напрямую. С деталями его реализации вы можете ознакомиться в документации Microsoft MSDN.

Однако можно поступить проще. В составе Delphi 7 имеется модуль UxTheme.pas, в котором как раз и реализованы возможности Theme API. Кроме этого, модуль Themes.pas содержит классы для основных элементов управления, которые могут использоваться при создании визуальных стилей, а также класс менеджера тем `TThemeServices`.

Так как детальное обсуждение возможностей Theme API выходит за рамки этой книги, в листинге 6.4 представлен схематичный пример использования функций этого программного интерфейса. Кроме того, как и все остальные API, работающие с GUI (Graphic User Interface) операционной системы, реальный код с использованием Theme API всегда перегружен многочисленными и ужасными на вид (а на самом деле вполне безобидными) функциями, рассчитывающими области вывода, неклиентские зоны оконных элементов и т. д.

Поэтому наш пример лишь показывает, как загрузить динамическую библиотеку theme.dll и получить ссылку на тему визуального стиля для текущего окна и кнопочного элемента управления.

Листинг 6.4. Пример использования функций Theme API в Delphi

```
var DC: HDC;
    CurrentThemeData: HTHEME;

begin
 if UseThemes and InitThemeLibrary then
 try
   DC := GetWindowDC(Handle);
   try
    CurrentThemeData := OpenThemeData(0, 'button');

    CloseThemeData(CurrentThemeData);
   finally
    ReleaseDC(Handle, DC);
   end
 finally
  FreeThemeLibrary;
 end
 else
  ShowMessage('Приложение или операционная система не поддерживают
  использование Theme API');
end;
```

Функция

```
function UseThemes: Boolean;
```

проверяет способность операционной системы и приложения использовать Theme API.

Методы

```
function InitThemeLibrary: Boolean;
procedure FreeThemeLibrary;
```

соответственно инициализируют и выгружают библиотеку theme.dll.

Графический контекст `DC` наверняка понадобится при отрисовке элементов управления *(см. гл. 10)*.

Функция

```
OpenThemeData: function(hwnd: HWND; pszClassList: LPCWSTR): HTHEME;
stdcall;
```

возвращает дескриптор темы для текущего визуального стиля и класса, имя которого представлено параметром `pszClassList`.

После того как тема загружена, можно использовать разнообразные графические функции, входящие в состав программного интерфейса.

При завершении работы не забудьте освободить занятые дескрипторы графического контекста и темы. Для темы используйте функцию

```
CloseThemeData: function(hTheme: HTHEME): HRESULT; stdcall;
```

Заинтересованный читатель найдет подробное описание нужных функций Theme API в Microsoft MSDN или же может полюбопытствовать содержимым модулей UxTheme.pas и Themes.pas.

Компоненты настройки цветовой палитры

Помимо создания собственных визуальных стилей, что является делом довольно трудоемким и хлопотным, вы можете изменить внешний вид пользовательского интерфейса приложения более легким способом. Впервые в составе Палитры компонентов Delphi 7 появились специализированные компоненты, позволяющие настраивать цветовую палитру всех возможных деталей пользовательского интерфейса одновременно. Эти компоненты расположены на странице **Additional**:

- `TStandardColorMap` — по умолчанию настроен на стандартную цветовую палитру Windows;
- `TXPColorMap` — по умолчанию настроен на стандартную цветовую палитру Windows XP;

- `TTwilightColorMap` — по умолчанию настроен на стандартную полутоновую (черно-белую) палитру Windows.

Все они представляют собой контейнер, содержащий цвета для раскраски различных деталей элементов управления. Разработчику необходимо лишь настроить эту цветовую палитру и по мере необходимости подключать к пользовательскому интерфейсу приложения. Для этого снова используется компонент `TActionManager`.

Все панели инструментов (класс `TActionToolBar`), созданные в этом компоненте *(см. гл. 8)*, имеют свойство

```
property ColorMap: TCustomActionBarColorMap;
```

в котором и задается необходимый компонент цветовой палитры. Сразу после подключения все элементы управления на такой панели инструментов перерисовываются в соответствии с цветами новой палитры.

Обратите внимание, что в компоненте `TToolBar`, перенесенном из Палитры компонентов на форму вручную, это свойство отсутствует.

Все компоненты настройки цветовой палитры имеют один метод-обработчик

```
property OnColorChange: TnotifyEvent;
```

который вызывается при изменении любого цвета палитры.

Резюме

В оформлении пользовательского интерфейса операционной системы Windows XP появилось новшество — визуальные стили, которые позволяют настраивать не только внешний вид элементов управления, но и их поведение, и даже управлять отрисовкой отдельных частей элементов управления. Это стало возможным благодаря системной библиотеке ComCtl32.dll версии 6.

Любое приложение может использовать возможности этой библиотеки по созданию "продвинутых" пользовательских интерфейсов. Для этого оно должно содержать манифест — документ XML, описывающий, какую версию ComCtl32.dll используют его элементы управления.

Для отрисовки частей элементов управления приложение может использовать функции Theme API, разработанные Microsoft. Кроме этого, данный программный интерфейс позволяет управлять менеджером тем, который сменяет текущие темы для визуального стиля. Темы хорошо известны пользователям, начиная с Windows 95.

Разработчики могут создавать в Delphi 7 собственные визуальные стили, разрабатывая их на основе класса `TActionBarStyleEx`.

ГЛАВА 7

Списки и коллекции

Практически любое приложение должно уметь выполнять ряд стандартных операций по обработке каких-либо данных. К ним относятся загрузка данных при открытии приложения, представление данных в удобном виде для использования внутри приложения, сохранение данных при завершении работы. Перечисленные действия необходимы и приложениям баз данных, и играм, и научным программам.

В принципе хранение и использование наборов значений можно обеспечить при помощи хорошо всем известных массивов. Однако их прямое использование требует от разработчика дополнительных усилий. Ведь при реализации программной логики необходимо добавлять в массив новые элементы, изменять существующие и удалять ненужные. Кроме этого, часто бывает необходимо найти элемент массива по значению. Все эти операции стандартны и повторяются для наборов любых типов данных.

Для решения перечисленных задач в Delphi доступны для использования специальные классы. Помимо хранения наборов значений в них реализованы свойства, позволяющие контролировать состояние списка и методы, обеспечивающие редактирование списка и поиск в нем отдельных элементов.

Для загрузки и сохранения данных используются потоки — классы, инкапсулирующие механизмы доступа к различным хранилищам информации — файлам, памяти и т. д. Их общим предком является класс `TStream`.

Для работы со строковыми списками предназначены классы `TStrings` и `TStringList`.

Любые типы данных можно заносить в список указателей, который реализован в классе `TList`.

Использование наборов объектов (широко применяются в классах VCL), которые называются *коллекциями*, осуществляется при помощи классов `TCollection` и `TCollectionItem`.

В этой главе рассматриваются следующие вопросы:

- что такое список; как устроено основное свойство всех списков, объединяющее его элементы;
- добавление, изменение и удаление элементов списка;
- поиск заданного элемента;
- механизм выделения памяти под элементы списка;
- список строк;
- список указателей;
- чем отличается коллекция от списка;
- коллекции;
- использование потоков.

Список строк

Строковый тип данных широко используется программистами. Во-первых, многие данные действительно необходимо представлять при помощи этого типа. Во-вторых, множество функций преобразования типов позволяют представлять числовые типы в виде строк, избегая тем самым проблем с несовместимостью типов.

По этой причине в первую очередь мы займемся изучением списка строк, который инкапсулирован в классах `TStrings` и `TStringList`. Первый класс является абстрактным и служит платформой для создания реально работающих потомков. Второй класс реализует вполне работоспособный список строк. Рассмотрим эти классы подробнее.

Класс *TStrings*

Класс `TStrings` является базовым классом, который обеспечивает потомков основными свойствами и методами, позволяющими создавать работоспособные списки строк. Его прямым предком является класс `TPersistent`.

Класс `TStrings` реализует все вспомогательные свойства и методы, которые обеспечивают управление списком. При этом методы, непосредственно добавляющие и удаляющие элементы списка, не реализованы и объявлены как абстрактные.

Внимание!

Попытка прямого использования в приложении экземпляра класса `TStrings` вызовет ошибку применения абстрактного класса на этапе выполнения программы, а именно при попытке заполнить список значениями. Простая замена

типа объектной переменной списка на `TStringList` делает приложение полностью работоспособным без какого-либо дополнительного изменения исходного кода.

Классы-наследники должны перекрывать методы добавления и удаления элементов списка. Реализованный в Delphi класс `TStringList` практически полностью повторяет функциональность предка, добавляя лишь несколько новых свойств и методов. Поэтому мы не станем останавливаться подробнее на классе `TStrings`, а перейдем сразу к его работоспособному потомку `TStringList`.

Класс *TStringList*

Класс `TStringList` обеспечивает реальное использование списков строк в приложении. По существу, класс представляет собой оболочку вокруг динамического массива значений списка, представленного свойством `Strings`. Объявление свойства (унаследованное от `TStrings`) выглядит так:

```
property Strings[Index: Integer]: string read Get write Put; default;
```

Для работы со свойством используются внутренние методы `Get` и `Put`, в которых применяется внутренняя переменная `FList`:

```
type
 PStringItem = ^TStringItem;
   TStringItem = record
     FString: string;
     FObject: TObject;
   end;
   PStringItemList = ^TStringItemList;
   TStringItemList = array[0..MaxListSize] of TStringItem;

FList: PStringItemList;
```

Из ее объявления видно, что список строк представляет собой динамический массив записей `TStringItem`. Эта запись позволяет объединить саму строку и связанный с ней объект.

Максимальный размер списка ограничен константой

```
MaxListSize = Maxint div 16;
```

значение которой после нехитрых вычислений составит 134 217 727. Таким образом, видно, что строковый список Delphi теоретически конечен, хотя на практике гораздо чаще размер списка ограничивается размером доступной памяти.

Обращение к отдельному элементу списка может осуществляться через свойство `Strings` таким образом:

```
SomeStrings.Strings[i] := Edit1.Text;
```

или так:

```
SomeStrings[i] := Edit1.Text;
```

Оба способа равноценны.

При помощи простого присваивания можно задавать новые значения только тогда, когда элемент уже создан. Для добавления нового элемента используются методы `Add` и `AddStrings`.

Функция

```
function Add(const S: string): Integer;
```

добавляет в конец списка новый элемент, присваивая ему значение `S` и возвращая индекс нового элемента в списке.

Метод

```
procedure Append(const S: string);
```

просто вызывает функцию `Add`. Единственное отличие заключается в том, что метод не возвращает индекс нового элемента.

Метод

```
procedure AddStrings(Strings: TStrings);
```

добавляет к списку целый набор новых элементов, которые должны быть заданы другим списком, передаваемым в параметре `Strings`.

При необходимости можно добавить новый элемент в произвольное место списка. Для этого применяется метод

```
procedure Insert(Index: Integer; const S: string);
```

который вставляет элемент `S` на место элемента с индексом `Index`. При этом все указанные элементы смещаются на одну позицию вниз.

Для удаления элемента списка используется метод

```
procedure Delete(Index: Integer);
```

Метод

```
procedure Move(CurIndex, NewIndex: Integer);
```

перемещает элемент, заданный индексом `CurIndex`, на новую позицию, заданную индексом `NewIndex`.

А метод

```
procedure Exchange(Index1, Index2: Integer);
```

меняет местами элементы с индексами `Index1` и `Index2`.

Довольно часто в списках размещается строковая информация следующего вида:

```
'Name=Value'
```

В качестве примера можно привести строки из файлов INI или системного реестра. Специально для таких случаев в списке предусмотрено представление строк в двух свойствах. В свойстве `Names` содержится текст до знака равенства. В свойстве `Values` содержится текст после знака равенства по умолчанию. Однако символ-разделитель можно заменить на любой другой, использовав свойство

```
property NameValueSeparator: Char;
```

Доступ к значениям свойства `Values` осуществляется по значению. Например, если в списке есть строка

```
City=Saint-Petersburg
```

то значение свойства `Value` будет равно

```
Value['City'] = 'Saint-Petersburg'
```

Кроме этого, значение свойства `Value` можно получить, если известен его индекс:

```
property ValueFormIndex[Index: Integer]: string;
```

Как видно из объявления внутреннего списка `FList` *(см. выше)*, с каждым элементом списка можно связать любой объект. Для этого используется свойство

```
property Objects[Index: Integer]: TObject;
```

Свойство `Strings` элемента и свойство `Objects` связанного с ним объекта имеют одинаковые индексы. Если строка не имеет связанного объекта, то свойство `Objects` равно `Nil`. Один объект может быть связан с несколькими строками списка одновременно.

Чаще всего объекты нужны для того, чтобы хранить для каждого элемента дополнительную информацию. Например, в списке городов для каждого элемента можно дополнительно хранить население, площадь, административный статус и т. д. Для этого можно создать примерно такой класс:

```
TCityProps = class(TObject)
    Square: LongInt;
    Population: LongInt;
    Status: String;
  end;
```

Для того чтобы добавить к строке из списка объект, используется метод `AddObject`:

```
function AddObject(const S: string; AObject: TObject): Integer; virtual;
```

Обратите внимание, что в параметре `AObject` необходимо передавать указатель на объект. Проще всего это сделать таким образом:

```
SomeStrings.AddObject('SomeItem', TCityProps.Create);
```

Или же так:

```
var SPb: TCityProps;
...
SPb := TCityProps.Create;  {Создание объекта}
SPb.Population := 5000000;
...
SomeStrings.Strings[i] := 'Санкт-Петербург';
SomeStrings.Objects[i] := SPb;     {Связывание объекта и строки}
...
```

Можно поступить и подобным образом (помните, что строка уже должна существовать):

```
...
SomeStrings.Strings[i] := 'Санкт-Петербург';
SomeStrings.Objects[i] := TCityProps.Create;
(SomeStrings.Objects[i] as TCityProps).Population := 5000000;
...
```

Аналогично методу `Insert`, элемент и связанный с ним объект можно вставить в произвольное место списка методом

```
procedure InsertObject(Index: Integer; const S: string; AObject: TObject);
```

При перемещении методом `Move` вместе с элементом переносится и указатель на связанный объект.

Обратите внимание на две особенности, связанные с удалением указателей на объекты и самих связанных объектов.

При удалении элемента списка удаляется только указатель на объект, а сам объект остается в памяти. Для его уничтожения следует предпринять дополнительные усилия:

```
...
for i := 0 to SomeList.Count — 1 do
 SomeList.Objects[i].Destroy;
...
```

Если при удалении связанного объекта необходимо выполнить некоторые действия, предусмотренные в деструкторе, приведение типов

```
TCityProps(SomeList.Objects[i]).Destroy;
```

выполнять не обязательно — нужный деструктор будет вызван автоматически, хотя в данном случае приведение типов ошибкой не является.

Метод

```
procedure Clear; override;
```

полностью очищает список, удаляя все его элементы.

Помимо перечисленных, класс `TStringList` обладает рядом дополнительных свойств и методов. Вспомогательные свойства класса обеспечивают разработчика информацией о состоянии списка. Дополнительные методы осуществляют поиск в списке и взаимодействие с файлами и потоками.

Свойство только для чтения

```
property Count: Integer;
```

возвращает число элементов списка.

Так как основу списка составляет динамический массив, то для него в процессе работы должна выделяться память. При добавлении в список новой строки память для нее выделяется автоматически. Свойство

```
property Capacity: Integer;
```

определяет число строк, для которых выделена память. Вы можете самостоятельно управлять этим параметром, помня при этом, что значение `Capacity` всегда должно быть больше или равно значению `Count`.

Свойство

```
property Duplicates: TDuplicates;
```

определяет, можно ли добавлять в список повторные значения.

Тип

```
type
  TDuplicates = (dupIgnore, dupAccept, dupError);
```

определяет реакцию списка на добавление повторного элемента:

- `dupIgnore` — запрещает добавление повторных элементов;
- `dupAccept` — разрешает добавление повторных элементов;
- `dupError` — запрещает добавление повторных элементов и генерирует исключительную ситуацию.

Класс `TStringList` немыслимо представить себе без возможностей сортировки. Если вас удовлетворит обычная сортировка, то для этого можно использовать свойство `Sorted` (сортировка выполняется при значении `True`) или метод `Sort`. Под "обычной" имеется в виду сортировка по тексту строк с использованием функции `AnsiCompareStr` (т. е. с учетом национальных символов, в порядке возрастания). Если вы хотите отсортировать список по другому критерию, к вашим услугам метод:

```
type
  TStringListSortCompare = function(List: TStringList; Index1,
Index2: Integer): Integer;
procedure CustomSort(Compare: TStringListSortCompare);
```

Чтобы отсортировать список, вы должны описать функцию сравнения двух элементов с индексами `Index1` и `Index2`, которая должна возвращать следующие результаты:

- 1 — если элемент с индексом `Index1` вы хотите поместить впереди элемента `Index2`;
- 0 — если они равны;
- 1 — если элемент с индексом `Index1` вы хотите поместить после элемента `Index2`.

Для описанного выше примера с объектом-городом нужны три процедуры:

```
function SortByStatus(List: TStringList; Index1, Index2: Integer):
Integer;
begin
Result := AnsiCompareStr((List.Objects[Index1] as TCityProps).Status,
  (List.Objects[Index2] as TCityProps).Status;
end;
function SortBySquare(List: TStringList; Index1, Index2: Integer):
Integer;
begin
 if (List.Objects[Index1] as TCityProps).Square <
  (List.Objects[Index2] as TCityProps). Square) then Result := -1
 else if (List.Objects[Index1] as TCityProps).Square =
  (List.Objects[Index2] as TCityProps).Square then Result := 0
 else Result := 1;
end;
function SortByPopulation(List: TStringList; Index1, Index2: Integer):
Integer;
begin
 if (List.Objects[Index1] as TCityProps).Population <
  (List.Objects[Index2] as TCityProps). Population then Result := -1
  else
  if (List.Objects[Index1] as TCityProps). Population =
   (List.Objects[Index2] as TCityProps). Population
   then Result := 0
   else Result := 1;
end;
```

Передаем одну из процедур в метод `CustomSort`:

```
Cities.CustomSort(SortByPopulation);
```

Для поиска нужного элемента используется метод

```
function Find(const S: string; var Index: Integer): Boolean;
```

В параметре `S` передается значение для поиска. В случае успеха функция возвращает значение `True`, а в параметре `Index` содержится индекс найденного элемента.

Метод

```
function IndexOf(const S: string): Integer;
```

возвращает индекс найденного элемента `S`. Иначе функция возвращает −1.

Метод

```
function IndexOfName(const Name: string): Integer;
```

возвращает индекс найденного элемента, для которого свойство `Names` совпадает со значением параметра `Name`.

Для поиска связанных объектов используется метод

```
function IndexOfObject(AObject: TObject): Integer;
```

В качестве параметра `AObject` должна передаваться ссылка на искомый объект.

А свойство

```
property CaseSensitive: Boolean;
```

включает или отключает режим поиска и сортировки с учетом регистра символов.

Помимо свойства `Strings`, содержимое списка можно получить при помощи свойств

```
property Text: string;
```

и

```
property CommaText: string;
```

Они представляют все строки списка в виде одной строки. При этом в первом свойстве элементы списка разделены символами возврата каретки и переноса строки. Во втором свойстве строки заключены в двойные кавычки и разделены запятыми или пробелами. Так, для списка городов (Москва, Петербург, Одесса) свойство `Text` будет равно

```
Москва#$D#$AПетербург#$D#$AОдесса
```

а свойство `CommaText` равно

```
"Москва", "Петербург", "Одесса".
```

Важно иметь в виду, что эти свойства доступны не только по чтению, но и по записи. Так что заполнить список вы сможете не только циклически,

вызывая и используя методы `Add` или `Insert`, но и одним-единственным присвоением значения свойствам `Text` или `CommaText`.

Список может взаимодействовать с другими экземплярами класса `TStringList`.

Широко распространенный метод

```
procedure Assign(Source: TPersistent);
```

полностью переносит список `Source` в данный.

Метод

```
function Equals(Strings: TStrings): Boolean;
```

возвращает значение `True`, если элементы списка `Strings` полностью совпадают с элементами данного списка.

Список можно загрузить из файла или потока. Для этого используются методы

```
procedure LoadFromFile(const FileName: string);
```

и

```
procedure LoadFromStream(Stream: TStream);
```

Сохранение списка выполняется методами

```
procedure SaveToFile(const FileName: string);
```

и

```
procedure SaveToStream(Stream: TStream);
```

Перед изменением списка вы можете получить управление, описав обработчик события

```
property OnChange: TNotifyEvent;
```

а после изменения

```
property OnChanging: TNotifyEvent;
```

На дискете, прилагаемой к этой книге, вы можете ознакомиться с примером использования списков строк `DemoStrings`.

Список указателей

Для хранения списка указателей на размещенные в адресном пространстве структуры (объекты, динамические массивы, переменные) предназначен класс `TList`. Так же, как и список строк `TStringList`, список указателей обеспечивает эффективную работу с элементами списка.

Класс *TList*

Основой класса `TList` является список указателей. Сам список представляет собой динамический массив указателей, к которому можно обратиться через индексированное свойство

```
property Items[Index: Integer]: Pointer;
```

Нумерация элементов начинается с нуля.

Прямой доступ к элементам массива возможен через свойство

```
type
  PPointerList = ^TPointerList;
  TPointerList = array[0..MaxListSize-1] of Pointer;
property List: PPointerList;
```

которое имеет атрибут "только для чтения".

Так как элементы списка являются указателями на некоторые структуры, прямое обращение к составным частям этих структур через свойство `Items` невозможно. Как это можно сделать, рассказывается ниже в примере.

Примечание

В списке могут содержаться указатели на разнородные структуры. Не обязательно хранить в списке только указатели на объекты или указатели на записи.

Реализованные в классе `TList` операции со списком обеспечивают потребности разработчика и совпадают с операциями списка строк.

Для добавления в конец списка нового указателя используется метод

```
function Add(Item: Pointer): Integer;
```

Прямое присваивание значения элементу, который еще не создан при помощи метода `Add`, вызовет ошибку времени выполнения.

Новый указатель можно добавить в нужное место списка. Для этого используется метод

```
procedure Insert(Index: Integer; Item: Pointer);
```

В параметре `Index` указывается необходимый порядковый номер в списке.

Перенос существующего элемента на новое места осуществляется методом

```
procedure Move(CurIndex, NewIndex: Integer);
```

Параметр `CurIndex` определяет старое положение указателя. Параметр `NewIndex` задает новое его положение.

Также можно поменять местами два элемента, определяемые параметрами `Index1` и `Index2`:

```
procedure Exchange(Index1, Index2: Integer);
```

Для удаления указателей из списка используются два метода. Если известен индекс, применяется метод

```
procedure Delete(Index: Integer);
```

Если известен сам указатель, используется метод

```
function Remove(Item: Pointer): Integer;
```

Эти методы не уменьшают объем памяти, выделенной под список. При необходимости сделать это следует использовать свойство `Capacity`. Также существует метод `Expand`, который увеличивает отведенную память автоматически в зависимости от текущего размера списка.

```
function Expand: TList;
```

Для того чтобы метод сработал, необходимо, чтобы `Count = Capacity`. Алгоритм работы метода представлен в табл. 7.1.

Таблица 7.1. *Алгоритм увеличения памяти списка*

Значение свойства `Capacity`	**На сколько увеличится свойство `Capacity`**
<4	4
4..8	8
>8	16

Метод

```
procedure Clear; dynamic;
```

используется для удаления всех элементов списка сразу.

Для поиска указателя по его значению используется метод

```
function IndexOf(Item: Pointer): Integer;
```

Метод возвращает индекс найденного элемента в списке. При неудачном поиске возвращается −1.

Для сортировки элементов списка применяется метод

```
type TListSortCompare = function (Item1, Item2: Pointer): Integer;
procedure Sort(Compare: TListSortCompare);
```

Так как состав структуры, на которую указывает элемент списка, невозможно заранее обобщить, разработка процедуры, осуществляющей сортировку, возлагается на программиста. Метод `Sort` лишь обеспечивает попарное сравнение указателей на основе созданного программистом алгоритма *(пример сортировки см. выше в разд. "Класс TStringList")*.

Полностью все свойства и методы класса `TList` представлены в табл. 7.2.

Таблица 7.2. *Свойства и методы класса* `TList`

Объявление	Описание
`property Capacity: Integer;`	Определяет число строк, для которых выделена память
`property Count: Integer;`	Возвращает число строк в списке
`property Items[Index: Integer]: Pointer;`	Список указателей

`type`
`TPointerList = array[0..MaxListSize-1] of Pointer;`
`PPointerList = ^TPointerList;`
`property List: PPointerList;`
	Динамический массив указателей

`function Add(Item: Pointer): Integer;`	Добавляет к списку новый указатель
`procedure Clear; dynamic;`	Полностью очищает список
`procedure Delete(Index: Integer);`	Удаляет указатель с индексом `Index`
`class procedure Error(const Msg: string; Data: Integer); virtual;`	Генерирует исключительную ситуацию `EListError`. Сообщение об ошибке создается из форматирующей строки `Msg` и числового параметра `Data`
`procedure Exchange(Index1, Index2: Integer);`	Меняет местами указатели с индексами `Index1` и `Index2`
`function Expand: TList;`	Увеличивает размер памяти, отведенной под список
`function First: Pointer;`	Возвращает первый указатель из списка
`function IndexOf(Item: Pointer): Integer;`	Возвращает индекс указателя, заданного параметром `Item`
`procedure Insert(Index: Integer; Item: Pointer);`	Вставляет новый элемент `Item` в позицию `Index`
`function Last: Pointer;`	Возвращает последний указатель в списке
`procedure Move(CurIndex, NewIndex: Integer);`	Перемещает элемент списка на новое место

Таблица 7.2 *(окончание)*

Объявление	Описание
`procedure Pack;`	Удаляет из списка все пустые (`Nil`) указатели
`function Remove(Item: Pointer): Integer;`	Удаляет из списка указатель `Item`
`type TListSortCompare = function (Item1, Item2: Pointer): Integer;` `procedure Sort(Compare: TListSortCompare);`	Сортирует элементы списка

Пример использования списка указателей

Рассмотрим использование списков указателей на примере приложения DemoList. При щелчке мышью на форме приложения отображается точка, которой присваивается порядковый номер. Одновременно координаты и номер точки записываются в соответствующие свойства создаваемого экземпляра класса `TMyPixel`. Указатель на этот объект передается в новый элемент списка `PixList`.

В результате после очистки формы всю последовательность точек можно восстановить, использовав указатели на объекты точек из списка.

Список точек можно отсортировать по координате X в порядке возрастания.

Листинг 7.1. Модуль главной формы проекта DemoList

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons;

type
  TMainForm = class(TForm)
    ListBtn: TBitBtn;
    ClearBtn: TBitBtn;
    DelBtn: TBitBtn;
    SortBtn: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure ListBtnClick(Sender: TObject);
    procedure ClearBtnClick(Sender: TObject);
    procedure DelBtnClick(Sender: TObject);
    procedure SortBtnClick(Sender: TObject);
  private
    PixList: TList;
    PixNum: Integer;
  public
    { Public declarations }
  end;

  TMyPixel = class(TObject)
    FX:     Integer;
    FY:     Integer;
    FText:  Integer;
    constructor Create(X, Y, Num: Integer);
    procedure SetPixel;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const PixColor = clRed;

var CurPixel: TMyPixel;

constructor TMyPixel.Create(X, Y, Num: Integer);
begin
 inherited Create;
 FX := X;
 FY := Y;
 FText := Num;
 SetPixel;
end;

procedure TMyPixel.SetPixel;
begin
 MainForm.Canvas.PolyLine([Point(FX, FY), Point(FX, FY)]);
 MainForm.Canvas.TextOut(FX + 1, FY + 1, IntToStr(FText));
end;
```

```
function PixCompare(Item1, Item2: Pointer): Integer;
var Pix1, Pix2: TMyPixel;
begin
 Pix1 := Item1;
 Pix2 := Item2;
 Result := Pix1.FX — Pix2.FX;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
 PixList := TList.Create;
 PixNum := 1;                  {Счетчик точек}
 Canvas.Pen.Color := PixColor; {Цвет точки}
 Canvas.Pen.Width := 3;        {Размер точки}
 Canvas.Brush.Color := Color;  {Цвет фона текста равен цвету формы}
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 PixList.Free;
end;

procedure TMainForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 PixList.Add(TMyPixel.Create(X, Y, PixNum));
 Inc(PixNum);
end;

procedure TMainForm.ListBtnClick(Sender: TObject);
var i: Integer;
begin
 with PixList do
 for i := 0 to Count — 1 do
 begin
  CurPixel := Items[i];
  CurPixel.SetPixel;
 end;
end;

procedure TMainForm.ClearBtnClick(Sender: TObject);
begin
 Canvas.FillRect(Rect(0, 0, Width, Height));
end;
```

```
procedure TMainForm.DelBtnClick(Sender: TObject);
begin
 PixList.Clear;
 PixNum := 1;
end;

procedure TMainForm.SortBtnClick(Sender: TObject);
var i: Integer;
begin
 PixList.Sort(PixCompare);
 with PixList do
 for i := 0 to Count — 1 do TMyPixel(Items[i]).FText := i + 1;
end;

end.
```

Класс `TMyPixel` обеспечивает хранение координат точки и ее порядковый номер в серии. Эти параметры передаются в конструктор класса. Метод `SetPixel` обеспечивает отрисовку точки на канве формы *(см. гл. 10)*.

Экземпляр класса создается для каждой новой точки при щелчке кнопкой мыши в методе-обработчике `FormMouseDown`. Здесь же указатель на новый объект сохраняется в создаваемом при помощи метода `Add` элементе списка `PixList`. Таким образом, программа "запоминает" расположение и порядок следования точек.

Метод-обработчик `ListBtnClick` обеспечивает отображение точек. Для этого в цикле текущий указатель списка передается в переменную объектного типа `CurPixel`, т. е. в этой переменной по очереди "побывают" все созданные объекты, указатели на которые хранятся в списке.

Это сделано для того, чтобы получить доступ к свойствам объектов (непосредственно через указатель этого сделать нельзя). Второй способ приведения типа рассмотрен в методе-обработчике `SortBtnClick`.

Перед вторичным отображением точек необходимо очистить поверхность формы. Эту операцию выполняет метод-обработчик `ClearBtnClick`.

Список точек можно отсортировать по координате X в порядке возрастания. Для этого в методе-обработчике `SortBtnClick` вызывается метод `Sort` списка `PixList`. В параметре метода (переменная процедурного типа) передается функция `PixCompare`, которая обеспечивает инкапсулированный в методе `Sort` механизм перебора элементов списка алгоритмом принятия решения о старшинстве двух соседних элементов.

Если функция возвращает положительное число, то элемент `Item1` больше элемента `Item2`. Если результат отрицательный, то `Item1` меньше, чем `Item2`. Если элементы равны, функция должна возвращать ноль.

В нашем случае сравнивались координаты X двух точек. В результате такой сортировки по возрастанию объекты оказались расположены так, что первый элемент списка указывает на объект с минимальной координатой X, а последний — на объект с максимальной координатой X.

После сортировки осталось заново пронумеровать все точки. Это делает цикл в методе-обработчике `SortBtnClick`. Обратите внимание на примененный в этом случае способ приведения типа, обеспечивающий обращение к свойствам экземпляров класса `TMyPixel`.

Метод-обработчик `DelBtnClick` обеспечивает полную очистку списка `PixList`.

Коллекции

Коллекция представляет собой разновидность списка указателей, оптимизированную для работы с объектами определенного вида. Сама коллекция инкапсулирована в классе `TCollection`. Элемент коллекции должен быть экземпляром класса, унаследованного от класса `TCollectionItem`. Это облегчает программирование и позволяет обращаться к свойствам и методам объектов напрямую.

Коллекции объектов широко используются в компонентах VCL. Например, панели компонента `TCoolBar` *(см. гл. 5)* объединены в коллекцию. Класс `TCoolBands`, объединяющий панели, является наследником класса `TCollection`. А отдельная панель — экземпляром класса `TCoolBar`, происходящего от класса `TCollectionItem`.

Поэтому знание свойств и методов классов коллекции позволит успешно использовать их при работе со многими компонентами (`TDBGrid`, `TListView`, `TStatusBar`, `TCoolBar` и т. д.).

Для работы с коллекцией, независимо от инкапсулирующего ее компонента, применяется специализированный Редактор коллекции (рис. 7.1), набор

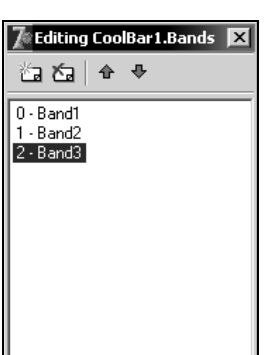


Рис. 7.1. Редактор коллекции

элементов управления которого может немного изменяться для разных компонентов.

Список Редактора объединяет элементы коллекции. При выборе одной строки из списка свойства объекта коллекции становятся доступны в Инспекторе объектов. В список можно добавлять новые элементы и удалять существующие, а также менять их взаимное положение.

Примеры использования коллекций представлены при описании соответствующих компонентов.

Класс *TCollection*

Класс `TCollection` является оболочкой коллекции, обеспечивая разработчика набором свойств и методов для управления ею (табл. 7.3).

Сама коллекция содержится в свойстве

```
property Items[Index: Integer]: TCollectionItem;
```

Полное объявление свойства в классе выглядит следующим образом:

```
property Items[Index: Integer]: TCollectionItem read GetItem write SetItem;
```

Методы `GetItem` и `SetItem` обращаются к внутреннему полю `FItems`:

```
FItems: TList;
```

Именно оно хранит коллекцию объектов во время выполнения. Отсюда следует, что коллекция представляет собой список указателей на экземпляры класса `TCollectionItem` или его наследника. Класс `TCollection` обеспечивает удобство использования элементов списка.

Таблица 7.3. *Свойства и методы класса* `TCollection`

Объявление	Описание
`property Count: Integer;`	Возвращает число элементов коллекции
`type TcollectionItemClass = class of TcollectionItem;` `property ItemClass: TcollectionItemClass;`	Возвращает класс-наследник `TCollectionItem`, экземпляры которого собраны в коллекции
`property Items[Index: Integer]: TcollectionItem;`	Коллекция экземпляров класса
`function Add: TcollectionItem;`	Добавляет к коллекции новый экземпляр класса
`procedure Assign(Source: TPersistent); override;`	Копирует коллекцию из объекта `Source` в данный объект

Таблица 7.3 *(окончание)*

Объявление	Описание
`procedure BeginUpdate; virtual;`	Отменяет перерисовку коллекции. Используется при внесении изменений в коллекцию
`procedure Clear;`	Удаляет из коллекции все элементы
`procedure EndUpdate; virtual;`	Отменяет действие метода `BeginUpdate`
`function FindItemID(ID: Integer): TCollectionItem;`	Возвращает объект коллекции с номером `ID`
`function GetNamePath: string; override;`	Возвращает имя класса коллекции во время выполнения, если коллекция не имеет владельца. Иначе возвращает название свойства класса, владеющего коллекцией
`function Insert(Index: Integer): TcollectionItem;`	Вставляет в коллекцию новый объект на место с номером `Index`

Класс *TCollectionItem*

Класс `TCollectionItem` инкапсулирует основные свойства и методы элемента коллекции (табл. 7.4). Свойства класса обеспечивают хранение информации о расположении элемента в коллекции.

Таблица 7.4. *Свойства и методы класса* `TCollectionItem`

Объявление	Описание
`property Collection: Tcollection;`	Содержит экземпляр класса коллекции, которой принадлежит данный элемент
`property DisplayName: string;`	Содержит имя элемента, которое представляет его в Редакторе коллекции
`property ID: Integer;`	Содержит уникальный номер элемента в коллекции, который не может изменяться
`property Index: Integer;`	Содержит порядковый номер элемента в коллекции. Он соответствует положению элемента в списке и может изменяться

Резюме

Списки, объединяющие элементы различных типов, играют важную роль при создании программной логики приложения. В Delphi используются три основных вида списков.

- Классы `TStrings` и `TStringList` обеспечивают применение списков строк.
- Класс `TList` инкапсулирует список указателей.
- Классы `TCollection` и `TCollectionItem` позволяют применять в компонентах и программном коде коллекции группы однородных объектов.

В среде Delphi вы можете найти еще много полезных классов общего применения. В модуле CLASSES.PAS есть класс `TBits`, обеспечивающий побитное чтение и запись информации. В модуле CONTNRS.PAS есть классы `TStack` и `TQueue` (стек и очередь), а также потомки `TList` — `TClassList`, `TComponentList` и т. д. Они помогут вам решать типовые задачи быстро и без "изобретения велосипеда".

ГЛАВА 8

Действия (Actions) и связанные с ними компоненты

С давних пор повелось, что стандарты на пользовательский интерфейс Windows-приложений Microsoft задает "явочным порядком". Первая громкая история на эту тему связана с появлением в 1994 г. Excel 2.0 for Windows, когда разработчики из Редмонда впервые применили интерфейс со многими документами (впоследствии широко известный как MDI) и даже не подумали задокументировать и опубликовать его. После справедливого возмущения широких кругов программистской общественности Microsoft исправился и теперь новые возможности интерфейса публикуются если не до выхода продукта, то, по крайней мере, ненамного позже. Вспомним, с Internet Explorer появилась панель **CoolBar** и кнопки, картинки, которые подсвечивались при прохождении над ними курсора мыши. Большинство же новинок связано с флагманским продуктом Microsoft — Office. Одна из них — весьма, кстати, полезная — это система настраиваемых пользователем меню и панелей инструментов.

В Delphi 7 разработчику предоставляется доступ к аналогичным возможностям. Правда, для работы с ними придется забыть "старый" интерфейс — компоненты `TMainMenu`, `TToolBar` — и полностью переучиться. Теперь "продвинутый" интерфейс состоит из новых компонентов `TActionManager`, `TActionMainMenuBar`, `TActionToolBar` и примкнувшего к ним `TCustomizeDlg` (страница Палитры компонентов **Additional**). Для читателя уже знакомого с действиями (Actions) названия этих компонентов покажутся знакомыми. Действительно, это — гибриды прежнего `TActionList` и панелей инструментов, плюс новые свойства и методы, плюс большое число полезных стандартных действий.

Данная глава посвящена рассмотрению принципов их использования. Сначала поговорим о понятии действия (Action) и рассмотрим компонент `TActionList`, который является кроссплатформенным (работает как в Delphi 7, так и в Kylix). Далее рассмотрим обширный набор стандартных действий.

И в заключение читатель узнает о Windows-потомке `TActionList` под названием `TActionManager` и о связанных с ним компонентах.

Действия. Компонент *TActionList*

Пользовательский интерфейс современных приложений весьма многообразен, и зачастую один и тот же результат можно получить разными способами — щелчком на кнопке на панели инструментов, выбором пункта меню, нажатием комбинации клавиш и т. п. Можно решить проблему "в лоб" и повторить один и тот же код два, три раза и т. д. Недостатки такого подхода, очевидно, не обязательно комментировать. Можно воспользоваться Инспектором объектов и назначить пункту меню тот же обработчик события, что и кнопке, благо событие `OnClick` имеет везде одинаковый синтаксис. Этот способ неплох, но при большом количестве взаимных ссылок легко запутаться.

Наконец, современный способ — это воспользоваться компонентом `TActionList` и разом решить эти проблемы. Как следует из названия, это — централизованное хранилище, где воздействия со стороны пользователя связываются с реакциями на них.

Действием (Action) будем именовать операцию, которую пользователь хочет произвести, воздействуя на элементы интерфейса. Тот компонент, на который он хочет воздействовать, называется *целью действия* (Action target). Компонент, посредством которого действие инициировано (кнопка, пункт меню), — *клиент действия* (Action client). Таким образом, в иерархии классов Delphi действие `TAction` — это невизуальный компонент, который играет роль "черного ящика", получающего сигнал от одного или нескольких клиентов, выполняющих действия над одной (или разными) целями.

Примечание

Действия могут работать только будучи объединенными в список компонентов `TActionList` или `TActionManager`. Вне этих компонентов применение действий невозможно.

Спроектировав на бумаге пользовательский интерфейс, начните работу с помещения на форму компонента `TActionList`. Он находится в Палитре компонентов на первой странице **Standard** (вот видите какое ему уделяется внимание!). После этого следует запустить редактор списка действий двойным щелчком мышью на компоненте или с помощью контекстного меню (рис. 8.1).

Теперь можно начинать добавление действий. Для программиста предусмотрен набор типовых, наиболее часто встречающихся действий, которые описаны в следующем разделе.

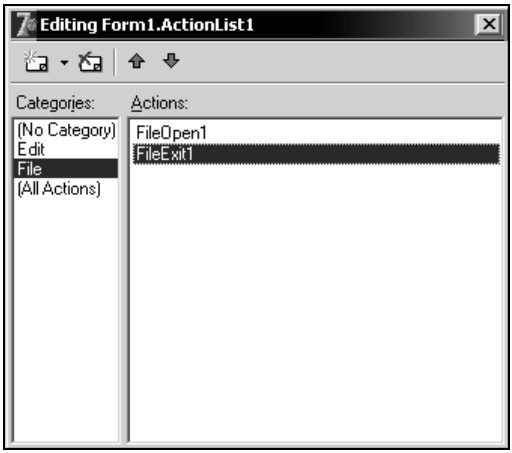


Рис. 8.1. Внешний вид редактора действий компонента `TActionList`

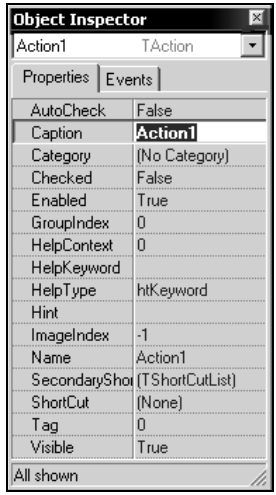


Рис. 8.2. Опубликованные свойства объекта `TAction`

Помимо них можно вставить и обычное действие, которое получит имя `Action1`. Итак, что же из себя представляет действие? Его опубликованные свойства показаны на рис. 8.2. Рассмотрим их по группам.

События, связанные с действиями

Компонент `TAction` реагирует на три события: `OnExecute`, `OnUpdate` и `OnHint`. Первое — и самое главное — должно быть как раз реакцией на данное действие. Это событие возникает в момент нажатия кнопки, пункта меню — короче, при поступлении сигнала от клиента действия. Здесь — как правило — и пишется обработчик. Почему "как правило"? Потому что схема обработки сигнала 4-этапная:

1. Сначала вызывается обработчик события `OnExecute` списка действий `TActionList`:

   ```
   property OnExecute: TActionEvent;
   TActionEvent = procedure (Action: TBasicAction; var Handled: Boolean)
   of object;
   ```

   Если обработчик этого события вами не предусмотрен, или в параметре `Handled` он вернул значение `False`, происходит генерация следующего события — шаг 2.

2. Вызывается обработчик события `OnActionExecute` глобального объекта `Application` (тип события тот же — `TActionEvent`). Если и оно не обработало сигнал действия, переходим к следующему шагу.

3. Вызывается обработчик события `OnExecute` самого действия (объекта типа `TAction` или его потомка).
4. Если первые три шага не обработали ситуацию (вернули `False`), то, вероятно, это было связано с неправильной целью (`Target`) действия. В качестве "последнего шанса" приложению посылается сообщение `CM_ACTIONEXECUTE`. В этом случае происходит поиск другой цели для данного действия *(об алгоритме поиска цели см. ниже)*.

Первые две возможности в этом списке используются относительно редко. Тем не менее они полезны, если вам нужно глобально добавлять/удалять/разрешать/запрещать действия.

Введение события `OnUpdate` является очень хорошей находкой, о нем напишем подробно. И автор этих строк, и, возможно, вы потратили немало времени, чтобы в разрабатываемых программах элементы управления находились в актуальном состоянии. Если, скажем, вашей программой открыт первый файл, то нужно активировать ряд кнопок и пунктов меню (**Save**, **Save as**, **Print** и т. п.); как только закрыт последний — отключить их. Если в буфере обмена есть что-то подходящее, необходимо активизировать пункт меню и кнопку **Paste**, если нет — отключить. В результате код, отслеживающий это, у неопытных программистов "размазывается" по всему приложению. А ведь можно поступить проще. Событие `TAction.OnUpdate` возникает в моменты простоя приложения, т. е. тогда, когда оно не занято обработкой сообщений (цикл содержится в методе `Idle` объекта `Application`). Это гарантирует, что оно возникнет ДО ТОГО, как пользователь щелкнет мышью и увидит выпадающие пункты меню; поэтому можно успеть обновить их состояние. Пример использования события `OnUpdate`:

```
procedure TForm1.PasteActionUpdate(Sender: TObject);
begin
   TAction(Sender).Checked := Clipboard.HasFormat(CF_TEXT);
end;
```

Примечание

Перед вызовом события `OnUpdate` также происходит 4-этапная последовательность действий, точно такая же, как при `OnExecute`.

Третье событие имеет такой тип:

```
THintEvent = procedure (var HintStr: string; var CanShow: Boolean)
of object;
```

Оно вызывается тогда, когда от элемента управления требуется показать подсказку, связанную с данным действием. В обработчике события можно указать, будет ли что-нибудь показываться (параметр `CanShow`) и, если да, то что именно (параметр `HintStr`).

Это были события, относящиеся к компоненту TAction. Сам компонент TActionList также имеет три события: OnExecute, OnUpdate и OnChange. О первых двух мы уже сказали; третье происходит в момент изменения списка (добавления или удаления действий).

Свойства, распространяемые на клиентов действия

Если у нескольких кнопок или пунктов меню общий обработчик, разумно потребовать, чтобы у них были и другие общие свойства. Так оно и реализовано в Delphi. В табл. 8.1 перечислены свойства, чье значение автоматически распространяется на всех клиентов данного действия.

Таблица 8.1. *Свойства компонента TAction, автоматически распространяемые на всех его клиентов*

Свойство	Назначение
property Caption: string;	Заголовок, связанный с действием
property Hint: string;	Подсказка к действию
property Enabled: Boolean;	Устанавливает, разрешено/запрещено ли действие
property Checked: Boolean;	Устанавливает, отмечено ли действие
property GroupIndex: Integer;	Индекс группы, в которую входит действие. Объекты TAction с одним значением этого свойства (причем бóльшим нуля) имеют зависимое переключение. Если свойство Checked любого объекта из группы устанавливается в True, у остальных оно сбрасывается в False
property AutoCheck: boolean;	Установка в True автоматически меняет значение свойства Checked на противоположное после каждого выполнения действия
property ImageIndex: Integer;	Индекс картинки в общем наборе картинок (набор указывается в свойствах родительского TActionList)
property HelpType: THelpType;	Указывает на тип значения, связывающего действие с разделом системы помощи (htKeyword/htContext)
property HelpContext: THelpContext;	Если свойство HelpType установлено в htContext, это свойство содержит ID раздела системы помощи

Таблица 8.1 *(окончание)*

Свойство	Назначение
`property HelpKeyword: string;`	Если свойство `HelpType` установлено в `htKeyword`, то свойство содержит ключевое слово (термин), по которому происходит открытие соответствующего раздела системы помощи

Вы привыкли к программам с картинками в меню и на панелях инструментов? Действие также можно снабдить картинкой. Компонент `TActionList` связывается со списком картинок `TImageList`, а действие `TAction` — с конкретной картинкой через свойство `ImageIndex`. Таким образом, все элементы управления, связанные с действием, — кнопки и пункты меню — будут иметь одну и ту же картинку, как показано на рис. 8.3. Впрочем, это относится ко всем остальным свойствам из табл. 8.1.

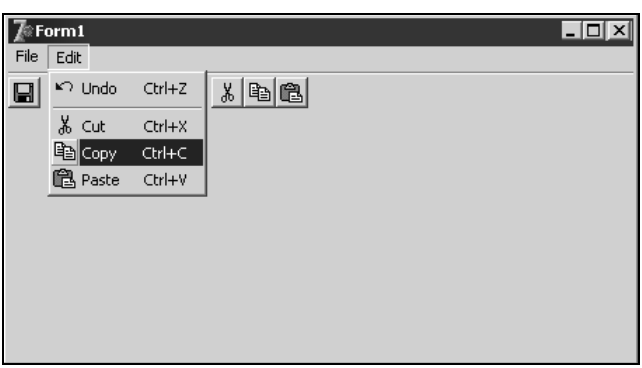


Рис. 8.3. Меню и панель инструментов используют один список действий

Прочие свойства

Чтобы связать с действием комбинацию "горячих" клавиш, выберите одну из возможных комбинаций в редакторе свойства `ShortCut`. Более того, в Delphi 7 существует возможность добавлять не одну, а множество комбинаций "горячих" клавиш. Вторая и последующие должны быть заданы в свойстве `SecondaryShortCuts`. Когда вызывается редактор этого свойства, пользователь видит обычный редактор набора строк. И вводить комбинации нужно по принципу "как слышится, так и пишется": например <Ctrl>+<F12>, <Shift>+<Alt>+<0> и т. п., по одной комбинации на каждой строке.

Для упорядочивания все действия разбиты на категории:

```
property Category: string;
```

Это свойство содержит условное название категории, к которой относится действие, например, **File**, **Edit**, **Help** и т. п. Роль категории сводится к тому, чтобы объединить сходные действия при показе в `ActionList` или `ActionManager`. Названия категорий вы видите на панели меню на самом верхнем уровне.

Иногда программисту все-таки необходимо знать, какой конкретно клиент — меню, кнопка — привел к выполнению действия. Узнать это можно, воспользовавшись значением свойства компонента `TAction`:

```
property ActionComponent: TComponent;
```

Перед вызовом `OnExecute` это свойство содержит указатель на клиента, инициировавшего действие. После вызова значение свойства очищается.

После того как определены действия и написан код, реагирующий на них, осталось поставить завершающую точку и связать их с пользовательским интерфейсом. У большого числа элементов управления (например, всех кнопок и пунктов меню) есть опубликованное свойство `Action`. Если действия к этому моменту описаны, то на панели инструментов из выпадающего списка среди возможных значений этого свойства достаточно выбрать нужное действие.

Стандартные действия

Те, кто собирается пропустить этот раздел, считая, что в нем описаны очевидные вещи, сильно ошибаются. Как раз в применении стандартных действий разработчики Borland продвинулись очень сильно. Кто хочет в этом убедиться, может открыть пример WordPad, поставляемый с Delphi 7. Полнофункциональный текстовый редактор, построенный полностью на стандартных действиях, содержит всего две строчки кода.

Шаблоны и заготовки для типовых меню и кнопок появились еще в самой первой версии Delphi. Но в шестой версии действия действительно стали действиями. Это значит, что раньше заготовка содержала только подходящий заголовок. Теперь они содержат в себе все субкомпоненты, весь программный код и делают всю необходимую работу сами.

Возьмем, например, действие `TFileOpen`. Оно уже содержит внутри компонент типа `TOpenDialog`, показывающий список открываемых файлов. Вместо ручного программирования процедуры задания имени файла теперь нужно написать обработчик события `TFileOpen.OnAccept` (если пользователь ввел в диалоге кнопку **OK**) или `OnCancel` (если отказался от открытия файла). Вот так выглядит весь программный код приложения WordPad:

```
procedure TForm1.FileOpen1Accept(Sender: TObject);
begin
```

```
  RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
end;

procedure TForm1.FileSaveAs1Accept(Sender: TObject);
begin
  RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);
end;
```

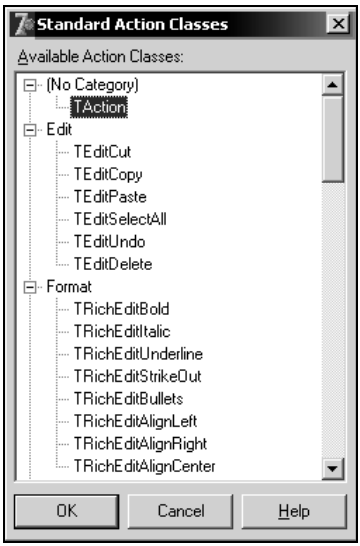


Рис. 8.4. Окно выбора стандартных действий

С точки зрения программирования стандартное действие — это класс-потомок `TCustomAction`. Классы действий описаны в трех модулях: более распространенные в `StdActns`, менее — в `ExtActns`, действия с наборами данных содержатся в `DBActns`. Наконец, два действия, работающие со списками, — `TStaticListAction` и `TVirtualLitAction` — описаны в отдельном модуле `ListActns`.

Для выполнения ряда стандартных действий нужно определить понятие "цели" действия (Action Target). Под целью понимается компонент, в отношении которого будет совершаться данное действие. Так, операции редактирования могут выполняться, когда на форме активен текстовый элемент управления (`TEdit`, `TMemo` и т. п.). У любого действия (потомка `TBasicAction`) есть три метода:

```
function HandlesTarget(Target: TObject): Boolean; virtual;
procedure UpdateTarget(Target: TObject); virtual;
procedure ExecuteTarget(Target: TObject); virtual;
```

Метод `HandlesTarget` проверяет, применимо ли действие к цели `Target`. Если да, то действие производится вызовом метода `ExecuteTarget`. Если нет, поиск подходящей цели продолжается.

Цель в Delphi 7 определяется по следующему правилу:

- первым кандидатом является активный элемент управления на форме (свойство `ActiveControl`);
- если такового нет или он не является подходящим (метод `HandlesTarget` вернул значение `False`), целью становится текущая форма, получившая сигнал о действии;
- если и она не подходит, происходит рекурсивный перебор всех компонентов на форме в поисках первого подходящего.

В ряде случаев вы можете произвести действие над желаемым компонентом, вызвав метод `ExecuteTarget` и передав в него в качестве параметра этот компонент.

Примечание

Стандартные действия редактирования, чьи имена начинаются с `TEdit`, и поиска (`TSearch...`) применимы только к потомкам компонента `TCustomEdit`. Стандартные действия расширенного редактирования, имена которых начинаются с `TRichEdit`, применимы только к потомкам `TCustomRichEdit`. Оконные стандартные действия (упорядочивание, смена, закрытие дочерних окон; имена начинаются с `TWindow`) применимы только к формам многодокументного интерфейса, чье свойство `FormStyle` установлено в `fsMDIForm` (рис. 8.4).

Многие классы стандартных действий не требуют элемента управления — цели. Так устроены все действия, вызывающие стандартные диалоговые окна (выбор файла, цвета, шрифта, настройка принтера и т. п.). Чтобы отреагировать на такое действие, нужно написать обработчики следующих событий:

```
property BeforeExecute: TNotifyEvent;
property OnAccept: TNotifyEvent;
property OnCancel: TNotifyEvent;
```

Первое возникает до показа диалога, второе — после нажатия пользователем кнопки **OK**, третье — после нажатия **Cancel**.

Примечание

Поскольку диалоги входят в действия в качестве дочерних компонентов, вы можете реагировать и на все "дочерние" события, которые происходят в соответствующем диалоге (`OnShow`, `OnCanClose`, `OnClose` и т. п.)

Поместив на форму стандартные действия, вы заметите, что все они имеют предопределенное значение свойства `ImageIndex`. Если так, то где изображе-

ние, на которое эти индексы указывают? Вы можете раздобыть его, открыв демонстрационный проект WordPad (папка Demos\ActionBands в поставке Delphi 7). Откройте редактор компонента `ImageList1` и экспортируйте весь список в виде одного большого файла формата BMP.

Категория *Edit*

В эту категорию входят компоненты, которые работают с редактируемыми элементами — потомками `TCustomEdit`. Это, к примеру, `TEdit`, `TMemo`, `TMaskedEdit`, `TRichEdit`, новый компонент `TLabeledEdit` и др. Причем целью может являться не любой редактируемый элемент, а только тот, что имеет фокус ввода. К категории относятся: `TEditCut`, `TEditCopy`, `TEditPaste`, `TEditSelectAll`, `TEditDelete`, `TEditUndo`.

Категория *Search*

Действия поиска и замены тоже производятся только над потомками `TCustomEdit`. И это не прихоть разработчиков Borland: это сделано для вашего же удобства. Раньше для поиска приходилось самому программировать события `OnFind` и `OnReplace` соответствующих диалогов, а сейчас требуемый код уже содержится внутри действий.

К компонентам этой категории относятся: `TSearchFind`, `TSearchFindFirst`, `TSearchFindNext`, `TSearchReplace`.

Категория *Help*

С помощью этих действий (табл. 8.2) вы можете вызвать справочную систему вашего приложения.

Таблица 8.2. *Стандартные действия категории* ***Help***

Компонент	Назначение
`THelpContents`	Показывает оглавление системы справки
`THelpOnHelp`	Показывает подсказку по использованию системы справки
`THelpContext`	Показывает справку по контексту активного элемента управления (причем он должен быть ненулевым)
`THelpTopicSearch`	Показывает окно поиска системы справки

Категория *File*

Эти действия скорее всего будут наиболее востребованы разработчиками. И они же являются довольно простыми в использовании. `TFileOpen`,

TFileSaveAs, TFilePrintSetup — это оболочки над соответствующими диалогами. О том, как работать с такими действиями, описано выше. Действие TFileExit вообще не требует комментариев — оно просто завершает приложение, закрывая его главную форму.

Особняком стоит только TFileRun!!!

Категория *Dialog*

Эта категория примыкает к предыдущей, в ней содержатся остальные пять типовых действий-диалогов: TPrintDlg, TColorSelect, TFontEdit (из модуля StdActns), TOpenPicture, TSavePicture (модуль ExtActns).

Категория *Window*

Эти действия стоит включать в интерфейс, только если вы используете многодокументный интерфейс (MDI). Названия компонентов говорят сами за себя: TWindowClose, TWindowCascade, TWindowTileHorizontal, TWindowTileVertical, TWindowMinimizeAll, TWindowArrange.

Категория *Tab*

Здесь всего два компонента — TNextTab и TPreviousTab. Если цель действия — набор страниц (TPageControl), они переключат его на следующую и предыдущую страницу соответственно.

Категория *List*

В этой категории выделяются две группы действий. Первые пять из них (табл. 8.3) автоматизируют работу с выделенными элементами списков. Оставшиеся два — TStaticListAction и TVirtualListAction — требуют отдельного рассмотрения.

Таблица 8.3. *Действия по работе с выделенными элементами списков*

Действие	Назначение
TListControlSelectAll	Выделяет все элементы списка. Активно, только если у списка свойство MultiSelect установлено в значение True
TListControlClearSelection	Отменяет выделение элементов в списке
TListControlDeleteSelection	Удаляет выделенные элементы
TListControlCopySelection	Копирует выделенные элементы списка в список Destination

Таблица 8.3 *(окончание)*

Действие	Назначение
`TListControlMoveSelection`	Переносит выделенные элементы списка в список `Destination`

Действия работают с компонентом `TListBox`, а в среде Kylix — еще и с `TListView` (не путать с одноименным компонентом для Windows — он не годится для данной категории). Подходит также и `TComboBox`.

В отличие от многих других действий члены этой категории могут явно связываться с нужным вам списком. Если задано значение свойства `ListControl`, то все действия производятся над ним. Если нет, то выбирается активный список из числа имеющихся на форме.

Особняком стоят два действия — `TStaticListAction` и `TVirtualListAction`

По замыслу разработчиков они являются централизованными хранилищами элементов для многих списков. Причем элементы списка могут храниться сразу с заданными картинками (т. е. свойствами `ImageIndex`) и указателями на сопутствующие данные.

Дальнейшее просто — разработчик выбирает нужные компоненты `TListBox`, `TComboBox` и т. п. и в их свойстве `Action` указывает на действие — хранилище. Опубликовано свойство `Action` у компонента `TComboBoxEx` (впервые появившегося в Delphi 6). У остальных потомков `TControl` это свойство относится к группе видимости `public`, поэтому вы можете сделать присвоение при запуске приложения (в методе `OnCreate` главной формы).

Если действие и компонент-список связаны, то должны происходить две вещи:

- при изменении текущего элемента в любом из компонентов происходит синхронное изменение во всех остальных;
- когда пользователь выбирает один из элементов списка, выполняется действие, связанное с этим списком, и вызывается метод-обработчик

```
type TItemSelectedEvent = procedure(Sender: TCustomListAction;
Control: TControl) of object;
property OnItemSelected: TItemSelectedEvent;
```

Категория *Internet*

Здесь всего три — типовых для пользователя Сети — действия.

Действие `TBrowseURL` открывает URL, заданный в одноименном свойстве. Поскольку это происходит при помощи функции `ShellExecute`, для просмотра открывается браузер, зарегистрированный в системе по умолчанию.

Действие `TSendMail` запускает программу — почтового клиента для отправки письма (с помощью интерфейса MAPI). Текст письма вы можете задать в свойстве `Text`. Но! Ни получателя, ни тему, ни вложений задать нельзя — это придется делать вручную в почтовой программе. При желании полностью автоматизировать процесс отправки вам придется породить дочерний компонент от действия `TSendMail`, где и перекрыть метод `ExecuteTarget`. Исходные тексты — в модуле ExtActns.

Наконец, самый сложный компонент `TDownloadURL`. Он позволяет загрузить содержимое с адреса URL и сохранить его на локальной машине под именем `FileName`.

Поскольку загрузка — процесс долгий, в то время, пока она происходит, периодически возникает событие

```
property OnDownloadProgress: TDownloadProgressEvent;
TDownloadProgressEvent = procedure(Sender: TDownLoadURL; Progress,
    ProgressMax: Cardinal; StatusCode: TURLDownloadStatus; StatusText:
    String;
    var Cancel: Boolean) of object;
```

Параметры обработчика этого события следующие.

- `Progress` и `ProgressMax` — текущее и максимальное значение показателя хода скачивания. Во-первых, не все HTTP-серверы правильно сообщают о размере ответа; во-вторых, для некоторых типов файлов (например, HTML) эти параметры вычисляются не всегда верно (вы можете это видеть в Internet Explorer); в-третьих, из-за маршрутизации пакетов ожидать ритмичного изменения параметра `Progress` не следует. Поэтому пользователю надо показывать соотношение `Progress/ProgressMax`.

Примечание

Значение `ProgressMax` может быть равно нулю. В этом случае о ходе загрузки численно судить нельзя. Информацию несут другие параметры события.

- `StatusCode` и `StatusText` — код, описывающий текущее состояние операции и соответствующий ему текст. Список возможных кодов содержит около 30 значений. Для тех, кто знает протокол HTTP и хочет разобраться в этом глубже, следует обратиться к описанию интерфейса `IBindStatusCallback` в MSDN. Если же вам достаточно показать пользователю текст, то он содержится во втором параметре. По содержанию он представляет примерно то же, что вы видите при загрузке файлов с помощью Internet Explorer.
- `Cancel` — этот параметр одинаков для всех долго продолжающихся операций. Установив его в значение `True`, вы можете прервать выполнение загрузки.

Категория *Format*

Действия этой категории представляют собой расширенные операции редактирования для "продвинутого" редактора `TRichEdit`. Эти операции должны быть знакомы вам по программе WordPad из состава Windows. В крайнем случае откройте демонстрационный пример Delphi с тем же названием — там присутствуют действия настоящей категории и подавляющее большинство остальных. В списке присутствуют `TRichEditBold`, `TRichEditItalic`, `TRichEditUnderline`, `TRichEditStrikeout` (установка стиля шрифта), `TRichEditBullets` (значки абзацев), `TRichEditAlignLeft`, `TRichEditAlignRight`, `TRichEditAlignCenter` (выравнивание текста).

Категория *Dataset*

Эти действия можно увидеть, например, в качестве кнопок на любом компоненте TDBNavigator: `TDataSetFirst`, `TDataSetPrior`, `TDataSetNext`, `TDataSetLast`, `TDataSetDelete`, `TDataSetInsert`, `TDataSetEdit`, `TDataSetPost`, `TDataSetCancel`, `TDataSetRefresh`. Читатель задаст вопрос: а как действие связывается с набором данных? Очень просто: через дополнительное (для данной категории) свойство `DataSource`. Если источник данных существует и связан с имеющимся набором данных (свойство `DataSource.DataSet`), то действие выполняется над ним.

Категория *Tools*

Здесь содержится один-единственный член: `TCustomizeActionBars`. Будучи вызванным, это действие вызывает диалог настройки панелей действий, относящихся к компоненту `TActionManager`, о котором, собственно, сейчас и пойдет речь.

Компонент *TActionManager*

Если вы не думаете о переносе своего приложения в среду Linux, то имеются все основания воспользоваться потомком `TActionList` — компонентом `TActionManager` (далее в тексте — менеджер действий). Более современный и "продвинутый" он обеспечит вас многими дополнительными возможностями. Итак, обо всем по порядку.

Будучи применен сам по себе, компонент `TActionManager` ничем не отличается от предшественника. Отличия проявляются, если действия из этого компонента разместить на специальных панелях — `TActionMainMenuBar` (будем называть его панелью главного меню) и `TActionToolBar` (далее — панель действий).

На первой странице редактора `TActionManager` (вызывается двойным щелчком или командой **Customize** из контекстного меню; показан на рис. 8.5)

как раз и содержится список всех панелей, связанных с данным менеджером действий. Вы можете добавить новый или убрать компонент `TActionToolBar` нажатием кнопок **New** и **Delete** соответственно. С компонентом `TActionMainMenuBar` так по понятным причинам поступить нельзя — меню полагается иметь одно.

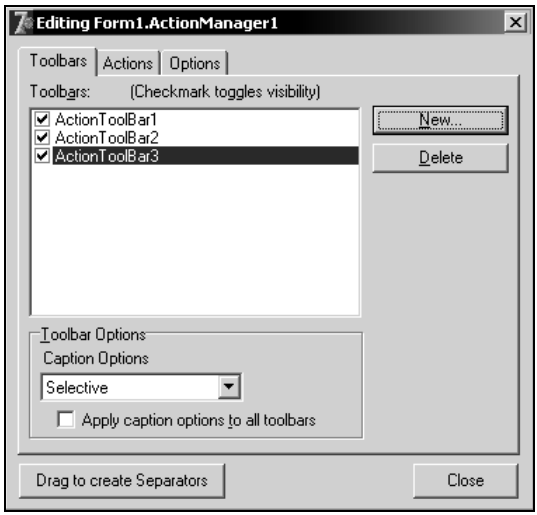


Рис. 8.5. Первая страница редактора свойств компонента `TActionManager`

Самый простой и рекомендованный Borland способ для связи действий с одной стороны и панелей меню и инструментов с другой — это перетаскивание (Drag-and-Drop). На второй странице редактора содержится список всех действий по категориям. И отдельное действие, и целую категорию можно брать и тащить мышью на нужную панель (рис. 8.6).

Когда вы перетаскиваете действие на панель, на нем появляется специальный компонент, похожий на пункт меню или кнопку. Его роль — служить клиентом данного действия. Поэтому, естественно, он сразу и автоматически получает нужные `Caption`, `ImageIndex`, `Hint` и прочие общие для всех клиентов свойства, о которых говорилось выше. Класс этого клиента — `TActionClientItem`; будем называть их псевдокнопками или псевдоэлементами.

При перетаскивании нет особых сложностей, но надо иметь в виду следующие аспекты:

- при перетаскивании всей категории на панель главного меню она появляется в виде пункта меню верхнего уровня и содержит при этом все свои дочерние действия;
- при перетаскивании всей категории на панель действий создаются псевдокнопки для всех дочерних действий в категории. Логично поступить по

принципу "одна категория — одна панель действий", это будет полезно для настройки интерфейса пользователем;

- если вы ошиблись при перетаскивании, не нажимайте кнопку **Delete** — при этом удалится не только псевдокнопка, но и само действие. Перетяните ненужный псевдоэлемент за пределы панелей действий, тогда он будет удален;
- если вы уже перетянули категорию на панель главного меню, а потом решили добавить к ней действия, то вам придется убрать соответствующий ей псевдоэлемент и перетянуть всю категорию заново. Либо воспользоваться ручным способом, который описывается ниже.

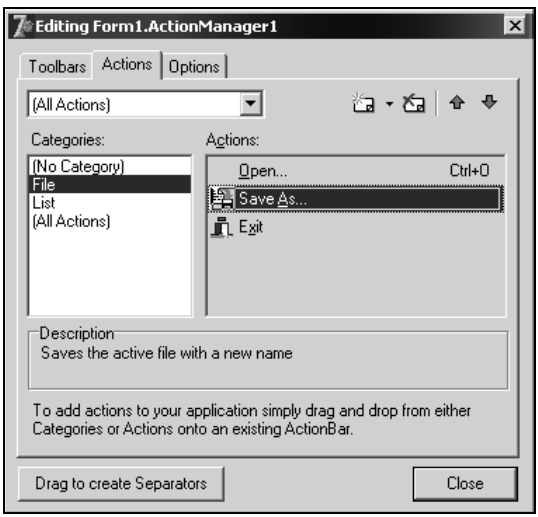


Рис. 8.6. Страница действий редактора свойств компонента `TActionManager`

Изменение и настройка внешнего вида панелей

Мы подошли к совсем новому свойству панелей — `TActionMainMenuBar`. Теперь — как в Microsoft Office — возможно прятать редко используемые пункты меню. В самом деле, интерфейс программ подчас настолько сложен, что используют его на 100% минимальное количество пользователей. Поэтому элементы интерфейса, которые пользователь не задействовал в каком-то числе предыдущих запусков, автоматически прячутся.

Что и когда прятать, определяется свойством

```
property PrioritySchedule: TStringList;
```

значение которого по умолчанию приведено в табл. 8.4. В левой колонке содержится общее количество запусков приложения, в течение которых

пользователь применял данное действие; в правой колонке — число запусков, прошедших со времени последнего его использования. По истечении этого числа запусков клиенты действия маскируются. Например, в меню они доступны не сразу, а после нажатия специального пункта с двумя стрелочками, обращенными вниз.

Естественно, чем чаще пользователь обращался к действию, тем дольше оно удержится на виду. Впрочем, если у вас другие взгляды на интерфейс, вы можете изменить значение `PriorityScedule`.

Таблица 8.4. *Условия скрытия элементов панелей действий*

Количество запусков приложения с обращением к действию	Количество запусков приложения после последнего обращения
0, 1	3
2	6
3	9
4, 5	12
6—8	17
9—13	23
14—24	29
Более 25	31

Для подсчета величин, указанных в этой таблице, введены такие свойства:

- у объекта `TActionBars` (дочерний объект `TActionManager`) есть свойство

  ```
  property SessionCount: Integer;
  ```

  которое представляет собой глобальный счетчик запусков приложения;
- у каждого объекта `TActionClientItem` есть два свойства:
  - `property LastSession: Integer;`

    в этом свойстве хранится номер последнего запуска приложения, в течение которого использовался данный элемент (нумерация совпадает с `SessionCount`);
  - `property UsageCount: Integer;`

    счетчик использования элемента.

Но для того, чтобы оперировать данными о количестве запусков, их надо где-то хранить. Организована система хранения следующим образом. У самого менеджера действий есть свойство

```
property FileName: TFileName;
```

которое указывает на файл, содержащий все настройки панелей, связанных с данным менеджером. Он имеет формат двоичной формы и считывается/записывается при запуске и выходе из приложения. Впрочем, можно это сделать и в любой момент при помощи методов `LoadFormFile` и `SaveToFile`.

Все эти величины меняются автоматически, и их описание приведено для понимания сути происходящего. Сбросить же счетчик статистики запусков можно следующим образом: на этапе разработки на странице **Options** редактора свойств менеджера действий есть кнопка **Reset Usage Count**. На этапе выполнения точно такая кнопка есть в диалоге `TCustomizeDlg`.

Помимо данных для подсчета запусков в этом файле хранится и вся прочая информация о настройках. Последний из не упоминавшихся нами компонентов — диалог настройки `TCustomizeDlg`. Он представляет собой точную копию редактора свойств `TActionManager`, но позволяет делать все операции с действиями в режиме выполнения. Вызвать его просто — вызовом метода `Show`. А можно поступить еще проще — есть стандартное действие `Customize` (категория **Tools**), которое и подразумевает вызов этого диалога.

Ручное редактирование коллекций панелей и действий

Перетаскивание имеет много достоинств, однако оно не всегда удобно. Поэтому было бы странно, если бы не было предусмотрено другого способа. Хоть он напрямую и не рекомендован в документации, но в ряде случаев более эффективен.

Рассмотрим работу с дочерними объектами менеджера действий, которые упакованы один в другой, как матрешки.

Итак, щелкнем на свойстве `ActionManager` на форме и посмотрим на содержимое Инспектора объектов. Внутри него мы обнаружим сразу две "матрешки" — свойство `ActionBars` содержит коллекцию ссылок на дочерние панели, свойство `LinkedActionList` может содержать дополнительный список действий, отданных "в управление" данному менеджеру — например, для централизованной установки общих свойств.

Щелкнем на свойстве `ActionBars`. Появится редактор панелей (рис. 8.7), а в Инспекторе объектов обратим внимание на следующее свойство объекта `ActionBars`:

```
property Customizable: Boolean;
```

Это свойство указывает, может ли коллекция редактироваться во время выполнения.

В коллекции содержатся не сами панели, а их "заместители" — объекты типа `TActionBarItem`, которые на них указывают. Надпись на рисунке **"1-ActionBar -> ActionToolBar2"** показывает, что первый элемент коллекции

связан с панелью ActionToolBar2. Вы можете добавлять и удалять элементы этой коллекции, по мере необходимости связывая их через свойство ActionBar с уже существующей панелью.

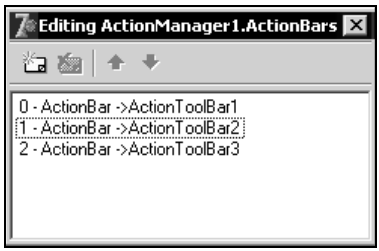


Рис. 8.7. Редактор коллекции панелей компонента TActionManager

Через Инспектор объектов вы можете изменять внешний вид объектов типа TActionBarItem и соответствующих им панелей.

Свойство

```
property Color: TColor;
```

отвечает за фоновый цвет панели. Если вам изменения цвета недостаточно, в качестве фона выберите картинку

```
property Background: TPicture;
```

которая будет расположена на панели в соответствии со значением свойства

```
property BackgroundLayout: TBackgroundLayout;
TBackgroundLayout = (blNormal, blStretch, blTile, blLeftBanner,
blRightBanner);
```

Помимо внешнего вида можно разрешить/запретить перетаскивание панелей и их дочерних элементов. Обратимся к свойству

```
property ChangesAllowed: TChangesAllowedSet;
TChangesAllowed = (caModify, caMove, caDelete);
TChangesAllowedSet = set of TChangesAllowed;
```

Множество из трех возможных значений позволяет запретить те или иные нежелательные изменения для дочерних элементов панели. Если в него не включен режим caDelete, то элемент нельзя убирать (перетаскивать) с панели. Если нет режима caMove — нельзя передвигать внутри панели. Наконец, отсутствие режима caModify означает запрет на изменение визуальных свойств (заголовка и т. п.).

Внутри коллекции TActionBarItem спрятаны еще две "матрешки" — свойства Items и ContextItems. Оба свойства представляют из себя коллекции объек-

тов, указывающих на действия (класс коллекции `TActionClients`, класс элемента коллекции `TActionClientItem`). Первое свойство указывает непосредственно на дочерние действия, второе — на действия, которые будут показаны в качестве всплывающего меню при нажатии правой кнопки мыши.

У коллекции `TActionClients` есть заслуживающие особого упоминания свойства.

Свойство

```
property CaptionOptions: TCaptionOptions;
TCaptionOptions = (coNone, coSelective, coAll);
```

задает показ/отсутствие заголовков дочерних действий. В случае установки в `coNone` они не показываются, `coAll` — показываются все, `coSelective` — показываются в соответствии со значением `ShowCaption` дочернего объекта `TActionClientItem`. Это свойство можно также установить на первой странице редактора менеджера действий в одноименном выпадающем списке.

Свойство

```
property SmallIcons: Boolean;
```

указывает размер значков, соответствующих действиям. По умолчанию установлено в значение `True` (маленькие значки). Визуально оно доступно через тот же редактор — третья страница, флажок **Large Icons**.

Свойство

```
property HideUnused: Boolean;
```

разрешает скрытие редко используемых действий, описанное в предыдущем разделе. Если вы не хотите пользоваться механизмом скрытия, на третьей странице редактора менеджера действий и диалога `TCustomizeDlg` есть флажок **Menu show recent items first**. Сбросьте его, и свойства `HideUnused` у клиентов действий установятся в значение `False`.

И, наконец, коллекцию можно сделать нередактируемой. Для этого у нее есть свойство `Customizable`.

Ну вот, мы уже добрались до самой маленькой матрешки — `TActionClientItem`. Этот объект связывается напрямую с одним действием через свойство `Action`. Правда в него можно спрятать еще меньшую матрешку — у него также есть свойства `Items` и `ContextItems`. Эти свойства используются при организации многоуровневых меню и меню, выпадающих из кнопок (точнее, псевдокнопок — напомним, объекты `TActionClientItem` на панелях не являются ни кнопками, ни компонентами вообще).

Резюме

Хорошо знакомые со времен Delphi 1 составляющие интерфейса — меню (`TMainMenu`, `TPopupMenu`), кнопки (`TButton`, `TSpeedButton`), панели `TPanel` — постепенно уходят, уступая место компонентам с расширенной функциональностью. Центральным местом, где обрабатывается весь ввод пользователя, становится хранилище действий — `TActionList` или `TActionManager`.

В этой главе мы подробно рассмотрели оба компонента; читателю решать, на базе чего строить свой интерфейс.

С одной стороны, материал этой главы не оказывает прямого влияния на последующие. С другой, пользовательский интерфейс — та "печка", от которой "пляшут" при создании больших приложений. Поэтому имейте в виду при чтении всех прочих данную главу.

ГЛАВА 9

Файлы и устройства ввода/вывода

Большинство приложений создаются для того, чтобы обрабатывать данные — это прописная истина. С развитием программных технологий мы имеем возможность получать и посылать все более крупные и сложные массивы данных; однако до сих пор 90% из них хранятся в файлах.

Для использования файлов в приложении разработчику приходится решать множество задач. Главные из них — поиск необходимого файла и выполнение с ним операций ввода/вывода.

Основные принципы и структура файловой системы мало изменились со времен MS-DOS. Файловые системы (FAT32, NTFS), появившаяся в Windows 2000 служба Active Directory не изменяют главного — понятия файла и способов обращения к нему.

Среда Delphi дает вам возможность выбрать один из четырех вариантов работы:

- использование традиционного набора функций работы с файлами, унаследованного от Turbo Pascal;
- использование функций ввода/вывода из Windows API;
- использование потоков (`TStream` и его потомки);
- использование отображаемых файлов.

В этой главе мы изучим все основные способы работы с файлами в приложениях Delphi на конкретных примерах создания программного кода.

Использование файловых переменных. Типы файлов

Зачастую современный программный код Delphi для чтения данных из файла удивительно похож на аналогичный, написанный, к примеру, в Turbo

Pascal 4.0. Это возможно потому, что программисты Borland сохранили неизменным "старый добрый" набор файловых функций, работающих через файловые переменные.

При организации операций файлового ввода/вывода в приложении большое значение имеет, какого рода информация содержится в файле. Чаще всего это строки, но встречаются двоичные данные или структурированная информация, например массивы или записи.

Естественно, что сведения о типе хранящихся в файле данных важно изначально задать. Для этого используются специальные файловые переменные, определяющие тип файла. Они делятся на нетипизированные и типизированные.

Перед началом работы с любым файлом необходимо описать файловую переменную, соответствующую типу данных этого файла. В дальнейшем эта переменная используется при обращении к файлу.

В Delphi имеется возможность создавать *нетипизированные файлы*. Для их обозначения используется ключевое слово `file`:

```
var UntypedFile: file;
```

Такие файловые переменные используются для организации быстрого и эффективного ввода/вывода безотносительно к типу данных. При этом подразумевается, что данные читаются или записываются в виде двоичного массива. Для этого применяются специальные процедуры блочного чтения и записи *(см. ниже)*.

Типизированные файлы обеспечивают ввод/вывод с учетом конкретного типа данных. Для их объявления используется ключевое слово `file of`, к которому добавляется конкретный тип данных. Например, для работы с файлом, содержащим набор байтов, файловая переменная объявляется так:

```
var ByteFile: file of byte;
```

При этом можно использовать любые типы фиксированного размера, за исключением указателей. Разрешается применять структурные типы, если их составные части удовлетворяют названному выше ограничению. Например, можно создать файловую переменную для записи:

```
type Country = record
      Name:       String;
      Capital:    String;
      Population: LongInt;
      Square:     LongInt;
     end;

var CountryFile: file of Country;
```

Для работы с текстовыми файлами используется специальная файловая переменная `TextFile` или `Text`:

```
var F: TextFile;
```

Операции ввода/вывода

Теперь рассмотрим две самые распространенные операции, выполняемые при работе с файлами. Это чтение и запись. Для их осуществления применяются специальные функции файлового ввода/вывода.

Итак, для выполнения операции чтения или записи необходимо произвести следующие действия:

1. Объявить файловую переменную необходимого типа.
2. При помощи функции `AssignFile` связать эту переменную с требуемым файлом.
3. Открыть файл при помощи функций `Append`, `Reset`, `Rewrite`.
4. Выполнить операции чтения или записи. При этом, в зависимости от сложности задачи и структуры данных, может использоваться целый ряд вспомогательных функций.
5. Закрыть файл при помощи функции `CloseFile`.

Внимание!

По сравнению с Turbo Pascal изменились названия только двух функций: `Assign` стала `AssignFile`, а `Close` превратилась в `CloseFile`.

В качестве примера рассмотрим небольшой фрагмент исходного кода.

```
...
var F: TextFile;
    S: string;
begin
 if OpenDlg.Execute
  then AssignFile(F, OpenDlg.FileName)
  else Exit;
 Reset(F);
 while Not EOF(F) do
 begin
  Readln(F, S);
  Memo.Lines.Add(S);
 end;
 CloseFile(F);
end;
...
```

Если в диалоге открытия файла `OpenDlg` был выбран файл, то его имя связывается с файловой переменной `F` при помощи процедуры `AssignFile`. В качестве имени файла рекомендуется всегда передавать полное имя файла (включая его маршрут). Как раз в таком виде возвращают результат выбора файла диалоги работы с файлами `TOpenDialog`, `TOpenPictureDialog`. Затем при помощи процедуры `Reset` этот файл открывается для чтения и записи.

В цикле выполняется чтение из файла текстовых строк и запись их в компонент `TMemo`. Процедура `Readln` осуществляет чтение текущей строки файла и переходит на следующую строку. Цикл выполняется, пока функция `EOF` не сообщит о достижении конца файла.

После завершения чтения файл закрывается.

Такой же исходный код можно использовать и для записи данных в файл. Необходимо только заменить процедуру чтения на процедуру записи.

Теперь остановимся подробнее на назначении используемых для файлового ввода/вывода функций.

Открытие файла может осуществляться тремя процедурами — в зависимости от типа его дальнейшего использования.

Процедура

```
procedure Reset(var F: File [; RecSize: Word ]);
```

открывает существующий файл для чтения и записи, текущая позиция устанавливается на первой строке файла.

Процедура

```
procedure Append(var F: Text);
```

открывает файл для записи информации после его последней строки, текущая позиция устанавливается на конец файла.

Процедура

```
procedure Rewrite(var F: File [; RecSize: Word ]);
```

создает новый файл и открывает его, текущая позиция устанавливается в начало файла. Если файл с таким именем уже существует, то он перезаписывается.

Переменная `RecSize` используется только при работе с нетипизированными файлами и определяет размер одной записи для операции передачи данных. Если этот параметр опущен, то по умолчанию `RecSize` равно 128 байт.

Чтение данных из типизированных и текстовых файлов выполняют процедуры `Read` и `Readln`.

Процедура `Read` имеет различное объявление для текстовых и других типизированных файлов:

- `procedure Read([var F: Text;] V1 [, V2,...,Vn]);`

  для текстовых файлов;

- `procedure Read(F, V1 [, V2,...,Vn]);`

  для других типизированных файлов.

При одном вызове процедуры можно читать данные в произвольное число переменных. Естественно, что тип переменных должен совпадать с типом файла. При чтении в очередную переменную читается ровно столько байтов из файла, сколько занимает тип данных. В следующую переменную читается столько же байтов, расположенных следом. После выполнения процедуры текущая позиция устанавливается на первом непрочитанном байте. Аналогично работают несколько процедур `Read` для одной переменной, выполненных подряд.

Процедура

```
procedure Readln([ var F: Text; ] V1 [, V2,...,Vn ]);
```

считывает одну строку текстового файла и устанавливает текущую позицию на следующей строке. Если использовать процедуру без переменных `V1..Vn`, то она просто передвигает текущую позицию на очередную строку файла.

Процедуры для записи в файл `Write` и `Writeln` описаны аналогично:

```
procedure Write([var F: Text; ] P1 [, P2,..., Pn]);
procedure Writeln([ var F: Text; ] P1 [, P2,...,Pn ]);
```

Параметры `P1`, `P2`, ..., `Pn` могут быть одним из целых или вещественных типов, одним из строковых типов или логическим типом. Но у них есть возможность дополнительного форматирования при выводе. Каждый параметр записи может иметь форму:

```
Pn [: MinWidth [: DecPlaces ] ]
```

`Pn` — выводимая переменная или выражение;

`MinWidth` — минимальная ширина поля в символах, которая должна быть больше 0;

`DecPlaces` — содержит количество десятичных символов после запятой при отображении вещественных чисел с фиксированной точкой.

Обратите внимание, что для текстовых файлов в функциях `Read` и `Write` файловая переменная `F` может быть опущена. В этом случае чтение и запись осуществляются в стандартные файлы ввода/вывода. Когда программа компилируется как консольное приложение (флаг `{$APPTYPE CONSOLE}`), Delphi автоматически связывает входной и выходной файлы с окном консоли.

Для контроля за текущей позицией в файле применяются две основные функции. Функция `EOF(F)` возвращает значение `True`, если достигнут конец файла. Функция `EOLN(F)` аналогично сигнализирует о достижении конца строки. Естественно, в качестве параметра в функции необходимо передавать файловую переменную.

Процедура

```
procedure Seek(var F; N: Longint);
```

обеспечивает смещение текущей позиции на `N` элементов. Размер одного элемента в байтах зависит от типа данных файла (от типизированной переменной).

Рассмотрим теперь режим блочного ввода/вывода данных между файлом и областью адресного пространства (буфером). Этот режим отличается значительной скоростью передачи данных, причем скорость пропорциональна размеру одного передаваемого блока — чем больше блок, тем больше скорость.

Для реализации этого режима необходимо использовать только нетипизированные файловые переменные. Размер блока определяется в процедуре открытия файла (`Reset`, `Rewrite`). Непосредственно для выполнения операций используются процедуры `BlockRead` и `BlockWrite`.

Процедура

```
procedure BlockRead(var F: File; var Buf; Count: Integer
[; var AmtTransferred: Integer]);
```

выполняет запись блока из файла в буфер. Параметр `F` ссылается на нетипизированную файловую переменную, связанную с нужным файлом.

Параметр `Buf` определяет любую переменную (число, строку, массив, структуру), в которую читаются байты из файла. Параметр `Count` содержит число считываемых блоков. Наконец, необязательный параметр `AmtTransferred` возвращает число реально считанных блоков.

При использовании блочного чтения или записи размер блока необходимо выбирать таким образом, чтобы он был кратен размеру одного значения того типа, который хранится в файле. Например, если в файле хранятся значения типа `Double` (8 байт), то размер блока может быть равен 8, 16, 24, 32 и т. д. Фрагмент исходного кода блочного чтения при этом выглядит следующим образом:

```
...
var F:           File;
    DoubleArray: array [0..255] of Double;
    Transfered:  Integer;
```

```
begin
 if OpenDlg.Execute then AssignFile(F, OpenDlg.FileName) else Exit;
 Reset(F, 64);
 BlockRead(F, DoubleArray, 32, Transfered);
 CloseFile(F);
 ShowMessage('Считано '+IntToStr(Transfered)+' блоков');
end;
...
```

Как видно из примера, размер блока установлен в процедуре `Reset` и кратен размеру элемента массива `DoubleArray`, в который считываются данные. В переменной `Transfered` возвращается число считанных блоков. Если размер файла меньше заданного в процедуре `BlockRead` числа блоков, ошибка не возникает, а в переменной `Transfered` передается число реально считанных блоков.

Процедура

```
procedure BlockWrite(var f: File; var Buf; Count: Integer
[; var AmtTransferred: Integer]);
```

используется аналогично.

Оставшиеся функции ввода/вывода, работающие с файловыми переменными, подробно описаны в табл. 9.1.

Таблица 9.1. *Процедуры и функции для работы с файлом*

Объявление	Описание
`function ChangeFileExt(const FileName, Extension: string): string;`	Функция позволяет изменить расширение файла. При этом сам файл не переименовывается
`procedure ChDir(S: string);`	Процедура изменяет текущий каталог на другой, путь к которому описан в строке `S`
`procedure CloseFile(var F);`	Вызов процедуры разрывает связь между файловой переменной и файлом на диске. Имя этой процедуры изменено из-за конфликта имен в Delphi (в Borland Pascal используется процедура `Close`)
`function DeleteFile(const FileName: string): Boolean;`	Функция производит удаление файла `FileName` с диска и возвращает значение `False`, если файл удалить не удалось или файл не существует

Таблица 9.1 *(продолжение)*

Объявление	Описание
`function ExtractFileExt(const FileName: string): string;`	Функция возвращает расширение файла
`function ExtractFileName(const FileName: string): string;`	Извлекает имя и расширение файла, содержащегося в параметре `FileName`
`function ExtractFilePath(const FileName: string): string;`	Функция возвращает полный путь к файлу
`procedure Erase(var F);`	Удаляет файл, связанный с файловой переменной `F`
`function FileSearch(const Name, DirList: string): string;`	Данная процедура производит поиск в каталогах `DirList` файла `Name`. Если в процессе выполнения `FileSearch` обнаруживается искомое имя файла, то функция возвращает в строке типа `String` полный путь к найденному файлу. Если файл не найден, то возвращается пустая строка
`function FileSetAttr(const FileName: string; Attr: Integer): Integer;`	Присваивает файлу с именем `FileName` атрибуты `Attr`. Функция возвращает 0, если присвоение атрибутов прошло успешно. В противном случае возвращается код ошибки
`function FilePos(var F): Longint;`	Возвращает текущую позицию файла. Функция используется для нетекстовых файлов. Перед вызовом `FilePos` файл должен быть открыт
`function FileSize(var F): Integer;`	`FileSize` возвращает размер файла в байтах или количество записей в файле, содержащем записи. Перед вызовом данной функции файл должен быть открыт. Для текстовых файлов функция `FileSize` не используется
`procedure Flush(var F: Text);`	Процедура очищает буфер текстового файла, открытого для записи. `F` — файловая переменная. Когда текстовый файл открыт для записи с использованием функции `Rewrite` или `Append`, `Flush` очищает выходной буфер, связанный с файлом. После выполнения данной процедуры все символы, которые направлены для записи в файл, будут гарантированно записаны в нем

Таблица 9.1 *(продолжение)*

Объявление	Описание
`procedure GetDir(D: Byte; var S: string);`	Возвращает число, соответствующее диску, на котором содержится текущий каталог `S`. `D` может принимать одно из следующих значений: • 0 — по умолчанию (текущий); • 1 — A; • 2 — B; • 3 — C и т. д. Процедура не генерирует код ошибки. Если имя диска в `D` оказывается ошибочным, то в строке `S` возвращается значение X:\, как если бы текущая папка была на этом ошибочно указанном диске
`function IOResult: Integer;`	Функция возвращает статус последней произведенной операции ввода/вывода, если контроль ошибок выключен `{$I-}`
`procedure MkDir(S: string);`	Процедура создает новый каталог, который описывается в строке `S`
`procedure Rename(var F; NewName: string);`	Процедура изменяет имя файла, связанного с файловой переменной `F`. Переменная `NewName` является строкой типа `string` или `PChar` (если включена поддержка расширенного синтаксиса)
`procedure RmDir(S: string);`	Процедура удаляет пустой каталог, путь к которому задается в строке `S`. Если указанный каталог не существует или он не пустой, то возникает сообщение об ошибке ввода/вывода
`procedure Seek(var F; N: Longint);`	Перемещает текущую позицию курсора на `N` позиций. Данная процедура используется только для открытых типизированных или нетипизированных файлов для проведения чтения/записи с нужной позиции файла. Началу файла соответствует нулевой номер позиции. Для добавления новой информации в конец существующего файла необходимо установить указатель на символ, следующий за последним. Для этого можно использовать выражение `Seek(F, FileSize(F))`

Таблица 9.1 *(окончание)*

Объявление	Описание
`function SeekEof[(var F: Text)]: Boolean;`	Возвращает значение `True`, если указатель текущей позиции находится на символе конца файла. `SeekEof` может быть использован только с открытым текстовым файлом
`function SeekEoln[(var F: Text)]: Boolean;`	Возвращает значение `True`, если указатель текущей позиции находится на символе конца строки. `SeekEoln` может быть использован только с открытым текстовым файлом
`procedure SetTextBuf(var F: Text; var Buf [; Size: Integer]);`	Связывает с текстовым файлом буфер ввода/вывода. `F` — файловая переменная текстового типа. Каждая файловая переменная текстового типа имеет внутренний буфер емкостью 128 байт, в котором накапливаются данные при чтении и записи. Такой буфер пригоден для большинства операций. Однако при выполнении программ с интенсивным вводом/выводом буфер может переполниться, что приведет к записи операций ввода/вывода на диск и, как следствие, к существенному замедлению работы приложения. `SetTextBuf` позволяет помещать в текстовый файл `F` информацию об операциях ввода/вывода вместо ее размещения в буфере. `Size` указывает размер буфера в байтах. Если этот параметр опускается, то полагается размер, равный `SizeOf(Buf)`. Новый буфер действует до тех пор, пока `F` не будет связана с новым файлом процедурой `AssignFile`
`procedure Truncate(var F);`	Удаляет все позиции, следующие после текущей позиции в файле. А текущая позиция становится концом файла. С переменной `F` может быть связан файл любого типа за исключением текстового

Ввод/вывод с использованием функций Windows API

Для тех, кто переходит на Delphi не с прежних версий Turbo Pascal, а с C, других языков или начинает освоение с "нуля", более привычными будут стандартные функции работы с файлами Windows. Тем более, что возмож-

ности ввода/вывода в них расширены. Каждый файл в этом наборе функций описывается не переменной, а дескриптором (`Handle`) — 32-разрядной величиной, которая идентифицирует файл в операционной системе.

В Win32 файл открывается при помощи функции, имеющей обманчивое название:

```
function CreateFile(lpFileName: PChar; dwDesiredAccess,
dwShareMode: DWORD;   lpSecurityAttributes: PSecurityAttributes;
dwCreationDistribution, dwFlagsAndAttributes: DWORD;
hTemplateFile: THandle): THandle;
```

Хоть ее название и начинается с `Create`, но она позволяет не только создавать, но и открывать уже существующие файлы.

Такое огромное количество параметров оправдано, т. к. `CreateFile` используется для открытия файлов на диске, устройств, каналов, портов и вообще любых источников ввода/вывода. Назначение параметров описано в табл. 9.2.

Таблица 9.2. *Параметры функции* `CreateFile`

Параметр	Описание
`lpFileName: pChar`	Имя открываемого объекта. Может представлять собой традиционную строку с путем и именем файла, UNC (для открытия объектов в сети, имя порта, драйвера или устройства)
`dwDesiredAccess,: DWORD`	Способ доступа к объекту. Может быть равен: • `GENERIC_READ` — для чтения; • `GENERIC_WRITE` — для записи. Их комбинация позволяет открыть файл для чтения и записи. Параметр 0 применяется, если нужно получить атрибуты файла без его фактического открытия
`dwShareMode: DWORD`	Режим совместного использования файла: • 0 — совместный доступ запрещен; • `FILE_SHARE_READ` — для чтения; • `FILE_SHARE_WRITE` — для записи. Их комбинация — для полного совместного доступа
`lpSecurityAttributes: PSecurityAttributes`	Атрибуты защиты файла. В Windows 95/98 не используются (должны быть равны `nil`). В Windows NT/2000 этот параметр, равный `nil`, дает объекту атрибуты по умолчанию

Таблица 9.2 *(окончание)*

Параметр	Описание
dwCreationDistribution: DWORD;	Способ открытия файла: • CREATE_NEW — создается новый файл, если таковой уже существует, функция возвращает ошибку ERROR_ALREADY_EXISTS; • CREATE_ALWAYS — создается новый файл, если таковой уже существует, он перезаписывается; • OPEN_EXISTING — открывает существующий файл, если таковой не найден, функция возвращает ошибку; • OPEN_ALWAYS — открывает существующий файл, если таковой не найден, он создается
dwFlagsAndAttributes: DWORD;	Набор атрибутов (скрытый, системный, сжатый) и флагов для открытия объекта. Подробное описание см. в документации по Win32
hTemplateFile: THandle	Файл-шаблон, атрибуты которого используются для открытия. В Windows 95/98 не используется и должен быть равен 0

Функция CreateFile возвращает дескриптор открытого объекта ввода/вывода. Если открытие невозможно из-за ошибок, возвращается код INVALID_HANDLE_VALUE, а расширенный код ошибки можно узнать, вызвав функцию GetLastError.

Закрывается файл в Win32 функцией CloseHandle (не CloseFile, а CloseHandle! Правда, "легко" запомнить? Что поделать, так их назвали разработчики Win32).

Приведем из большого разнообразия несколько приемов использования функции CreateFile. Часто программисты хотят иметь возможность организовать посекторный доступ к физическим устройствам хранения — например к дискете. Сделать это не так уж сложно, но при этом методы для Windows 98 и Windows 2000 различаются. В Windows 2000 придется открывать устройство ('\\.\A:'), а в Windows 98 — специальный драйвер доступа (обозначается '\\.\vwin32'). И то и другое делается функцией CreateFile.

Листинг 9.1. Чтение сектора с дискеты при помощи функции CreateFile

```
type
pDIOCRegs = ^TDIOCRegs;
```

```
TDIOCRegs = packed record
rEBX,rEDX,rECX,rEAX,rEDI, rESI, rFlags : DWORD;
end;

const VWIN32_DIOC_DOS_IOCTL = 1;
  VWIN32_DIOC_DOS_INT13 = 4;   // Прерывание 13

  SectorSize = 512;

function ReadSector(Head, Track, Sector: Integer; buffer : pointer;
Floppy: char):Boolean;
var hDevice : THandle;
  Regs : TDIOCRegs;
  DevName : string;
  nb : Integer;
begin
 if WIN32PLATFORM <> VER_PLATFORM_WIN32_NT then
 begin {win95/98}
 hDevice := CreateFile('\\.\vwin32',  GENERIC_READ, 0, nil, 0,
FILE_FLAG_DELETE_ON_CLOSE, 0);
 if (hDevice = INVALID_HANDLE_VALUE) then
begin
 Result := FALSE;
 Exit;
end;
 regs.rEDX := Head * $100 + Ord(Floppy in ['b', 'B']);
 regs.rEAX := $201; // код операции read sector
 regs.rEBX := DWORD(buffer); // buffer
 regs.rECX := Track * $100 + Sector;
 regs.rFlags := $0;

 Result := DeviceIoControl(hDevice,VWIN32_DIOC_DOS_INT13,
   @regs, sizeof(regs),  @regs, sizeof(regs), nb, nil)
  and ((regs.rFlags and  $1)=0);
 CloseHandle(hDevice);
 end {win95/98}
else
begin // Windows NT/2000
 DevName :='\\.\A:';
 if Floppy in ['b', 'B'] then DevName[5] := Floppy;
 hDevice := CreateFile(pChar(Devname), GENERIC_READ,     FILE_SHARE_READ
or FILE_SHARE_WRITE, nil, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
  if (hDevice = INVALID_HANDLE_VALUE) then
 begin
  Result := FALSE;
```

```
 Exit;
end;
 SetFilePointer(hDevice, (Sector-1)*SectorSize, nil, FILE_BEGIN);
                                                        // нумерация с 1
 Result := ReadFile(hDevice, buffer^, SectorSize, nb, nil) and
(nb=SectorSize);
 CloseHandle(hDevice);
 end; // Windows NT/2000
end;
```

Для чтения и записи данных в Win32 используются функции:

```
function ReadFile(hFile: THandle; var Buffer; nNumberOfBytesToRead:
DWORD; var lpNumberOfBytesRead: DWORD; lpOverlapped: POverlapped): BOOL;
function WriteFile(hFile: THandle; const Buffer; nNumberOfBytesToWrite:
DWORD; var lpNumberOfBytesWritten: DWORD; lpOverlapped: POverlapped):
BOOL;
```

Здесь все сходно с `BlockRead` и `BlockWrite`: `hFile` — это дескриптор файла, `Buffer` — адрес, по которому будут читаться (писаться) данные; третий параметр означает требуемое число читаемых (записываемых) байтов, а четвертый — фактически прочитанное (записанное). Последний параметр — `lpOverlapped` — обсудим чуть позже.

Функция `CreateFile` используется и для доступа к портам ввода/вывода. Часто программисты сталкиваются с задачей: как организовать обмен данными с различными нестандартными устройствами, подключенными к параллельному или последовательному порту? В Turbo Pascal для DOS был очень хороший псевдомассив `Ports`: пишешь `Port[X] := y;` и не знаешь проблем. В Win32 прямой доступ к портам запрещен и приходится открывать их как файлы:

```
...
hCom := CreateFile('COM2', GENERIC_READ or GENERIC_WRITE,
                  0, NIL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);
if hCom = INVALID_HANDLE_VALUE then
  begin
  raise EAbort.CreateFmt('Ошибка открытия порта: %d',[GetLastError]);
  end;
```

Самое большое отличие от предыдущего примера — в скромном флаге `FILE_FLAG_OVERLAPPED`. О роли этих изменений — в следующем разделе.

Отложенный (асинхронный) ввод/вывод

Эта принципиально новая возможность введена впервые в Win32 с появлением реальной многозадачности. Вызывая функции чтения и записи данных, вы на самом деле передаете исходные данные одному из потоков

(threads) операционной системы, который и осуществляет фактические обязанности по работе с устройством. Время доступа всех периферийных устройств гораздо больше доступа к ОЗУ, и ваша программа, вызвавшая `Read` или `Write`, будет дожидаться окончания операции ввода/вывода. Замедление работы программы налицо.

Выход был найден в использовании *отложенного* (overlapped) ввода/вывода. До начала отложенного ввода/вывода инициализируется дескриптор объекта типа события (функция `CreateEvent`) и структура типа `TOverlapped`. Вы вызываете функцию `ReadFile` или `WriteFile`, в которой последним параметром указываете на `TOverlapped`. Эта структура содержит дескриптор события Windows (event).

ОС начинает операцию (ее выполняет отдельный программный поток, скрытый от программиста) и немедленно возвращает управление; вы можете не тратить время на ожидание. Признак того, что операция началась и продолжается — получение кода возврата `ERROR_IO_PENDING`. Пусть вас не пугает слово "error" в названии — это совершенно нормально. Если операция продолжается долго (а чтение и запись файлов на дискете, да и на диске, именованных каналов можно отнести к "длинным" операциям), то программа может спокойно выполнять последующие операторы. Событие будет "взведено" ОС тогда, когда ввод/вывод закончится.

Когда, по мнению программиста, ввод/вывод должен быть завершен, можно проверить это, использовав функцию `WaitForSingleObject`.

```
function WaitForSingleObject(hHandle: THandle; dwMilliseconds: DWORD):
DWORD;
```

Объект ожидания (параметр `hHandle`) в этом случае — тот самый, который создан нами, указан в структуре `TOverlapped` и передан в качестве параметра в функцию `ReadFile` или `WriteFile`. Можно указать любое время ожидания, в том числе бесконечное (параметр `TimeOut` при этом равен константе `INFINITE`). Признаком нормального завершения служит получение кода возврата `WAIT_OBJECT_0`.

Листинг 9.2. Пример отложенной операции чтения

```
function TMyClass.Read(var Buffer; Count: Longint): Longint;
var succ : boolean;nb : Cardinal;LastError : Longint;
Overlap: TOverlapped;
begin
FillChar(Overlap,SizeOf(Overlap),0);
Overlap.hEvent := CreateEvent(nil, True, False, nil);
Result := MaxInt;
succ := ReadFile(FHandle, Buffer, Count, nb, @OverlapRd);
```

```
//
// Здесь можно вставить любые операторы, которые
// могут быть выполнены до окончания ввода/вывода
//
 if not succ then
    begin
     LastError := GetLastError;
     if LastError = ERROR_IO_PENDING
      then
       begin
        if WaitForSingleObject(OverlapRd.hEvent, INFINITE)=WAIT_OBJECT_0
        then
          GetOverlappedResult(FHandle, OverlapRd, nb, TRUE);
       end
       else
        raise EAbort.Create(Format('Read failed, error %d',[LastError]));
     end;
 Result := nb;
CloseHandle(hEvent);
end;
```

Если вы задали конечный интервал в миллисекундах, а операция еще не закончена, `WaitForSingleObject` вернет код завершения `WAIT_TIMEOUT`. Функция `GetOverlappedResult` возвращает в параметре `nb` число байтов, действительно прочитанных или записанных во время отложенной операции.

Контроль ошибок ввода/вывода

При работе с файлами разработчик обязательно должен предусмотреть обработку возможных ошибок. Практика показывает, что именно операции ввода/вывода вызывают бóльшую часть ошибок, возникающих в приложении из-за воздействия окружающей программной среды.

Контроль за ошибками ввода/вывода зависит от применяемых функций. При использовании доступа через Win32 API все функции возвращают код ошибки Windows, который и нужно проанализировать.

При возникновении ошибок ввода/вывода в функциях, использующих файловые переменные, генерируется исключительная ситуация класса `EInOutError`. Но так происходит только в том случае, если включен контроль ошибок ввода/вывода. Для этого используются соответствующие директивы компилятора:

- `{$I+}` — контроль включен (установлен по умолчанию);
- `{$I-}` — контроль отключен.

Класс `EInOutError` отличается тем, что у него есть поле `ErrorCode`. При возникновении этой исключительной ситуации вы можете получить его значение и принять решение. Основные коды имеют такие значения:

- 2 — файл не найден;
- 3 — неверное имя файла;
- 4 — слишком много открытых файлов;
- 5 — доступ запрещен;
- 100 — достигнут конец файла;
- 101 — диск переполнен;
- 106 — ошибка ввода.

При отключенном контроле в случае возникновения ошибки выполнение программы продолжается без остановки. Однако в этом случае устранение возможных последствий ошибки возлагается на разработчика. Для этого применяется функция

```
function IOResult: Integer;
```

которая возвращает значение 0 при отсутствии ошибок.

Атрибуты файла. Поиск файла

Еще одна часто выполняемая с файлом операция — поиск файлов в заданном каталоге. Для организации поиска и отбора файлов используются специальные процедуры, а также структура, в которой сохраняются результаты поиска.

Запись

```
type
TFileName = string;
TSearchRec = record
 Time: Integer;      {Время и дата создания}
 Size: Integer;      {Размер файла}
 Attr: Integer;      {Параметры файла}
 Name: TFileName;     {Полное имя файла}
 ExcludeAttr: Integer;   {Не используется}
 FindHandle: THandle;   {Дескриптор файла}
 FindData: TWin32FindData; {Не используется}
end;
```

обеспечивает хранение характеристик файла после удачного поиска. Дата и время создания файла хранятся в формате MS-DOS, поэтому для получения

этих параметров в принятом в Delphi формате `TDateTime` необходимо использовать следующую функцию:

```
function FileDateToDateTime(FileDate: Integer): TDateTime;
```

Обратное преобразование выполняет функция

```
function DateTimeToFileDate(DateTime: TDateTime): Integer;
```

Свойство `Attr` может содержать комбинацию следующих флагов-значений:

- `faReadOnly` — только для чтения;
- `faHidden` — скрытый;
- `faSysFile` — системный;
- `faVolumeID` — метка тома;
- `faDirectory` — каталог;
- `faArchive` — архивный;
- `faAnyFile` — любой.

Для определения параметров файла используется оператор `AND`:

```
if (SearchRec.Attr AND faReadOnly) > 0
  then ShowMessage('Файл только для чтения');
```

Непосредственно для поиска файлов используются функции `FindFirst` и `FindNext`.

Функция

```
function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec):
Integer;
```

находит первый файл, заданный полным маршрутом `Path` и параметрами `Attr` *(см. выше)*. Если заданный файл найден, функция возвращает 0, иначе — код ошибки Windows. Параметры найденного файла возвращаются в записи `F` типа `TSearchRec`.

Функция

```
function FindNext(var F: TSearchRec): Integer;
```

применяется для повторного поиска следующего файла, удовлетворяющего критерию поиска. При этом используются те параметры поиска, которые заданы последним вызовом функции `FindFirst`. В случае удачного поиска возвращается 0.

Для освобождения ресурсов, выделенных для выполнения поиска, применяется функция:

```
procedure FindClose(var F: TSearchRec);
```

В качестве примера организации поиска файлов рассмотрим фрагмент исходного кода, в котором маршрут поиска файлов задается в однострочном текстовом редакторе `DirEdit`, а список найденных файлов передается в компонент `TListBox`.

```
procedure TForm1.FindBtnClick(Sender: TObject);
begin
 ListBox.Items.Clear;
 FindFirst(DirEdit.Text, faArchive + faHidden, SearchRec);
 while FindNext(SearchRec) = 0 do
 ListBox.Items.Add(SearchRec.Name);
 FindClose(SearchRec);
end;
```

Потоки

Потоки — очень удачное средство для унификации ввода/вывода для различных носителей. Потоки представляют собой специальные объекты-наследники абстрактного класса `TStream`. Сам `TStream` "умеет" открываться, читать, писать, изменять текущее положение и закрываться. Поскольку для разных носителей эти вещи происходят по-разному, конкретные аспекты реализованы в его потомках. Наиболее часто используются потоки для работы с файлами на диске и памятью.

Многие классы VCL имеют унифицированные методы `LoadFromStream` и `SaveToStream`, которые обеспечивают обмен данными с потоками. От того, с каким физическим носителем работает поток, зависит место хранения данных.

Базовые классы *TStream* и *THandleStream*

В основе иерархии классов потоков лежит класс `TStream`. Он обеспечивает выполнение основных операций потока безотносительно к реальному носителю информации. Основными из них являются чтение и запись данных.

Класс `TStream` порожден непосредственно от класса `TObject`.

Потоки также играют важную роль в чтении/записи компонентов из файлов ресурсов (DFM). Большая группа методов обеспечивает взаимодействие компонента и потока, чтение свойств компонента из ресурса и запись значений свойств в ресурс.

Таблица 9.3. *Свойства и методы класса* `TStream`

Объявление	Описание
`property Position: Longint;`	Определяет текущую позицию в потоке
`property Size: Longint;`	Определяет размер потока в байтах

Таблица 9.3 *(окончание)*

Объявление	Описание
`function CopyFrom(Source: TStream; Count: Longint): Longint;`	Копирует из потока `Source` `Count` байты, начиная с текущей позиции. Возвращает число скопированных байтов
`function Read(var Buffer; Count: Longint): Longint; virtual; abstract;`	Абстрактный класс, перекрываемый в наследниках. Считывает из потока `Count` байты в буфер `Buffer`. Возвращает число скопированных байтов
`procedure ReadBuffer(var Buffer; Count: Longint);`	Считывает из потока `Count` байты в буфер `Buffer`. Возвращает число скопированных байтов
`function Seek(Offset: Longint; Origin: Word): Longint; virtual; abstract;`	Абстрактный класс, перекрываемый в наследниках. Смещает текущую позицию в реальном носителе данных на `Offset` байтов в зависимости от условия `Origin` *(см. ниже)*
`function Write(const Buffer; Count: Longint): Longint; virtual; abstract;`	Абстрактный класс, перекрываемый в наследниках. Записывает в поток `Count` байты из буфера `Buffer`. Возвращает число скопированных байтов
`procedure WriteBuffer(const Buffer; Count: Longint);`	Записывает в поток `Count` байты из буфера `Buffer`. Возвращает число скопированных байтов
`function ReadComponent(Instance: TComponent): TComponent;`	Передает данные из потока в компонент `Instance`, заполняя его свойства значениями
`function ReadComponentRes(Instance: TComponent): TComponent;`	Считывает заголовок ресурса компонента `Instance` и значения его свойств из потока.
`procedure ReadResHeader;`	Считывает заголовок ресурса компонента из потока
`procedure WriteComponent(Instance: TComponent);`	Передает в поток значения свойств компонента `Instance`
`procedure WriteComponentRes(const ResName: string; Instance: TComponent);`	Записывает в поток заголовок ресурса компонента `Instance` и значения его свойств

Итак, в основе операций считывания и записи данных в потоке лежат методы `Read` и `Write`. Именно они вызываются для реального выполнения операции внутри методов `ReadBuffer` и `WriteBuffer`, `ReadComponent` и

`WriteComponent`. Так как класс `TStream` является абстрактным, то методы `Read` и `Write` также являются абстрактными. В классах-наследниках они перекрываются, обеспечивая работу с конкретным физическим носителем данных.

Метод `Seek` используется для изменения текущей позиции в потоке. "Точка отсчета" позиции зависит от значения параметра `Origin`:

- `soFromBeginning` — смещение должно быть положительным и отсчитывается от начала потока;
- `soFromCurrent` — смещение относительно текущей позиции в потоке;
- `soFromEnd` — смещение должно быть отрицательным и отсчитывается от конца потока.

Группа методов обеспечивает чтение и запись из потока ресурса компонента. Они используются при создании компонента на основе данных о нем, сохраненных в формате файлов ресурсов. Для чтения ресурса используется метод `ReadComponentRes`, в котором последовательно вызываются:

- метод `ReadResHeader` — для считывания заголовка ресурса компонента из потока;
- метод `ReadComponent` — для считывания значений свойств компонента.

Для записи ресурса в поток применяется метод `WriteComponentRes`.

Класс `THandleStream` инкапсулирует поток, связанный с физическим носителем данных через дескриптор.

Для создания потока используется конструктор

```
constructor Create(AHandle: Integer);
```

в параметре которого передается дескриптор. Впоследствии доступ к дескриптору осуществляется через свойство:

```
property Handle: Integer;
```

Класс *TFileStream*

Класс `TFileStream` позволяет создать поток для работы с файлами. При этом поток работает с файлом без учета типа хранящихся в нем данных *(см. выше)*.

Полное имя файла задается в параметре `FileName` при создании потока:

```
constructor Create(const FileName: string; Mode: Word);
```

Параметр `Mode` определяет режим работы с файлом. Он составляется из флагов режима открытия:

- `fmCreate` — файл создается;
- `fmOpenRead` — файл открывается для чтения;

- `fmOpenWrite` — файл открывается для записи;
- `fmOpenReadWrite` — файл открывается для чтения и записи.

И флагов режима совместного использования:

- `fmShareExclusive` — файл недоступен для открытия другими приложениями;
- `fmShareDenyWrite` — другие приложения могут читать данные из файла;
- `fmShareDenyRead` — другие приложения могут писать данные в файл;
- `fmShareDenyNone` — другие приложения могут производить с файлом любые операции.

Для чтения и записи из потока используются методы `Read` и `Write`, унаследованные от класса `THandleStream`:

```
procedure TForm1.CopyBtnClick(Sender: TObject);
var Stream1, Stream2: TFileStream;
    IntBuf: array[0..9] of Integer;
begin
 if Not OpenDlg.Execute then Exit;
 try
  Stream1 := TFileStream.Create(OpenDlg.FileName, fmOpenRead);
  Stream1.ReadBuffer(IntBuf, SizeOf(IntBuf));
  try
   Stream2 := TFileStream.Create('TextFile.tmp', fmOpenWrite);
   Stream2.Seek(0, soFromEnd);
   Stream2.WriteBuffer(IntBuf, SizeOf(IntBuf));
  finally
   Stream2.Free;
  end;
 finally
  Stream1.Free;
 end;
end;
```

Обратите внимание, что в данном фрагменте кода функция `Seek` используется для записи данных в конец файлового потока.

При необходимости копирования одного файла в другой целиком используется метод `CopyFrom`, унаследованный от класса `TStream`:

```
procedure TForm1.CopyBtnClick(Sender: TObject);
var Stream1, Stream2: TFileStream;
begin
 if Not OpenDlg.Execute then Exit;
```

```
 try
  Stream1 := TFileStream.Create(OpenDlg.FileName, fmOpenRead);
  Stream2 := TFileStream.Create('Sample.tmp', fmOpenWrite);
  Stream2.Seek(0, soFromEnd);
  Stream2.CopyFrom(Stream1, Stream1.Size);
 finally
  Stream1.Free;
  Stream2.Free;
 end;
end;
```

Обратите внимание, что в данном случае для определения размера передаваемого потока необходимо использовать свойство `Stream.Size`, которое дает реальный объем данных, содержащихся в потоке. Функция `SizeOf(Stream)` в этом случае даст размер объекта потока, и не более того.

Класс *TMemoryStream*

Класс `TMemoryStream` обеспечивает сохранение данных в адресном пространстве. При этом методы доступа к этим данным остаются теми же, что и при работе с файловыми потоками. Это позволяет использовать адресное пространство для хранения промежуточных результатов работы приложения, а также при помощи стандартных методов осуществлять обмен данными между памятью и другими физическими носителями.

Свойство

```
property Memory: Pointer;
```

определяет область памяти, отведенную для хранения данных потока.

Изменение размера отведенной памяти осуществляется методом

```
procedure SetSize(NewSize: Longint); override;
```

Для очистки памяти потока используется метод

```
procedure Clear;
```

Чтение/запись данных в память выполняется привычными методами `Read` и `Write`.

Также запись данных в память может осуществляться методами:

- `procedure LoadFromFile(const FileName: string);` — из файла;
- `procedure LoadFromStream(Stream: TStream);` — из другого потока.

Дополнительно можно использовать методы записи данных в файл или поток:

```
procedure SaveToFile(const FileName: string);
procedure SaveToStream(Stream: TStream);
```

Класс *TStringStream*

Так как строковые константы и переменные широко применяются при разработке приложений, то для удобства работы с ними создан специальный класс `TStringStream`. Он обеспечивает хранение строки и доступ к ней во время выполнения приложения.

Он обладает стандартным для потоков набором свойств и методов, добавляя к ним еще несколько, упрощающих использование строк.

Свойство только для чтения

```
property DataString: string;
```

обеспечивает доступ к хранимой строке.

Методы

```
function Read(var Buffer; Count: Longint): Longint; override;
```

и

```
function Write(const Buffer; Count: Longint): Longint; override;
```

реализуют обычный для потоков способ чтения и записи строки для произвольной переменной `Buffer`.

Метод

```
function ReadString(Count: Longint): string;
```

обеспечивает чтение `Count` байтов строки потока, начиная с текущей позиции.

Метод

```
procedure WriteString(const AString: string);
```

дописывает к строке строку `AString`, начиная с текущей позиции.

При работе с файлами и потоками используются дополнительные классы исключительных ситуаций.

Класс `EFCreateError` возникает при ошибке создания файла, а `EFOpenError` — при открытии файла.

При чтении/записи данных в поток могут возникнуть исключительные ситуации `EReadError` и `EWriteError`.

Оповещение об изменениях в файловой системе

Многие программисты задавались вопросом: как получить сигнал от операционной системы о том, что в файловой системе произошли какие-то изменения? Такой вид оповещения позаимствован из ОС UNIX и теперь доступен программистам, работающим с Win32.

Для организации мониторинга файловой системы нужно использовать три функции — FindFirstChangeNotification, FindNextChangeNotification и FindCloseChangeNotification. Первая из них возвращает дескриптор объекта файлового оповещения, который можно передать в функцию ожидания. Объект активизируется тогда, когда в заданной папке произошли те или иные изменения (создание или уничтожение файла или папки, изменение прав доступа и т. д.). Вторая функция — готовит объект к реакции на следующее изменение. Наконец, с помощью третьей функции следует закрыть, ставший ненужным, объект.

Так может выглядеть код метода Execute потока, созданного для мониторинга:

```
var DirName : string;
...
procedure TSimpleThread.Execute;
var r: Cardinal;
  fn : THandle;
begin
 fn := FindFirstChangeNotification(pChar(DirName),True,
FILE_NOTIFY_CHANGE_FILE_NAME);
 repeat
 r := WaitForSingleObject(fn,2000);
 if r = WAIT_OBJECT_0 then
  Form1.UpdateList;
 if not FindNextChangeNotification(fn) then break;
 until Terminated;
 FindCloseChangeNotification(fn);
end;
```

На главной форме должны находиться компоненты, нужные для выбора обследуемой папки, а также компонент TListBox, в который будут записываться имена файлов:

```
procedure TForm1.Button1Click(Sender: TObject);
var dir : string;
begin
 if SelectDirectory(dir,[],0) then
  begin
   Edit1.Text := dir;
   DirName := dir;
  end;
end;
procedure TForm1.UpdateList;
var SearchRec: TSearchRec;
```

```
begin
  ListBox1.Clear;
  FindFirst(Edit1.Text+'\*.*', faAnyFile, SearchRec);
  repeat
   ListBox1.Items.Add(SearchRec.Name);
  until FindNext(SearchRec) <> 0;
  FindClose(SearchRec);
end;
```

Приложение готово. Чтобы оно стало полнофункциональным, предусмотрите в нем механизм перезапуска потока при изменении обследуемой папки.

Использование отображаемых файлов

Последний — самый нетрадиционный вид работы с файлами — это так называемые отображаемые файлы.

Вообще говоря, в 32-разрядной Windows под "памятью" подразумевается не только оперативная память (ОЗУ), но также и память, резервируемая операционной системой на жестком диске. Этот вид памяти называется *виртуальной памятью*. Код и данные отображаются на жесткий диск посредством *страничной системы* (paging system) подкачки. Страничная система использует для отображения страничный файл (win386.swp в Windows 95/98 и pagefile.sys в Windows NT). Необходимый фрагмент виртуальной памяти переносится из страничного файла в ОЗУ и, таким образом, становится доступным.

А что, если так же поступить и с любым другим файлом и сделать его частью адресного пространства? В Win32 это возможно. Для выделения фрагмента памяти должен быть создан специальный системный объект Win32, называемый *отображаемым файлом*. Этот объект "знает", как соотнести файл, находящийся на жестком диске, с памятью, адресуемой процессами.

Одно или более приложений могут открыть отображаемый файл и получить тем самым доступ к данным этого объекта. Таким образом, данные, помещенные в страничный файл приложением, использующим отображаемый файл, могут быть доступны другим приложениям, если они открыли и используют тот же самый отображаемый файл.

Создание и использование объектов файлового отображения осуществляется посредством функций Windows API. Этих функций три:

```
CreateFileMapping
MapViewOfFile
UnMapViewOfFile
```

Отображаемый файл создается операционной системой при вызове функции `CreateFileMapping`. Этот объект поддерживает соответствие между со-

держимым файла и адресным пространством процесса, использующего этот файл. Функция `CreateFileMapping` имеет шесть параметров:

```
function CreateFileMapping(hFile: THandle; lpFileMappingAttributes:
PSecurityAttributes; flProtect, dwMaximumSizeHigh, dwMaximumSizeLow:
DWORD; lpName: PChar): THandle;
```

Первый параметр имеет тип `THandle`. Он должен соответствовать дескриптору уже открытого при помощи функции `CreateFile` файла. Если значение параметра `hFile` равно `$FFFFFFFF`, то это приводит к связыванию объекта файлового отображения со страничным файлом операционной системы.

Второй параметр — указатель на запись типа `TSecurityAttributes`. При отсутствии требований к защите данных в Windows NT значение этого параметра всегда равно `nil`. Третий параметр имеет тип `DWORD`. Он определяет атрибут защиты. Если при помощи отображаемого файла вы планируете совместное использование данных, третьему параметру следует присвоить значение `PAGE_READWRITE`.

Четвертый и пятый параметры также имеют тип `DWORD`. Когда выполняется функция `CreateFileMapping`, значение типа `DWORD` четвертого параметра сдвигается влево на четыре байта и затем объединяется со значением пятого параметра посредством операции `and`. Проще говоря, значения объединяются в одно 64-разрядное число, равное объему памяти, выделяемой объекту файлового отображения из страничного файла операционной системы. Поскольку вы вряд ли попытаетесь осуществить выделение более чем 4 Гбайт данных, то значение четвертого параметра всегда должно быть равно нулю. Используемый затем пятый параметр должен показывать, сколько памяти в байтах необходимо зарезервировать в качестве совместной. Если вы хотите отобразить весь файл, четвертый и пятый параметры должны быть равны нулю.

Шестой параметр имеет тип `PChar` и представляет собой имя объекта файлового отображения.

Функция `CreateFileMapping` возвращает значение типа `THandle`. В случае успешного завершения возвращаемое функцией значение представляет собой дескриптор созданного объекта файлового отображения. В случае возникновения какой-либо ошибки возвращаемое значение будет равно 0.

Следующая задача — спроецировать данные файла в адресное пространство нашего процесса. Этой цели служит функция `MapViewOfFile`. Функция `MapViewOfFile` имеет пять параметров:

```
function MapViewOfFile(hFileMappingObject: THandle; dwDesiredAccess:
DWORD; dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap: DWORD):
Pointer;
```

Первый параметр имеет тип `THandle`. Его значением должен быть дескриптор созданного объекта файлового отображения — тот, который возвращает

функция CreateFileMapping. Второй параметр определяет режим доступа к файлу: FILE_MAP_WRITE, FILE_MAP_READ или FILE_MAP_ALL_ACCESS.

Третий и четвертый параметры также имеют тип DWORD. Это — смещение отображаемого участка относительно начала файла в байтах. В нашем случае эти параметры должны быть установлены в нуль, поскольку значение, которое мы даем пятому (последнему) параметру функции MapViewOfFile, также равно нулю.

Пятый (и последний) параметр функции MapViewOfFile, как и предыдущие параметры, имеет тип DWORD. Он используется для определения (в байтах) количества данных объекта файлового отображения, которые надо отобразить в процесс (сделать доступными для вас). Для достижения наших целей это значение должно быть установлено в нуль, что означает автоматическое отображение в процессе всех данных, выделенных перед этим функцией CreateFileMapping.

Значение, возвращаемое функцией MapViewOfFile, имеет тип "указатель". Если функция отработала успешно, то она вернет начальный адрес данных объекта файлового отображения.

Следующий фрагмент кода демонстрирует вызов функции MapViewOfFile:

```
var
  hMappedFile: THandle;
  pSharedBuf:  PChar;
begin
 hMappedFile :=
CreateFileMapping(FHandle, nil, PAGE_READWRITE, 0, 0, 'SharedBlock');
 if (hMappedFile = 0) then
    ShowMessage('Mapping error!')
    else
    begin
      pSharedBuf :=
      MapViewOfFile(hMappedFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
      if (pSharedBuf = nil) then
         ShowMessage ('MapView error');
    end;
end;
```

После того как получен указатель pSharedBuf, вы можете работать со своим файлом как с обычной областью памяти, не заботясь о вводе, выводе, позиционировании и т. п. Все эти проблемы берет на себя файловая система.

Последние две функции, имеющие отношение к объекту файлового отображения, называются UnMapViewOfFile и CloseHandle. Функция UnMapViewOfFile делает то, что подразумевает ее название. Она прекращает отображение

в адресное пространство процесса того файла, который перед этим был отображен при помощи функции `MapViewOfFile`. Функция `CloseHandle` закрывает дескриптор объекта файлового отображения, возвращаемый функцией `CreateFileMapping`.

Функция `UnMapViewOfFile` должна вызываться перед функцией `CloseHandle`. Функции `UnMapViewOfFile` передает единственный параметр типа указатель:

```
procedure TClientForm.FormDestroy(Sender: TObject);
begin
 UnMapViewOfFile(pSharedBuf);
 CloseHandle(hFileMapObj);
end;
```

Отображаемые файлы уже будут использоваться и в других главах этой книги. Не стоит удивляться, ведь это очень мощный инструмент: помимо возможности совместного доступа он позволяет заметно ускорить доступ к файлам, особенно большого размера.

Резюме

При разработке приложений очень часто приходится решать задачи обмена данными между приложениями или приложением и устройством ввода/вывода. При этом большая часть созданного кода обеспечивает работу приложения с файлами.

В этой главе рассмотрены методы, обеспечивающие взаимодействие программы с файловой системой и примеры их использования.

Для организации обмена данными в приложениях используются специальные объекты — потоки, которые не только хранят информацию во время выполнения приложения, но и предоставляют разработчику набор стандартных свойств и методов для управления данными.

Затруднительно сказать, при изучении каких глав будет полезен материал этой главы. Скорее всего, он понадобится везде.

ГЛАВА 10

Использование графики

"Одно изображение стоит тысячи слов", — говорил древнекитайский император Сун; его слова верны и в наши времена. 80% информации мозг человека получает по зрительному каналу, и не удивительно, что программисты стараются придать внешнему виду своих программ максимум привлекательности.

Поэтому в Delphi с самого начала появились развитые средства для работы с графическими возможностями Windows. Этому набору объектов и посвящена данная глава.

Графические инструменты Delphi

Разработчики Delphi уделили большое внимание возможностям работы с деловой графикой: простота и удобство ее использования напрямую сказываются на простоте и удобстве созданных приложений. Вместо дебрей графического интерфейса Windows разработчик получил несколько инструментов, сколь понятных, столь же и мощных.

В стандартном графическом интерфейсе Windows (GDI) основой для рисования служит дескриптор контекста устройства `HDC` и связанные с ним шрифт, перо и кисть. В состав VCL входят объектно-ориентированные надстройки над последними, назначением которых является удобный доступ к свойствам инструментов и прозрачная для пользователя обработка всех их изменений.

Обязательным для любого объекта, связанного с графикой в Delphi, является событие:

```
property OnChange: TNotifyEvent;
```

Его обработчик вызывается всякий раз, когда меняются какие-то характеристики объекта, влияющие на его внешний вид.

Класс *TFont*

Класс инкапсулирует шрифт Windows. В Delphi допускаются только горизонтально расположенные шрифты. В конструкторе объекта по умолчанию принимается шрифт System, цвета `clWindowText` и размером 10 пунктов.

Свойства класса приведены в табл. 10.1.

Таблица 10.1. *Свойства класса* `TFont`

Свойство	Описание
`property Handle: HFont;`	Содержит дескриптор шрифта
`property Name: TFontName;`	Содержит имя (начертание) шрифта, например, Arial

`property Style: TFontStyles;`
`TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);`
`TFontStyles = set of TFontStyle;`
	Содержит стиль (особенности начертания) шрифта: соответственно жирный, курсив, подчеркнутый и перечеркнутый

`property Color: TColor;` `TColor = -(COLOR_ENDCOLORS + 1)..$2FFFFFF;`	Определяет цвет шрифта
`property Charset: TFontCharset` `TFontCharset = 0..255;`	Содержит номер набора символов шрифта. По умолчанию равно 1 (`DEFAULT_CHARSET`). Для вывода символов кириллицы требуется `RUSSIAN_CHARSET`
`property Pitch: TFontPitch;` `TFontPitch = (fpDefault, fpVariable, fpFixed);`	Определяет способ установки ширины символов шрифта. Значение `fpFixed` соответствует моноширинным шрифтам; `fpVariable` — шрифтам с переменной шириной символа. Установка `fpDefault` означает принятие того способа, который определен начертанием
`property Height: Integer;`	Содержит значение высоты шрифта в пикселах
`property PixelsPerInch: Integer;`	Определяет число точек на дюйм. Первоначально равно числу точек на дюйм в контексте экрана. Программист не должен изменять это свойство, т. к. оно используется системой для приведения изображения на экране и на принтере к одному виду
`property Size: Integer;`	Содержит размер шрифта в пунктах (как принято в Windows). Это свойство связано с `Height` соотношением: `Font.Size := -Font.Height*72/Font.PixelsPerInch`

Установка этих свойств вручную, как правило, не нужна. Если вы хотите изменить шрифт для какого-то компонента, воспользуйтесь компонентом `TFontDialog`. В нем можно и поменять свойства, и сразу увидеть получившийся результат на тестовой надписи; потом выбранный шрифт присваивается свойству `Font` нужного компонента:

```
if FontDialog1.Execute then  Edit1.Font := FontDialog1.Font;
```

Примечание

Если вы хотите, не закрывая диалог, увидеть результат применения шрифта на вашем тексте, включите опцию `fdApplyButton` в свойстве `Options` объекта `TFontDialog` и напишите для него обработчик события `OnApply`. При этом в диалоговом окне появится кнопка **Apply**, по нажатии которой (событие `OnApply`) можно изменить параметры шрифта.

Класс *TPen*

Этот класс инкапсулирует свойства пера GDI Windows. В конструкторе по умолчанию создается непрерывное (`psSolid`) черное перо шириной в один пиксел. Свойства класса приведены в табл. 10.2.

Таблица 10.2. *Свойства класса* `TPen`

Свойство	Описание
`property Handle: HPen;`	Содержит дескриптор пера
`property Color: TColor;`	Определяет цвет пера
`property Mode: TPenMode;` `TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor);`	Содержит идентификатор одной из растровых операций, которые определяют взаимодействие пера с поверхностью. Эти операции соответствуют стандартным, определенным в Windows
`property Style: TPenStyle;` `TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame);`	Определяет стиль линии, рисуемой пером. Соответствующие стили также определены в Windows
`property Width: Integer;`	Содержит значение толщины пера в пикселах

К сожалению, пунктирные и штрихпунктирные линии (стили `psDash`, `psDot`, `psDashDot`, `psDashDotDot`) могут быть установлены только для линий единич-

ной толщины. Более толстые линии должны быть сплошными — такое ограничение существует в Windows.

Примечание

Операция `pmNotXor` подходит для рисования перемещающихся линий или фигур, например, при выделении мышью какой-либо области. Если вы два раза нарисуете одну и ту же фигуру таким пером, то после первого раза она появится, после второго — полностью сотрется.

Класс *TBrush*

Этот класс инкапсулирует свойства кисти — инструмента для заливки областей. Когда создается экземпляр этого класса, первоначально используется белая сплошная (`Style=bsSolid`) кисть. Свойства класса приведены в табл. 10.3.

Таблица 10.3. *Свойства класса* `TBrush`

Свойство	Описание
`property Handle: HBrush;`	Содержит дескриптор кисти
`property Color: TColor;`	Определяет цвет кисти
`property Style: TBrushStyle;` `TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross);`	Определяет стиль кисти (фактура закраски)
`property Bitmap: TBitmap;`	Содержит битовую карту, определенную пользователем для закраски поверхностей. Если это свойство определено, то свойства `Color` и `Style` недействительны

Шрифт, перо и кисть не могут использоваться самостоятельно. Они являются составными частями специального класса, который и будет сейчас рассмотрен.

Класс *TCanvas*

Этот класс — сердцевина графической подсистемы Delphi. Он объединяет в себе и "холст" (контекст конкретного устройства GDI), и "рабочие инструменты" (перо, кисть, шрифт), и даже "подмастерьев" (набор функций по рисованию типовых геометрических фигур). Будем называть его *канвой*.

Канва не является компонентом, но она присутствует в качестве свойства во многих других компонентах, которые должны уметь нарисовать себя и отобразить какую-либо информацию.

Читатели, знакомые с графикой Windows, узнают в `TCanvas` объектно-ориентированную надстройку над контекстом устройства Windows (Device Context, DC). Дескриптор устройства, над которым "построена" канва, может быть востребован для различных низкоуровневых операций. Он задается свойством:

```
property Handle: HDC;
```

Для рисования канва включает в себя шрифт, перо и кисть:

```
property Font: TFont;
property Pen: TPen;
property Brush: TBrush;
```

Кроме того, можно рисовать и поточечно, получив доступ к каждому пикселу. Значение свойства:

```
property Pixels[X, Y: Integer]: TColor;
```

соответствует цвету точки с координатами X, Y.

Необходимость отрисовывать каждую точку возникает нередко. Однако, если нужно модифицировать все или хотя бы многие точки изображения, свойство `Pixels` надо сразу отбросить — настолько оно неэффективно. Гораздо быстрее редактировать изображение при помощи свойства `ScanLine` объекта `TBitmap`; об этом рассказано ниже.

Канва содержит методы-надстройки над всеми основными функциями рисования GDI Windows и свойства, которые приведены в табл. 10.4 и 10.5. При их рассмотрении имейте в виду, что все геометрические фигуры рисуются текущим пером. Те из них, которые можно закрашивать, закрашиваются с помощью текущей кисти. Кисть и перо при этом имеют текущий цвет.

Таблица 10.4. *Методы класса* `TCanvas`

Метод	Описание
`procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);`	Метод рисует сегмент эллипса. Эллипс определяется описывающим прямоугольником (X1, Y1)—(X2, Y2); его размеры должны лежать в диапазоне от 2 до 32 767 точек. Начальная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку (X3, Y3). Конечная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку (X4, Y4). Сегмент рисуется против часовой стрелки

Таблица 10.4 *(продолжение)*

Метод	Описание
`procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);`	Рисует хорду и заливает отсекаемую ею часть эллипса. Эллипс, начальная и конечная точки определяются, как в методе `Arc`
`procedure Ellipse(X1, Y1, X2, Y2: Integer);`	Рисует и закрашивает эллипс, вписанный в прямоугольник (X1, Y1) — (X2, Y2)
`procedure LineTo(X, Y: Integer);`	Проводит линию текущим пером из текущей точки в (X, Y)
`procedure MoveTo (X, Y: Integer);`	Перемещает текущее положение пера (свойство `PenPos`) в точку (X, Y)
`procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);`	Производит специальное копирование. Прямоугольник `Source` из битовой карты `Bitmap` копируется в прямоугольник `Dest` на канве; при этом цвет `Color` заменяется на цвет текущей кисти (`Brush.Color`). С помощью этого метода можно нарисовать "прозрачную" картинку. Для этого нужно выбрать соответствующий фону цвет кисти и затем заменить на него фоновый или наиболее часто встречающийся цвет битовой карты (см. `Bitmap.TransparentColor`)
`procedure CopyRect(const Dest: TRect; Canvas: TCanvas; const Source: TRect);`	Производит копирование прямоугольника `Source` из канвы `Canvas` в прямоугольник `Dest` в области самого объекта
`procedure FillRect(const Rect: TRect);`	Производит заливку прямоугольника (текущей кистью)
`procedure FrameRect(const Rect: TRect);`	Осуществляет рисование контура прямоугольника цветом текущей кисти (без заполнения)
`procedure Draw(X, Y: Integer; Graphic: Tgraphic);`	Осуществляет рисование графического объекта `Graphic` (точнее, вызов метода его рисования) в области с верхним левым углом (X, Y)
`procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);`	Осуществляет рисование объекта `Graphic` в заданном прямоугольнике `Rect`. Если их размеры не совпадают, `Graphic` масштабируется
`procedure DrawFocusRect(const Rect: TRect);`	Производит отрисовку прямоугольной рамки из точек (как на элементе, имеющем фокус ввода). Поскольку метод использует логическую операцию XOR (исключающее ИЛИ), повторный вызов для того же прямоугольника приводит изображение к начальному виду

Таблица 10.4 *(окончание)*

Метод	Описание
`procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TfillStyle);` `TFillStyle = (fsSurface, fsBorder);`	Производит заливку области текущей кистью. Процесс начинается с точки (X, Y). Если режим `FillStyle` равен `fsSurface`, то он продолжается до тех пор, пока есть соседние точки с цветом `Color`. В режиме `fsBorder` закрашивание, наоборот, прекращается при выходе на границу с цветом `Color`
`procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);`	Рисует сектор эллипса, описываемого прямоугольником (X1, Y1) — (X2, Y2). Стороны сектора лежат на лучах, проходящих из центра эллипса через точки (X3, Y3) и (X4, Y4)
`procedure Polygon(const Points: array of TPoint);`	Строит многоугольник, используя массив координат точек `Points`. При этом последняя точка соединяется с первой и внутренняя область закрашивается
`procedure Polyline(const Points: array of TPoint);`	Строит ломаную линию, используя массив координат точек `Points`
`procedure PolyBezier(const Points: array of TPoint);`	Строит кривую Безье (кубический сплайн), используя массив координат точек `Points`
`procedure PolyBezierTo(const Points: array of TPoint);`	Строит кривую Безье (кубический сплайн), используя массив координат точек `Points`. Текущая точка используется в качестве первой
`procedure Rectangle(X1, Y1, X2, Y2: Integer);`	Рисует прямоугольник с верхним левым углом в (X1, Y1) и нижним правым в (X2, Y2)
`procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);`	Рисует прямоугольник с закругленными углами. Координаты вершин — те же, что и в методе `Rectangle`. Закругления рисуются как сегменты эллипса с размерами осей по горизонтали и вертикали X3 и Y3
`function TextHeight(const Text: string): Integer;`	Задает высоту строки `Text` в пикселах
`function TextWidth(const Text: string): Integer;`	Задает ширину строки `Text` в пикселах
`procedure TextOut(X, Y: Integer; const Text: string);`	Производит вывод строки `Text`. Левый верхний угол помещается в точку канвы (X, Y)
`procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);`	Производит вывод текста с отсечением. Как и в `TextOut`, строка `Text` выводится с позиции (X, Y); при этом часть текста, лежащая вне пределов прямоугольника `Rect`, отсекается и не будет видна

Таблица 10.5. *Свойства класса* `TCanvas`

Свойство	Описание
`property ClipRect: TRect;`	Определяет область отсечения канвы. То, что при рисовании попадает за пределы этого прямоугольника, не будет изображено. Свойство доступно только для чтения — его значение переустанавливается системой в контексте устройства, с которым связана канва
`property PenPos: TPoint;`	Содержит текущую позицию пера канвы (изменяется посредством метода `MoveTo`)

Метод

```
procedure Refresh;
```

сбрасывает текущие шрифт, перо и кисть, заменяя их на стандартные, заимствованные из установок Windows (`BLACK_PEN`, `HOLLOW_BRUSH`, `SYSTEM_FONT`).

Предусмотрено два события для пользовательской реакции на изменение канвы:

```
property OnChange: TNotifyEvent;
property OnChanging: TNotifyEvent;
```

Эти события возникают при изменении свойств и вызове методов `TCanvas`, меняющих вид канвы (т. е. при любом рисовании. В методе `MoveTo`, например, они не возникают). Отличие их в том, что событие `OnChanging` вызывается до начала изменений, а событие `OnChange` — после их завершения.

Идентификатор (код) растровой операции при копировании прямоугольных блоков содержится в свойстве

```
property CopyMode: TCopyMode;
TCopyMode = Longint;
```

и определяет правило сочетания пикселов, копируемых на канву, с ее текущим содержимым. При этом можно создавать разные изобразительные эффекты. В Delphi определены следующие константы кодов: `cmBlackness`, `cmDstInvert`, `cmMergeCopy`, `cmMergePaint`, `cmNotSrcCopy`, `cmNotSrcErase`, `cmPatCopy`, `cmPatInvert`, `cmPatPaint`, `cmSrcAnd`, `cmSrcCopy`, `cmSrcErase`, `cmSrcInvert`, `cmSrcPaint`, `cmWhiteness`.

Все они стандартно определены в Windows, и подробное их описание можно найти в документации по GDI. Значением свойства `CopyMode` по умолчанию является `cmSrcCopy` — копирование пикселов источника поверх существующих.

Из возможностей, появившихся в классе `TCanvas`, следует отметить поддержку рисования кривых (полиномов) Безье. Эта возможность впервые

появилась в API Windows NT. Для построения одной кривой нужны минимум четыре точки — начальная, конечная и две опорные. По ним будет построена кривая второго порядка. Если задан массив точек, они используются для построения последовательных кривых, причем последняя точка одной кривой является первой для следующей кривой.

Хорошей иллюстрацией использования объекта `TCanvas` может служить пример `GraphEx`, поставляемый вместе с Delphi (папка \Demos\Doc\GraphEx). Есть только одно "но" — он приводится в неизменном виде, начиная с версии Delphi 1.0. Поэтому сделаем часть работы за программистов Borland. В нашем примере модернизированы Панели инструментов — они выполнены на компонентах `TToolBar` и `TControlBar`; добавлена поддержка файлов JPEG; и, наконец, добавлена возможность рисования кривых Безье. Обновленный внешний вид главной формы примера GraphEx показан на рис. 10.1.

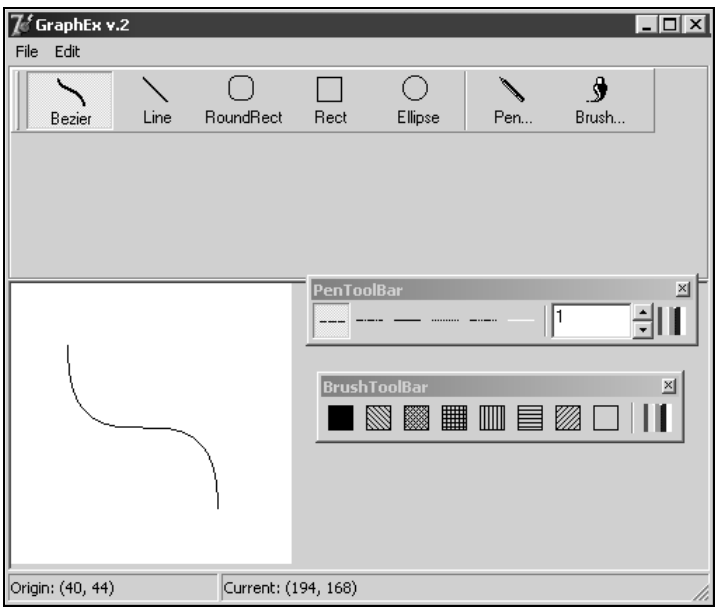


Рис. 10.1. Так теперь выглядит главная форма примера GraphEx

Где же найти ту канву, на которой предстоит рисовать? Во-первых, ею снабжены все потомки классов `TGraphicControl` и `TCustomControl`, т. е. почти все визуальные компоненты из Палитры компонентов; в том числе и форма. Во-вторых, канву имеет растровая картинка (класс `TBitmap`); вы можете писать и рисовать не на пустом месте, а на готовом изображении *(об этом см. ниже в разд. "Класс TBitmap" данной главы)*. Но иногда нужно рисовать и прямо на экране. В этом случае придется прибегнуть к использованию

функций API. Функция `GetDC` возвращает контекст устройства заданного окна, если ей передается параметр 0 — то всего экрана:

```
ScreenCanvas := TCanvas.Create;
ScreenCanvas.Handle := GetDC(0);
// Рисование на ScreenCanvas
ReleaseDC(0, ScreenCanvas.Handle);
ScreenCanvas.Free;
```

Пример необходимости рисования на экране — программы сохранения экрана (Screen savers).

Когда и где следует рисовать? Этот вопрос далеко не риторический, как может показаться с первого взгляда.

Помимо графических примитивов, таких как линии и фигуры, на канве можно разместить готовые изображения. Для их описания создан класс `TGraphic`.

Класс *TGraphic*

Канва, перо, кисть и шрифт нужны, чтобы нарисовать свою картинку. Чтобы загрузить готовую, необходимы объекты, "понимающие" графические форматы Windows.

Абстрактный класс `TGraphic` является родительским для трех видов изображений, общепринятых в графике Windows — значка (компонент `TIcon`), метафайла (компонент `TMetafile`) и растровой картинки (компонент `TBitmap`). Четвертым потомком `TGraphic` является `TJPEGImage` — сжатая растровая картинка в формате JPEG.

Работая над приложением в Delphi, вы никогда не будете создавать объекты класса `TGraphic`, но переменной этого типа вы можете присваивать указатель на любой из перечисленных классов-потомков.

Метод:

```
procedure Assign(Source: TPersistent);
```

переопределяет одноименный метод предка, допуская полиморфное присваивание графических объектов.

Загрузку и выгрузку графики в поток осуществляют методы:

```
procedure LoadFromStream(Stream: TStream);
procedure SaveToStream(Stream: TStream);
```

а загрузку и выгрузку в файл — методы:

```
procedure LoadFromFile(const Filename: string);
procedure SaveToFile(const Filename: string);
```

Эти методы создают соответствующий файловый поток и затем вызывают методы `LoadFromStream/SaveToStream`.

Два метода осуществляют взаимодействие с буфером обмена Windows:

```
procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle;
APalette: HPALETTE);
procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle;
var APalette: HPALETTE);
```

Здесь `AFormat` — используемый графический формат; `AData` и `APalette` — данные и палитра (если она требуется). Потомок должен иметь свой формат представления в буфере обмена и уметь обрабатывать данные, представленные в нем.

Загрузка больших графических файлов может продолжаться очень долго. Чтобы скрасить пользователю ожидание, программист может обработать событие `OnProgress`:

```
type
TProgressStage = (psStarting, psRunning, psEnding);
TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage;
PercentDone: Byte; RedrawNow: Boolean; const
 R: TRect; const Msg: string) of object;
property OnProgress: TProgressEvent;
```

Оно вызывается графическими объектами во время длительных операций. Параметр `Stage` означает стадию процесса (начало/протекание/завершение), а `PercentDone` — процент сделанной работы. Сразу оговоримся, что не все из тех объектов, которые будут нами описаны, вызывают обработчик события `OnProgress`.

Свойство:

```
property Empty: Boolean;
```

устанавливается в значение `True`, если графический объект пуст (в него не загружены данные).

Высота и ширина графического объекта задаются свойствами:

```
property Height: Integer;
property Width: Integer;
```

Для каждого дочернего типа эти параметры вычисляются своим способом. Наконец, свойство:

```
property Modified: Boolean;
```

показывает, модифицировался ли данный графический объект. Это свойство устанавливается в значение `True` внутри обработчика события `OnChange`.

Многие графические объекты при отрисовке должны быть прозрачными. Одни из них прозрачны всегда (значок, метафайл), другие — в зависимости от значения свойства

```
property Transparent: Boolean;
```

Класс *TPicture*

Это класс-надстройка над `TGraphic`, точнее — над его потомками. Он имеет поле `Graphic`, которое может содержать объекты классов `TBitmap`, `TIcon`, `TMetafile` и `TJPEGImage`. Предназначение класса `TPicture` — управлять вызовами соответствующих методов, скрывая при этом хлопоты с определением типа графического объекта и детали его реализации.

Кроме того, на уровне `TPicture` определены возможности регистрации и использования других — определенных пользователем — классов графических объектов, порожденных от `TGraphic`. Доступ к графическому объекту осуществляется посредством свойства:

```
property Graphic: TGraphic;
```

Если графический объект имеет один из трех предопределенных типов, то к нему можно обратиться и как к одному из свойств:

```
property Bitmap: TBitmap;
property Icon: TIcon;
property Metafile: TMetafile;
```

Обращаясь к этим функциям, нужно быть осторожным. Если в поле `Graphic` хранился объект одного класса, а затребован объект другого класса, то прежний объект уничтожается, а вместо него создается пустой объект требуемого класса. Например:

```
Image1.Picture.LoadFromFile('myicon.ico'); //Создан и загружен объект
класса TIcon
MyBitmap := Image1.Picture.Bitmap; // прежний TIcon уничтожается
```

Если же вы описали свой класс (допустим, `TGIFImage`), то к его методам и свойствам следует обращаться так:

```
(Graphic as TGIFImage).MyProperty := MyValue;
```

Перечислим остальные методы и свойства.

- `procedure LoadFromFile(const Filename: string);`

  Анализирует расширение имени файла `FileName` и если оно известно (зарегистрировано), то создается объект нужного класса и вызывается его метод `LoadFromFile`. В противном случае возникает исключительная ситуация `EInvalidGraphic`. Стандартными расширениями являются ico, wmf

(emf) и bmp. Если подключить к приложению модуль JPEG.PAS, то можно будет загрузить и файлы с расширениями jpg и jpeg.

- `procedure SaveToFile(const Filename: string);`

  Сохраняет графику в файле, вызывая соответствующий метод объекта `Graphic`.

- `procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle; APalette: HPALETTE);`

  Во многом аналогичен методу `LoadFromFile`. Если формат `AFormat` найден среди зарегистрированных, то `AData` и `APalette` передаются для загрузки методу соответствующего объекта. Изначально зарегистрированных форматов три: битовое изображение `CF_BITMAP`, метафайлы `CF_METAFILEPICT` и `CF_ENHMETAFILE`.

- `procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle; var APalette: HPALETTE);`

  Сохраняет графику в буфере обмена, вызывая метод объекта `Graphic`.

- `procedure Assign(Source: TPersistent);`

  Метод `Assign` переписан таким образом, чтобы присваиваемый объект мог принадлежать как классу `TPicture`, так и `TGraphic` или любого его потомка. Кроме того, он может быть равен `nil` — в этом случае поле `Graphic` очищается с удалением прежнего объекта.

- `class function SupportsClipboardFormat(AFormat: Word): Boolean;`

  Метод класса возвращает значение `True`, если формат `AFormat` поддерживается классом `TPicture` (зарегистрирован в системе). Напомним, что методы класса можно вызывать через ссылку на класс без создания экземпляра объекта.

- `class procedure RegisterFileFormat(const AExtension, ADescription: string; AGraphicClass: TGraphicClass);`
  `class procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass: TGraphicClass);`

  Предназначены для создателей новых графических классов. Они позволяют зарегистрировать формат файла и буфера обмена и связать их с созданным классом — потомком `TGraphic`, который умеет читать и записывать информацию в этом формате.

- `property Width: Integer;`
  `property Height: Integer;`

  Ширина и высота картинки. Значения этого свойства всегда те же, что и у объекта из свойства `Graphic`.

Все три разновидности графических объектов имеют свои системы кэширования. Это означает, что на один реально существующий в системе (и за-

нимающий долю ресурсов!) дескриптор могут одновременно ссылаться несколько объектов. Реализуется такое связывание через метод `Assign`. Выражение:

```
Icon1.Assign(Icon2);
```

означает, что два этих объекта разделяют теперь один, фактически находящийся в памяти, значок.

Более простым является кэширование для классов `TIcon` и `TMetafile`, которые умеют только отображать себя и не предназначены для редактирования (создатели Delphi считают, что дескриптор графического объекта дается программисту не для того, чтобы "ковыряться" в нем на уровне двоичных кодов). Гораздо сложнее устроен механизм кэширования для канала `TBitmap`, который имеет свою канву для рисования.

Внутреннее представление информации в графических объектах двоякое — она может храниться как поток типа `TMemoryStream` (в него загружается содержимое соответствующего файла), как область памяти с дескриптором (структура которой зависит от типа графического объекта) и одновременно в двух этих видах, содержимое которых автоматически синхронизируется. Поэтому будьте готовы, что загрузка изображения потребует вдвое большего объема памяти — особенно это актуально для больших картинок.

Кого-то может удивить отсутствие объявленных методов рисования, вроде метода `Draw` для классов `TIcon`, `TMetafile` и `TBitmap`. Объяснение простое — в процессе рисования они играют пассивную роль; рисуют не они — рисуют их. Все рисование должно осуществляться через вызовы методов `Draw` и `StretchDraw` канвы, содержащей графику, ибо канва соответствует тому контексту, в котором должна осуществляться операция.

Рассмотрим предопределенные графические классы.

Класс *TMetafile*

Инкапсулирует свойства метафайла Windows. С появлением Windows 95 к стандартному метафайлу (формат WMF) добавился расширенный (формат EMF), обладающий расширенными возможностями. Соответственно в объекте `TMetafile` имеется свойство

```
property Enhanced: Boolean;
```

Внутреннее представление метафайла всегда новое (EMF), и устанавливать свойство `Enhanced` в значение `False` следует только для обеспечения совместимости со старыми программами.

В классе `TMetafile` перекрываются методы `Assign`, `LoadFromStream`, `SaveToStream`, `LoadFromClipboardFormat`, `SaveToClipboardFormat`. В буфер об-

мена объект помещает свое содержимое в формате `CF_ENHMETAFILE`. Помимо общих, класс имеет следующие свойства:

- дескриптор метафайла

  `property Handle: HMETAFILE;`

- свойство `property Inch: Word`. Число точек на дюйм в координатной системе метафайла. Связано с установленным режимом отображения;
- свойства

  ```
  property MMHeight: Integer;
  property MMWidth: Integer;
  ```

  это настоящие высота и ширина метафайла в единицах, равных 0,01 мм. Свойства `Height` и `Width` задаются в пикселах;

- в метафайл можно добавить свою палитру:

  `property Palette: HPalette;`

- вы можете увековечить себя, установив два свойства метафайла:

  ```
  property Description: string;
  property CreatedBy: string;
  ```

  Содержащаяся в них информация записывается в файл и может быть прочитана благодарными потомками.

Класс *TIcon*

Этот класс инкапсулирует значок Windows.

Не пытайтесь изменить размеры значка — они по определению постоянны (и равны `GetSystemMetrics(SM_CXICON)` и `GetSystemMetrics(SM_CYICON)`), и при попытке присвоить новые значения возникает исключительная ситуация `EInvalidGraphicOperation`. Значок нельзя также читать и писать в буфер обмена, т. к. в Windows нет соответствующего формата.

Свойство `Transparent` для значка всегда равно значению `True`. Изменить его нельзя — значки прозрачны также по определению.

В этом классе перекрываются методы класса `TGraphic: Assign`, `LoadFromStream` и `SaveToStream`. Дополнительно также определены:

- `property Handle: HICON;` — дескриптор значка;
- `function ReleaseHandle: HICON;` — метод "отдает" дескриптор — возвращает его значение, обнуляя ссылку на него в объекте.

Класс *TBitmap*

Класс `TBitmap` является основой растровой графики в Delphi. В первых версиях среды этот класс соответствовал битовой карте, зависимой от устрой-

ства (Device Dependent Bitmap, DDB). Этот формат хорош для деловой графики — отображения небольших картинок с малой глубиной цвета, например, на кнопках. Формат DDB появился во времена первых версий Windows, когда еще не было графических ускорителей и кое-где еще помнили о EGA. Поэтому и форматы хранения были привязаны к определенным видеорежимам.

Со временем аппаратура совершенствовалась, росло и количество поддерживаемых видеорежимов. Появились режимы High Color (15—16 бит на точку) и True Color (24 бита на точку). Все это привело к тому, что картинка стала храниться в аппаратно-независимом формате (Device Independent Bitmap, DIB), а проблемы ее быстрого отображения легли на аппаратуру и драйверы.

За формат битовой карты — DIB или DDB — отвечает свойство:

```
type TBitmapHandleType = (bmDIB, bmDDB);
property HandleType: TBitmapHandleType;
```

По умолчанию устанавливается режим `bmDIB`. Впрочем, можно заставить приложение, написанное на Delphi, вернуться к старому типу. Для этого нужно установить глобальную переменную `DDBsOnly` (модуль GRAPHICS.PAS) в значение `True`. Впрочем, необходимость этого сомнительна. Все новые видеокарты и драйверы к ним, а также графические интерфейсы (такие, как DirectX) оптимизированы для использования DIB.

Желаемую глубину цвета битовой карты можно узнать и переустановить, меняя значение свойства:

```
TPixelFormat = (pfDevice, pf1bit, pf4bit, pf8bit, pf15bit, pf16bit,
pf24bit, pf32bit, pfCustom);
property PixelFormat: TPixelFormat;
```

Режим `pfDevice` соответствует битовой карте DDB. Глубина цвета в 1, 4 и 8 бит на пиксел — традиционная и предусматривает наличие у изображения палитры. Другие режимы заботятся о хранении непосредственных яркостей точек в каждом из трех основных цветов — красном (R), зеленом (G) и синем (B). Разрядность 15 бит соответствует распределению бит 5-5-5 (RGB555), 16 бит — RGB 565, 24 бит — RGB888. Режим 32 бит похож на 24-битный, но в нем дополнительно добавлен четвертый канал (альфа-канал), содержащий дополнительную информацию о прозрачности каждой точки. Режим `pfCustom` предназначен для реализации программистом собственных графических конструкций. В стандартном классе `TBitmap` установка свойства `PixelFormat` в режим `pfCustom` приведет к ошибке — поэтому использовать его нужно только в написанных вами потомках `TBitmap`.

Битовая карта является одним из видов ресурсов. Естественно, что класс `TBitmap` поддерживает загрузку из ресурсов приложения:

```
procedure LoadFromResourceID(Instance: THandle; ResID: Integer);
procedure LoadFromResourceName(Instance: THandle; const ResName: string);
```

Здесь `Instance` — это глобальная переменная модуля `System`, хранящая уникальный идентификатор запущенной копии приложения (или динамической библиотеки).

Канва битовой карты доступна через свойство:

```
property Canvas: TCanvas;
```

С ее помощью можно рисовать на поверхности растрового изображения. Обратите внимание, что никакие другие потомки `TGraphic` канвы не имеют.

Дескрипторы битовой карты и ее палитры доступны как свойства:

```
property Handle: HBITMAP;
property Palette: HPALETTE;
```

Имея дело с классом `TBitmap`, учитывайте, что принцип "один объект — один дескриптор" из-за наличия механизма кэширования неверен. Два метода:

```
function ReleaseHandle: HBITMAP;
function ReleasePalette: HPALETTE;
```

возвращают дескрипторы битовой карты и палитры соответственно, а после этого обнуляют дескрипторы, т. е. как бы "отдают" их пользователю.

При любом внешнем обращении к дескриптору битовой карты и любой попытке рисовать на ее канве разделение одной картинки несколькими объектами прерывается, и объект получает собственную копию содержимого дескриптора. Для этого есть методы:

- `procedure Dormant` — выгружает изображение в поток и уничтожает дескрипторы битовой карты и палитры;
- `procedure FreeImage` — "освобождающий" дескриптор битовой карты для дальнейшего использования и внесения изменений. Это означает, что если на данный дескриптор есть ссылки, то он дублируется; поток очищается.

Битовая карта может быть монохромной и цветной, что определено свойством:

```
property Monochrome: Boolean;
```

Значение `True` соответствует монохромной битовой карте. При его изменении происходит преобразование содержимого к требуемому виду.

За прозрачность битовой карты отвечают следующие свойства:

```
property TransparentColor: TColor;
type TTransparentMode = (tmAuto, tmFixed);
property TransparentMode: TTransparentMode;
```

Если свойство TransparentMode установлено в режим tmAuto, то за прозрачный (фоновый) принимается цвет верхнего левого пиксела. В противном случае этот цвет берется из свойства TransparentColor.

Битовая карта может использоваться в качестве маски для других битовых карт. В этом случае она превращается в двухцветную, где в белый цвет окрашиваются точки фона (см. свойство TransparentColor), а в черный — все остальные. Для поддержки этого режима служат следующие методы и свойства:

```
procedure Mask(TransparentColor: TColor);
property MaskHandle: HBitmap;
function ReleaseMaskHandle: HBitmap;
```

Наконец, последним по счету будет рассмотрено очень важное свойство битовой карты TBitmap. Если формат ее хранения — DIB, то есть возможность получить доступ к данным самой битовой карты:

```
property ScanLine[Row: Integer]: Pointer;
```

Это свойство представляет собой массив указателей на строки с данными битовой карты. Параметр Row содержит номер строки. Следует помнить, что в большинстве случаев строки в битовой карте упорядочены в памяти снизу вверх и фактически первой после заголовка хранится нижняя строка. Код, возвращающий значение свойства ScanLine, это учитывает; поэтому не удивляйтесь, если с ростом параметра Row значение свойства уменьшается.

Внутри строки данные упорядочены в соответствии с форматом (PixelFormat). Для формата pf8bit все просто — каждый байт в строке соответствует одному пикселу. Для форматов pf15bit и pf16bit пикселу соответствуют два байта (в этих 16 битах упакованы данные о трех каналах), pf24bit — три байта (по байту на канал).

Примерно так может выглядеть обработчик события OnMouseMove, выводящий на панель состояния информацию о яркости в данной точке (подразумевается, что формат битовой карты — 8 или 24 бита):

```
procedure TMainForm.Image1MouseMove(Sender: TObject; Shift: TShiftState;
X, Y: Integer);
begin
  if not Assigned(Image1.Picture.Bitmap) then Exit;
with Image1.Picture.Bitmap. do
case PixelFormat of
 pf8bit: Statusbar1.SimpleText := Format('x: %d y: %d b: %d',[x, y,
pByteArray(ScanLine[y])^[x] ]);
 pf24bit: Statusbar1.SimpleText := Format('x: %d y: %d R: %d,G: %d, B: %d',
 [x,y, pByteArray(ScanLine[y])^[3*x], pByteArray(ScanLine[y])^[ 3*x+1],
pByteArray(ScanLine[y])^[ 3*x+2]]);
end;
```

Само значение свойства `ScanLine` изменить нельзя (оно доступно только для чтения). Но можно изменить данные, на которые оно указывает. Вот так можно получить негатив 24-битной картинки:

```
Var line : pByteArray;
...
For i:=0 to Image1.Picture.Bitmap.Height – 1 do
 Begin
 Line := Image1.Picture.Bitmap.ScanLine[i];
 For j:=0 to Image1.Picture.Bitmap.Width * 3 – 1 do
 Line^[j] := 255 – Line^[j];
 End;
```

Если вы хотите решать более серьезные задачи — на уровне профессиональных средств — на помощь может прийти библиотека обработки изображений фирмы Intel (Intel Image Processing Library). Этот набор инструментов позволяет разработчику включать в программы алгоритмы обработки изображений, написанные и оптимизированные специально для процессоров фирмы Intel. Библиотека является свободно распространяемой, и последняя ее версия располагается на Web-сайте фирмы. Интерфейсы к функциям библиотеки для Delphi разработаны авторами этой книги и вместе с примерами находятся на прилагаемой к книге дискете.

Примечание

В Delphi можно столкнуться с "тезкой" рассматриваемого объекта — структурой `TBitmap`, описанной в файле WINDOWS.PAS. Поскольку обе они относятся к одной и той же предметной области, часто возникают коллизии, приводящие к ошибкам. Напомним, чтобы отличить структуры-синонимы, следует использовать имя модуля, в котором они описаны. Поэтому если в вашей программе есть модули Windows и Graphics, то описывайте и употребляйте типы `Windows.TBitmap` и `Graphics.TBitmap`.

В состав Windows входят карточные игры (точнее, пасьянсы), которые черпают ресурсы из динамической библиотеки cards.dll. Если вы откроете эту библиотеку в редакторе ресурсов, то увидите там изображения всех пятидесяти двух карт и десятка вариантов их рубашек (оборотных сторон). Используем эту возможность для рисования карт. Так загружается битовая карта для рубашки:

```
var CardsDll : THandle;
     BackBitmap : Graphics.TBitmap;
initialization
 CardsDll := LoadLibraryEx('cards.dll',0, LOAD_LIBRARY_AS_DATAFILE);
 BackBitmap := Graphics.TBitmap.Create;
 BackBitmap.LoadFromResourceID(CardsDll,64);
```

```
finalization
 BackBitmap.Free;
 FreeLibrary(CardsDll);
end.
```

Примечание

В Windows 95/98 эта динамическая библиотека — 16-разрядная, и работать так, как описано, не будет. Используйте библиотеку Cards.dll из состава Windows NT, 2000.

Аналогичным образом можно загрузить битовые карты для всей колоды. При показе карты, в зависимости от того, открыта она или закрыта, отрисовывается один из объектов `TBitmap`:

```
if Known then // карта открыта
   Canvas.StretchDraw(ClientRect, FaceBitmap)
  else
   Canvas.StretchDraw(ClientRect,BackBitmap)
 end;
```

Графический формат JPEG. Класс *TJPEGImage*

В 1988 году был принят первый международный стандарт сжатия неподвижных изображений. Он был назван по имени группы, которая над ним работала — JPEG (Joint Photographic Expert Group). Дело в том, что стандартные архиваторы (ZIP, ARJ) и традиционные алгоритмы сжатия в форматах GIF, TIFF и PCX не могут достаточно сильно сжать полутоновую или цветную картинку (типа фотографии) — максимум в 2—3 раза. Примененный в JPEG алгоритм позволяет достичь сжатия в десятки раз — правда, при этом изображение подвергается необратимому искажению, и из него пропадает часть деталей. Бессмысленно (и вредно!) подвергать хранению в формате JPEG чертежи, рисунки, а также любые изображения с малым числом градаций — он предназначен именно для изображений фотографического качества.

Поддержка формата JPEG реализована в Delphi посредством класса `TJPEGImage`, который является потомком класса `TGraphic`.

Примечание

Название `TJPEGImage` не совсем удачное. К `TImage` этот класс не имеет ни малейшего отношения. Скорее, это "двоюродный брат" класса `TBitmap`.

К такому объекту предъявляются двоякие требования. С одной стороны, он должен поддерживать сжатие данных для записи на диск. С другой — рас-

пакованные данные в формате DIB, чтобы по требованию системы отрисовать их. Поэтому объект класса `TJPEGImage` может хранить оба вида данных, а также производить их взаимные преобразования, т. е. сжатие и распаковку. Для этого в нем предусмотрены методы:

```
procedure Compress;
procedure DIBNeeded;
procedure JPEGNeeded;
```

Рекомендуется вызывать метод `DIBNeeded` заранее, перед отрисовкой картинки — это ускорит процесс ее вывода на экран.

Кроме того, полезно использовать метод `Assign`, который позволяет поместить в класс `TJPEGImage` объект `TBitmap` и наоборот:

```
MyJPEGImage.Assign(MyBitmap);
MyBitmap.Assign(MyJPEGImage);
```

При этом происходит преобразование форматов.

Свойства `TJPEGImage` можно условно разделить на две группы: используемые при сжатии и при распаковке.

Важнейшим из свойств, нужных при сжатии, является `CompressionQuality`:

```
type TJPEGQualityRange = 1..100;
property CompressionQuality: TJPEGQualityRange;
```

Оно определяет качество сжимаемого изображения и его размер. При малых значениях этого свойства файлы получаются очень маленькими, но с большими искажениями (напомним, что стандарт JPEG предусматривает сжатие с потерями качества). При значениях, близких к 100, потери незаметны, но и размер файла при этом максимален.

Примечание

Заранее предсказать размер сжатого файла нельзя — разные картинки сжимаются по-разному, даже при одном значении `CompressionQuality`.

По умолчанию значение этого свойства равно 75, что обеспечивает разумный компромисс между размером и качеством.

Кроме `CompressionQuality`, на качество отображения может повлиять и свойство

```
type TJPEGPerformance = (jpBestQuality, jpBestSpeed);
property Performance: TJPEGPerformance;
```

Оно нужно только при распаковке и отвечает за способ восстановления цветовой палитры из сжатой информации. На качество записываемого изображения оно никак не влияет.

Как и у класса TBitmap, у TJPEGImage есть свойство

```
type TJPEGPixelFormat = (jf24Bit, jf8Bit);
property PixelFormat: TJPEGPixelForm;
```

Для рассматриваемого объекта возможных значений всего два — jf8bit и jf24bit. По умолчанию используется 24-битный формат. Если информация о цвете не нужна, то можно установить свойство GrayScale в значение True — в этом случае изображение будет записано (или распаковано) в полутоновом виде (256 оттенков серого).

Свойства ProgressiveEncoding и ProgressiveDisplay определяют способ показа изображения при распаковке. Первое из них отвечает за порядок записи в файл сжатых компонентов. Если ProgressiveEncoding установлено в значение True, начинает играть роль свойство ProgressiveDisplay. От его значения зависит, будет ли показываться изображение по мере распаковки (при значении True), либо будет сначала полностью распаковано, а потом показано (при значении False).

Чтобы организовать предварительный просмотр большого числа больших изображений, уместно воспользоваться свойством:

```
type TJPEGScale = (jsFullSize, jsHalf, jsQuarter, jsEighth);
property Scale: TJPEGScale;
```

Искушенные в графике специалисты зададут вопрос: зачем оно? Ведь можно прочитать изображение, а затем уменьшить его масштаб стандартными способами? Представление информации в файлах JPEG таково, что можно достаточно просто извлечь изображение сразу в нужном масштабе. Таким образом достигается двойной выигрыш — на времени распаковки и на времени отображения.

Примечание

Печать растровых изображений может вызвать проблемы при согласовании его размеров с размерами листа принтера и его разрешением. Большую часть из них можно снять, изучив пример, поставляемый с Delphi — jpegProj. Он находится не в папке \Demos, как обычно, а в папке Help\Examples\Jpeg.

В заключение отметим, что класс TJPEGImage не имеет своей канвы для рисования — для этого его нужно преобразовать в классе TBitmap.

Компонент *TImage*

Этот компонент служит надстройкой над классом TPicture и замыкает всю иерархию графических объектов VCL. Именно на его поверхности и будут отображаться графические объекты, содержащиеся в свойстве:

```
property Picture: TPicture;
```

В качестве канвы компонента (свойство `Canvas`) используется канва объекта из свойства `Picture.Graphic`, но только если поле `Graphic` ссылается на объект класса `TBitmap`. Если это не так, то попытка обращения к свойству вызовет исключительную ситуацию `EInvalidOperation`, т. к. рисовать на метафайле или значке нельзя.

Следующие три свойства определяют, как именно графический объект располагается в клиентской области компонента:

- `property AutoSize: Boolean;`

  Означает, что размеры компонента настраиваются по размерам содержащегося в нем графического объекта. Устанавливать его в значение `True` нужно перед загрузкой изображения из файла или буфера обмена.

- `property Stretch: Boolean;`

  Если это свойство установлено в значение `True`, то изображение "натягивается" на клиентскую область, при необходимости уменьшая или увеличивая свои размеры. Если оно установлено в `False`, то играет роль следующее свойство `Center`.

- `property Center: Boolean;`

  Если это свойство установлено в значение `True`, изображение центрируется в пределах клиентской области. В противном случае оно располагается в ее верхнем левом углу.

Несмотря на то, что описанию свойств и методов графических объектов здесь отведено уже довольно много места, работа с ними проста и удобна. Программу для просмотра изображений в среде Delphi можно создать буквально "в три счета":

1. Поместите на форму следующие компоненты: область прокрутки `TScrollBox`, на нее — компонент `TImage` (их верхние левые углы должны совпадать), любую кнопку (например, `TButton`) и диалог открытия файлов `TOpenPictureDialog`.
2. Подключите к главному модулю создаваемого приложения модуль JPEG (в предложении `uses`); свойство `AutoSize` компонента `TImage` установите в значение `True`.
3. Дважды щелкните мышью на кнопке. В появившемся обработчике события `OnClick` напишите такой код:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
 OpenPictureDialog1.Filter := GraphicFilter(TGraphic);
 if OpenPictureDialog1.Execute
 then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Приложение готово. Обратите внимание на роль полиморфизма в методе `LoadFromFile` — по расширению файла определяется его формат и в зависимости от этого создается нужный графический объект.

Использование диалогов для загрузки и сохранения графических файлов

Для удобства открытия картинок существует пара компонентов-диалогов: `TOpenPictureDialog` и `TSavePictureDialog`.

Список форматов открываемых файлов определяется свойством `Filter`. Можно, как в случае со стандартными диалогами `TOpenDialog` или `TSaveDialog`, сформировать их вручную с помощью редактора свойства `Filter`. Можно поступить проще, воспользовавшись готовыми средствами. Для удобства формирования строк графических фильтров существуют три специальные функции:

- `function GraphicFilter(GraphicClass: TGraphicClass): string;`

  Формирует строку с полным текстом графического фильтра, позволяющего открывать все файлы, форматы которых являются потомками параметра `GraphicClass`. Если в качестве параметра этой функции будет передан класс `TGraphic`, то в строке будут перечислены все форматы:

  ```
  'All (*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf)|*.jpg;*.jpeg;*.bmp;*.ico;
  *.emf;*.wmf|JPEG Image File (*.jpg)|*.jpg|JPEG Image File
  (*.jpeg)|*.jpeg|Bitmaps (*.bmp)|*.bmp|Icons (*.ico)|*.ico|Enhanced
  Metafiles (*.emf)|*.emf|Metafiles (*.wmf)|*.wmf'
  ```

  Формат JPEG появляется в перечне, если в приложении используется модуль с тем же названием — JPEG. В приводимом ниже примере возникла необходимость совместить фильтры только для классов `TBitmap` и `TJPEGImage`, которые не являются предками друг друга. В этом случае получившиеся строки нужно соединить, использовав символ конкатенации "|":

  ```
  S := GraphicFilter(TBitmap)+'|'+GraphicFilter(TJpegImage)
  ```

- `function GraphicExtension(GraphicClass: TGraphicClass): string;`

  Возвращает расширение файла, формат которого соответствует графическому классу `GraphicClass`. Так, если передать в качестве параметра класс `TBitmap`, то функция вернет строку `'BMP'`;

- `function GraphicFileMask(GraphicClass: TGraphicClass): string;`

  Эта функция возвращает перечень расширений файлов с форматами — потомками `GraphicClass`, перечисленных через точку с запятой.

Для диалогов предусмотрено несколько событий, которые программист может обработать самостоятельно. Первые три — достаточно тривиальные: `OnShow`, `OnCanClose` и `OnClose`. Нужно предостеречь программиста: по чьему-то недосмотру последние два вызываются только в случае нормального завершения диалога (нажатием кнопки **Open** или **Save**), а если завершить диалог нажатием кнопки **Cancel** или "крестика" на заголовке диалога, то управления они не получат.

Другие три события связаны с изменениями, которые осуществляет пользователь во время выбора нужного файла. Они происходят в момент изменения формата открываемого файла (событие `OnTypeChange`), изменения текущей папки (`OnFolderChange`) и текущего файла (`OnSelectionChange`).

Но разработчики диалогов не предусмотрели одну очень нужную возможность. Дело в том, что у разных графических форматов возможны различные опции и варианты. Если вы имеете опыт работы с графическими пакетами вроде Adobe Photoshop или Corel, то знаете, что, в зависимости от выбранного формата сохранения данных, диалог изменяет свой внешний вид — к нему добавляются элементы управления, соответствующие параметрам формата.

Поступим так и мы, предусмотрев настройку при сохранении файлов формата JPEG. Для этого будет использовано событие `OnTypeChange` компонента `TSavePictureDialog`. Для события нужно проверить значение свойства `FilterIndex`. Если оно больше 1 (т. е. выбраны файлы формата JPEG), нужно увеличить высоту окна диалога и разместить на нем дополнительные компоненты: флажок, соответствующий свойству `ProgressiveEncoding`, и редактор свойства `CompressionQuality` (рис. 10.2). Если тип файла снова поменялся и стал равным 1, нужно эти компоненты убрать.

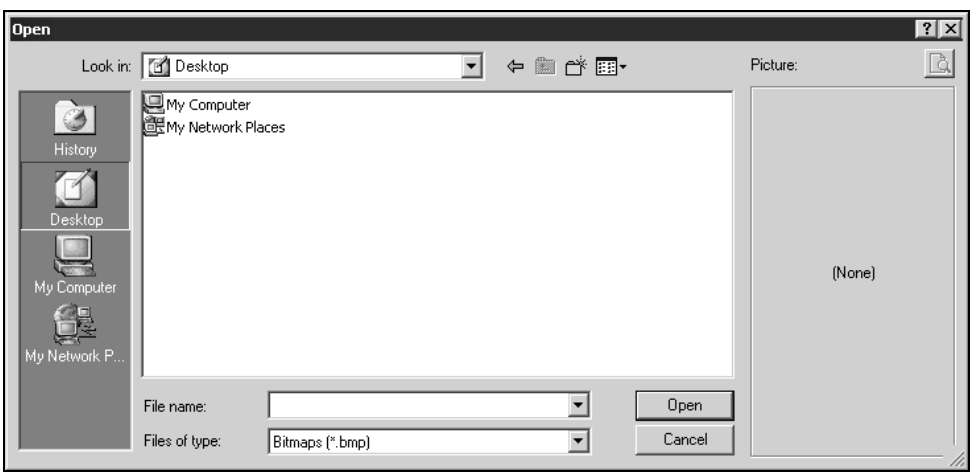


Рис. 10.2. Внешний вид модифицированного компонента `TSavePictureDialog`

Поможет нам в этом внимательное изучение исходных кодов диалогов, находящихся в модуле EXTDLGS.PAS. Программисты Borland пошли по пути модернизации внешнего вида стандартных диалогов, добавив к ним справа панель для отображения внешнего вида открываемых (записываемых) картинок. Можно пойти дальше и добавить таким же образом и свои элементы управления.

Приводимый ниже пример ModifDlg — усовершенствованная программа просмотра и сохранения файлов растровой графики, к которым относятся файлы форматов JPEG и BMP. Чтобы исключить метафайлы и значки (*.wmf, *.emf, *.ico), соответствующим образом настраиваются фильтры в диалогах открытия и сохранения.

Для изменения размеров диалогового окна нужно отыскать среди входящих в его состав компонентов панель `PicturePanel` (так назвали ее разработчики Borland) и увеличить ее высоту. Следует также поменять и размеры родительских окон. Поскольку они не являются компонентами Delphi (стандартные диалоги являются составными частями Windows) для этой цели используются функции API `GetWindowRect` и `SetWindowPos`.

Обратите также внимание, что при загрузке используется событие `OnProgress` класса `TGraphic`. В его обработчике информация об объеме проделанной работы отображается на компоненте `ProgressBar1`. Для маленьких картинок обработчик вызывается только в начале и в конце операции, пользователь ничего не заметит. Зато при загрузке большого изображения он будет спокоен, видя, что процесс загрузки идет и машина не зависла.

Листинг 10.1. Исходный текст главного модуля программы ModifDlg

```
unit mainUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtDlgs, StdCtrls, ComCtrls, ExtCtrls, Buttons;
type
  TForm1 = class(TForm)
    SavePictureDialog1: TSavePictureDialog;
    OpenPictureDialog1: TOpenPictureDialog;
    ScrollBox1: TScrollBox;
    Image1: TImage;
    ProgressBar1: TProgressBar;
    OpenBitBtn: TBitBtn;
    SaveBitBtn: TBitBtn;
```

```
    procedure SavePictureDialog1TypeChange(Sender: TObject);
    procedure Image1Progress(Sender: TObject; Stage: TProgressStage;
      PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
      const Msg: String);
    procedure SavePictureDialog1Close(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SavePictureDialog1Show(Sender: TObject);
    procedure OpenBitBtnClick(Sender: TObject);
    procedure SaveBitBtnClick(Sender: TObject);
  private
  public
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
uses jpeg;
const DeltaH : Integer = 80;
var Quality : TJpegQualityRange;
    ProgressiveEnc : Boolean;

procedure TForm1.OpenBitBtnClick(Sender: TObject);
begin
if OpenPictureDialog1.Execute
 then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;

procedure TForm1.SaveBitBtnClick(Sender: TObject);
var ji : TJpegImage;
begin
if SavePictureDialog1.Execute then
 begin
 ji := TJpegImage.Create;
 ji.CompressionQuality := Quality;
 ji.ProgressiveEncoding := ProgressiveEnc;
 ji.Assign(Image1.Picture.Bitmap);
 ji.SaveToFile(SavePictureDialog1.FileName);
 ji.Free;
 end;
end;

procedure TForm1.SavePictureDialog1TypeChange(Sender: TObject);
var ParentHandle:THandle;wRect:TRect;
```

```
PicPanel,PaintPanel:TPanel;JEdit : TEdit;
Expanded : boolean;
begin
With Sender as TSavePictureDialog do
begin
 PicPanel := (FindComponent('PicturePanel') as TPanel);
 if not Assigned(PicPanel) then Exit;
 ParentHandle:=GetParent(Handle);

 PaintPanel:=(FindComponent('PaintPanel') as TPanel);
 PaintPanel.Align := alNone;
 Expanded := FindComponent('JLabel') <> nil;
 if FilterIndex >1 then
   begin
   if not Expanded then
    begin
    GetWindowRect(ParentHandle,WRect);
    SetWindowPos(ParentHandle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top+DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    GetWindowRect(Handle,WRect);
    SetWindowPos(handle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top+DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    Expanded:=True;
    PicPanel.Height := PicPanel.Height+DeltaH;

    if FindComponent('JLabel')=nil then
    with TLabel.Create(Sender as TSavePictureDialog) do
     begin
      Parent := PicPanel;
      Name := 'JLabel';
      Caption := 'Quality';
      Left := 5;
      Height := 25;
      Top := PaintPanel.Top+PaintPanel.Height+5;
     end;

    if FindComponent('JEdit')=nil then
     begin
     JEdit := TEdit.Create(Sender as TSavePictureDialog);
     with JEdit do
      begin
       Parent := PicPanel;
       Name:='JEdit';
       Text := '75';
       Left:=50;Width := 50;
       Height := 25;
```

```
       Top := PaintPanel.Top+PaintPanel.Height+5;
      end;
     end;

    if FindComponent('JUpDown')=nil then
    with TUpDown.Create(Sender as TSavePictureDialog) do
     begin
      Parent := PicPanel;
      Name:='JUpDown';
      Associate := JEdit;
      Increment := 5;
      Min := 1; Max := 100;
      Position := 75;
     end;

    if FindComponent('JCheck')=nil then
    with TCheckBox.Create(Sender as TSavePictureDialog) do
     begin
      Name:='JCheck';
      Caption:='Progressive Encoding';
      Parent:=PicPanel;
      Left:=5;Width := PicPanel.Width — 10;
      Height:=25;
      Top := PaintPanel.Top+PaintPanel.Height+35;
     end;
   end;
   end
 else
   SavePictureDialog1Close(Sender);
end;
end;

procedure TForm1.Image1Progress(Sender: TObject; Stage: TProgressStage;
  PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
  const Msg: String);
begin
case Stage of
 psStarting: begin
             Progressbar1.Position := 0;
             Progressbar1.Max := 100;
             end;
 psEnding:   begin
             Progressbar1.Position := 0;
             end;
```

```
 psRunning:  begin
             Progressbar1.Position := PercentDone;
             end;
 end;
end;

procedure TForm1.SavePictureDialog1Close(Sender: TObject);
var PicPanel : TPanel; ParentHandle : THandle; WRect : TRect;
begin
With Sender as TSavePictureDialog do
  begin
  PicPanel := (FindComponent('PicturePanel') as TPanel);
  if not Assigned(PicPanel) then Exit;
  ParentHandle:=GetParent(Handle);
  if ParentHandle=0 then Exit;
  if FindComponent('JLabel')<>nil then
   begin
    FindComponent('JLabel').Free;
    FindComponent('JEdit').Free;
    ProgressiveEnc := (FindComponent('JCheck') as TCheckBox).Checked;
    FindComponent('JCheck').Free;
    Quality := (FindComponent('JUpDown') as TUpDown).Position;
    FindComponent('JUpDown').Free;

    PicPanel.Height:=PicPanel.Height-DeltaH;
    GetWindowRect(Handle,WRect);
    SetWindowPos(Handle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top-DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    GetWindowRect(ParentHandle,WRect);
    SetWindowPos(ParentHandle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top-DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    FilterIndex := 1;
   end;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
var s: string;
begin
 s :=GraphicFilter(TBitmap)+'|'+GraphicFilter(TJpegImage);
 OpenPictureDialog1.Filter := s;
 SavePictureDialog1.Filter := s;
end;
```

```
procedure TForm1.SavePictureDialog1Show(Sender: TObject);
begin
 with Sender as TSavePictureDialog do
 begin
  if FindComponent('JLabel')<>nil then
   begin
    FilterIndex := 2;
    SavePictureDialog1TypeChange(Sender);
   end;
 end;
end;

end.
```

Приведенный пример может послужить толчком, во-первых, к углубленному изучению формата JPEG, а во-вторых, — к модификации стандартных диалогов. На его базе можно создать диалоги открытия аудиозаписей, документов и других специализированных видов файлов.

Класс *TClipboard*

Класс `TClipboard` предоставляет программисту интерфейс с буфером (папкой) обмена (Clipboard) Windows. При включении в проект модуля CLIPBRD.PAS глобальный объект `Clipboard` создается автоматически и доступен приложению в течение всего времени его работы.

Методы открытия и закрытия буфера обмена:

```
procedure Open;
procedure Close;
```

вызываются во всех остальных методах `TClipboard`, поэтому программист редко нуждается в обращении к ним. В объекте ведется счетчик числа обращений к этим функциям, так что соответствующие функции API Windows вызываются только при первом открытии и последнем закрытии.

Очистка содержимого буфера (для всех форматов) производится вызовом метода:

```
procedure Clear;
```

О доступных форматах можно узнать, пользуясь следующими свойствами и методами:

- `property FormatCount: Integer;`

  Содержит число форматов в буфере на данный момент.

- `property Formats[Index: Integer]: Word;`

  Содержит их полный список.

- `function HasFormat(Format: Word): Boolean.`

  Проверяет, содержится ли в данный момент формат `Format`.

Волею разработчиков различаются способы обмена графической и текстовой информацией через буфер обмена. Рассмотрим их независимо.

Через вызов метода

```
procedure Assign(Source: TPersistent);
```

в буфер обмена помещаются данные классов `TGraphic`, точнее, его потомков — классов `TBitmap` (формат `CF_BITMAP`) и `TMetafile` (`CF_ENHMETAFILE`), а также данные класса `TPicture`. Данные класса `TIcon` не имеют своего формата и с классом `TClipboard` несовместимы.

Допустимо и обратное: в `TClipboard` есть специальные (скрытые) методы для присваивания содержимого объектам классов `TPicture`, `TBitmap` и `TMetafile`. Допустимы выражения вида:

```
MyImage.Picture.Assign(Clipboard);
Clipboard.Assign(MyImage.Picture);
```

Для работы с текстом предназначены методы:

- `function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;`

  Читает текст из буфера обмена в буфер `Buffer`, длина которого ограничена значением `BufSize`. Функция возвращает истинную длину прочитанного текста.

- `procedure SetTextBuf(Buffer: PChar);`

  Помещает текст из `Buffer` в буфер обмена в формате `CF_TEXT`.

Впрочем, можно поступить проще. Свойство

```
property AsText: string;
```

соответствует содержимому буфера обмена в текстовом формате `CF_TEXT` (приведенному к типу `string`). При отсутствии в буфере данных этого формата возвращается пустая строка.

Методы:

```
function GetAsHandle(Format: Word): THandle;
procedure SetAsHandle(Format: Word; Value: THandle);
```

соответственно читают и пишут данные в буфер в заданном формате `Format`. При чтении возвращается дескриптор находящихся в буфере данных (или 0 при отсутствии данных). Для дальнейшего использования эти данные долж-

ны быть скопированы. При записи данные, передаваемые в параметре `Value`, в дальнейшем должны быть уничтожены системой (а не программой пользователя).

Два метода предназначены для обмена компонентами через буфер обмена (в специально зарегистрированном формате `CF_COMPONENT`):

```
function GetComponent(Owner, Parent: TComponent): TComponent;
procedure SetComponent(Component: TComponent);
```

Они используются составными частями среды Delphi.

Класс *TScreen*

Этот компонент представляет свойства дисплея (в Windows 98 и 2000 — нескольких дисплеев), на котором выполняется приложение. Поскольку экземпляр данного класса только один (он создается системой при запуске приложения), то большинство методов и свойств имеют информационный характер и недоступны для записи.

Курсор приложения, общий для всех форм, доступен через свойство

```
property Cursor: TCursor;
```

Часто приходится включать "песочные часы" на время выполнения длительной операции. Правильнее всего это сделать следующим образом:

```
Screen.Cursor := crHourglass;
try
  {Calculations...}
finally
  Screen.Cursor := crDefault;
end;
```

Имеется список всех курсоров. Получить дескриптор курсора с индексом `Index` можно при помощи свойства:

```
property Cursors[Index: Integer]: HCURSOR;
```

Напомним, что индексы зарегистрированных курсоров лежат в диапазоне от −22 (`crSizeAll`) до 0 (`crDefault`).

Рассмотренный ниже фрагмент кода при инициализации формы заносит имена всех зарегистрированных в системе курсоров в список `ListBox1`. Затем при выборе элемента списка устанавливается соответствующий ему курсор:

```
procedure TForm1.FormCreate(Sender: TObject);
type
TGetStrFunc = function(const Value: string): Integer of object;
```

```
var
  CursorNames: TStringList;
  AddValue: TGetStrFunc;
begin
  CursorNames := TStringList.Create;
  AddValue := CursorNames.Add;
  GetCursorValues(TGetStrProc(AddValue));
  ListBox1.Items.Assign(CursorNames);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
 Screen.Cursor := StringToCursor(ListBox1.Items[ListBox1.ItemIndex]);
end;
```

Список курсоров, функции `GetCursorValues`, `StringToCursor` и некоторые другие содержатся в модуле CONTROLS.PAS.

Имена всех установленных в системе шрифтов помещаются в список, определенный в свойстве

```
property Fonts: TStrings;
```

Компонент сообщает неизменяемые свойства экрана (в данном видеорежиме). Его размеры в пикселах определены в свойствах

```
property Height: Integer;
property Width: Integer;
```

В последних версиях ОС Microsoft имеется поддержка отображения на нескольких мониторах одновременно. Для этой цели предусмотрены свойства

```
property MonitorCount: Integer;
property Monitors[Index: Integer]: TMonitor;
```

Каждый компонент `TMonitor` несет информацию о размерах и положении изображения на нем. Образовавшийся же виртуальный рабочий стол характеризуется следующими свойствами:

```
property DesktopLeft: Integer;
property DesktopTop: Integer;
property DesktopWidth: Integer;
property DesktopHeight: Integer;
```

Все координаты отсчитываются от верхнего левого угла первого монитора. Если монитор один, значения этих свойств совпадают с `Left`, `Top`, `Width` и `Height`.

Примечание

С исходными текстами Delphi 5 поставляется модуль MULTIMON.PAS, содержащий прототипы структур и функций Windows 98, 2000 для работы со многими мониторами.

Число точек на дюйм дисплея содержится в свойстве

```
property PixelsPerInch: Integer;
```

При появлении каждая форма заносит себя в список форм глобального объекта `Screen`. Два (доступных только для чтения) свойства дают информацию об этом списке:

```
property Forms[Index: Integer]: TForm;
property FormCount: Integer;
```

Нужно иметь в виду, что в списке указаны только формы, открытые приложением, а не все окна системы.

Следующие два свойства указывают на активную в данный момент форму и ее активный элемент управления:

```
property ActiveControl: TWinControl;
property ActiveForm: TForm;
```

При их изменении генерируются, соответственно, события

```
property OnActiveControlChange: TNotifyEvent;
property OnActiveFormChange: TNotifyEvent;
```

Хотя и "некстати", расскажем здесь о свойстве

```
property DefaultKbLayout: HKL;
```

Оно указывает на раскладку клавиатуры, принятую в системе по умолчанию. Часто раскладку клавиатуры нужно переключать программно, чтобы облегчить жизнь пользователю. Так, в приложении, в котором надо быстро вводить в базу данных большой объем информации на русском и английском языках, такое переключение при смене полей просто необходимо.

Сначала следует прочитать список имеющихся в системе раскладок и установить нужную:

```
var  RusLayout, EngLayout : THandle;

procedure TMainForm.FormCreate(Sender: TObject);
var Layouts : array[0..7] of THandle;
i,n : Integer;
begin
// Считывание раскладок
 RusLayout := 0; EngLayout := 0;
```

```
 n := GetKeyboardLayoutList(High(Layouts)+1, Layouts);
 if n>0 then
 for i:=0 to n-1 do
  if LoWord(Layouts[i]) and $FF = LANG_RUSSIAN then RusLayout := Layouts[i]
  else if LoWord(Layouts[i]) and $FF  = LANG_ENGLISH then EngLayout :=
  Layouts[i];
 // Если есть, включим русскую
 if RusLayout<>0 then  ActivateKeyboardLayout(RusLayout,0);
end;
```

Затем при входе в определенное поле (компонент редактирования данных) и выходе из него можно программно сменить раскладку:

```
procedure TMainForm.EditDocSerEnter(Sender: TObject);
begin
 if EngLayout<>0 then ActivateKeyboardLayout(EngLayout,0);
end;

procedure TMainForm.EditDocSerExit(Sender: TObject);
begin
 if RusLayout<>0 then  ActivateKeyboardLayout(RusLayout,0);
end;
```

Вывод графики с использованием отображаемых файлов

Спору нет — объект `TBitmap` удобен и универсален. Программисты Borland шагают в ногу с разработчиками графического API Windows, и исходный код модуля GRAPHICS.PAS от версии к версии совершенствуется. Но в ряде случаев возможностей, предоставляемых стандартным компонентом, недостаточно. Один из таких случаев — работа с большими и очень большими изображениями (до сотен Мбайт). С ними приходится иметь дело в полиграфии, медицине, при обработке изображений дистанционного зондирования Земли из космоса и т. п. Здесь класс `TBitmap` не подходит, т. к. запрашивает для хранения и преобразования картинки слишком много ресурсов.

Что делать? На помощь следует призвать Windows API, поддерживающий файлы, отображаемые в память (Memory Mapped Files). У них много полезных свойств, но здесь важно только одно из них. При создании битовой карты Windows распределяет для нее часть виртуального адресного пространства. А оно не безгранично — для выделения 50—100 Мбайт может не хватить размеров файла подкачки, не говоря уже об ОЗУ. Но можно напрямую отобразить файл в виртуальную память, сделав его частью виртуального адресного пространства. В этом случае нашему файлу с изображением будет

просто выделен диапазон адресов, которые можно использовать для последующей работы.

Процедура отображения файла в память и присвоения адреса его данным выглядит следующим образом:

```
Var Memory: pByteArray;
ec : Integer;

procedure TForm1.Open1Click(Sender: TObject);
var
 i: integer;
bmFile : pBitmapFileHeader;
bmInfo : pBitmapInfoHeader;
begin
if not OpenDialog1.execute then Exit;

hf := CreateFile(pChar(OpenDialog1.FileName), GENERIC_READ or
GENERIC_WRITE,
  FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
if hf=INVALID_HANDLE_VALUE then
  begin
   ec:=GetLastError;
   ShowMessage(' File opening error '+IntTostr(ec));
   Exit;
  end;

hm := CreateFileMapping(hf, nil, PAGE_READONLY, 0,0,nil);
if hm=0 then
  begin
   ShowMessage(' File Mapping error %d',[GetLastError]);
   Exit;
  end;

pb := MapViewOfFile(hm, FILE_MAP_READ, 0,0,0);
if pb=nil then
  begin
   ec:=GetLastError;
   ShowMessage('Mapping error '+IntTostr(ec));
   Exit;
  end;

bmFile := pBitmapFileHeader(pb);
if (bmFile^.bfType<>$4D42) then BEGIN Exit; END;
Memory:=@(pb^[bmFile^.bfOffBits]);
bmInfo := @(pb^[SizeOf(TBitmapFileHeader)]);
```

```
StrLen:=(((bmInfo^.biWidth*bmInfo^.biBitCount)+31) div 32)*4;

 PaintMe(Self);

end;
```

В этом коде последовательно получены дескрипторы файла (hf, с использованием функции CreateFile), его отображения в память (hm, с помощью функции CreateFileMapping) и указатель на отображенные данные (pb, посредством MapViewOfFile). Не будем вдаваться в детали внутренней реализации битовой карты — графический формат BMP известен достаточно хорошо. Отметим только, что результатом проделанных операций являются структура bmInfo типа TBitmapInfo, полностью характеризующая битовую карту, и указатель Memory на данные битовой карты. Теперь загруженные данные нужно суметь нарисовать на канве, в данном случае на канве объекта PaintBox. Делается это следующим образом:

```
procedure TForm1.PaintMe(Sender: TObject);
 var OldP : hPalette;i : integer;
begin

if Memory=nil then Exit;
OldP := SelectPalette(PaintBox.Canvas.Handle, Palette, False);
RealizePalette(PaintBox.Canvas.Handle);
SetStretchBltMode(PaintBox.Canvas.Handle, STRETCH_DELETESCANS);

case ViewMode of
vmStretch:
with bminfo^ do
i :=
StretchDIBits(PaintBox.Canvas.Handle,0,0,PaintBox.Height,PaintBox.Width,
 0,0,biWidth,Abs(biHeight),
 Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS,
 PaintBox.Canvas.CopyMode);

vm1x1:
with bminfo^,PaintBox.ClientRect  do
i := SetDIBitsToDevice(PaintBox.Canvas.Handle,Left,Top,Right-Left,
Bottom-Top,
 Left,Top,Top,Bottom-top,
Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS);

vmZoom:
begin
  with bminfo^,PaintBox.ClientRect  do
 i := StretchDIBits(PaintBox.Canvas.Handle,Left,Top,Right-Left,
Bottom-Top,
```

```
 0,0,biWidth,Abs(biHeight),
 Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS,
 PaintBox.Canvas.CopyMode);
end;

end;
if (i=0) or (i=GDI_ERROR) then
   begin
   ec :=GetLastError;
   Form1.Caption := 'Error code '+IntToStr(ec);
   end;

SelectPalette(PaintBox.Canvas.Handle, OldP, False);
end;
```

В зависимости от установленного режима отображения (`vmStretch`, `vmZoom` или `vm1x1`) применяются разные функции Win API: `StretchDIBits` или `SetDIBitsToDevice`. Выигрыш в скорости работы приложения особенно ощущается, если загружаемые файлы становятся велики и должны размещаться в файле подкачки. Наше же приложение не использует его и отображает данные прямо из файла на экран (рис. 10.3).

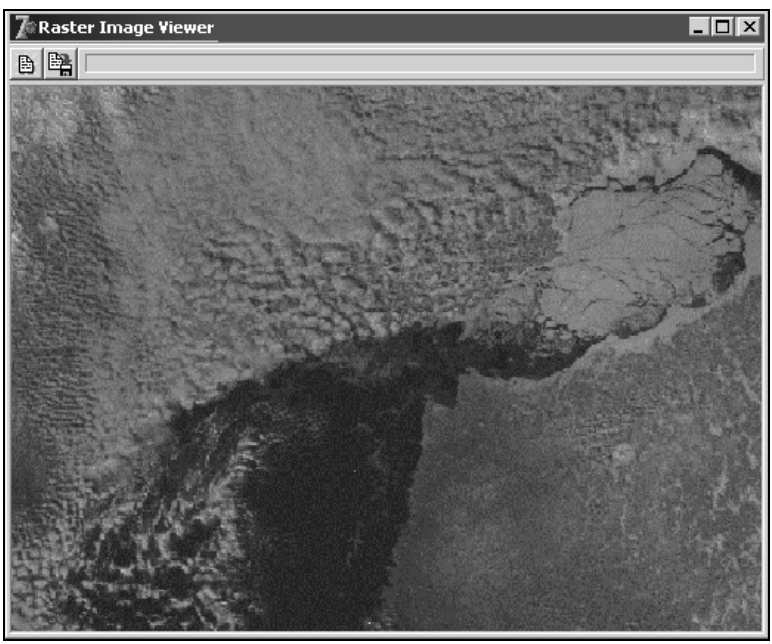


Рис. 10.3. Этот снимок с метеорологического спутника имеет размер десятки мегабайт

Класс *TAnimate*

В заключение — несколько слов для тех, кто хочет применить в своих программах анимированные (движущиеся) картинки. Самый простой путь для этого — быстрая смена нескольких последовательных битовых карт. Но, во-первых, их еще нужно нарисовать; во-вторых, если у вас достаточно большие картинки или недостаточно мощный компьютер, обязательно будет заметно мерцание, задержки и другие проблемы с выводом на экран. С появлением очередной версии библиотеки элементов управления COMCTL32 гораздо проще применить готовый компонент `TAnimate`.

Этот компонент предназначен для воспроизведения на форме файлов формата AVI (audio-video interleaved; появился впервые с выходом пакета Microsoft Video for Windows).

Альтернативными источниками таких файлов могут послужить:

- файл (с расширением avi). Его имя нужно задать в свойстве:

```
property FileName: TFileName;
```

- ресурс Windows. Он может быть задан одним из трех свойств:

```
property ResHandle: THandle;
property ResID: Integer;
property ResName: string;
```

Наконец, если вы не запаслись своим AVI-файлом, то можете воспользоваться готовым, имеющимся в Windows и иллюстрирующим один из происходящих в системе процессов. Для этого из списка свойства `CommonAVI` нужно выбрать один из вариантов (рис. 10.4).

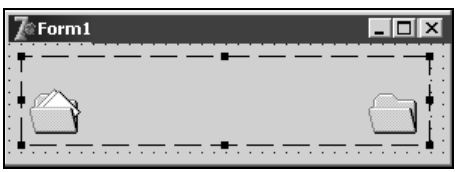


Рис. 10.4. Так выглядит ролик "перенос файлов"

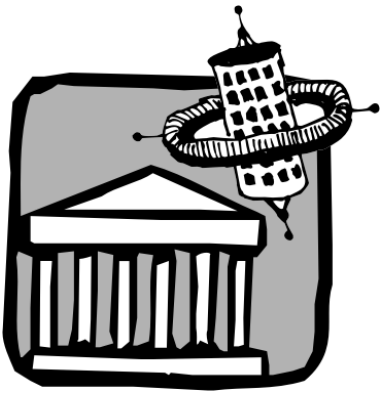
Все эти свойства при своей установке обнуляют прочие альтернативные варианты. Запуск ролика начинается при установке свойства `Active` в значение `True`; при этом показываются кадры, начиная с `StartFrame` и до `StopFrame`. Число повторений этой последовательности кадров задается свойством `Repetitions`; если вам нужен бесконечный цикл, установите это свойство в 0.

Что особенно удобно, компонент `TAnimate` снимает проблемы синхронизации показа ролика с другими процессами в системе и вашем приложении.

Если свойство `Timers` равно значению `False`, показ ролика происходит в отдельном программном потоке и никак не влияет на остальное; если оно равно значению `True`, ролик синхронизируется по внутреннему таймеру. Вы можете привязать к показу ролика, например, проигрывание звука.

Резюме

Графика — не самый маленький и не самый простой раздел программирования в Windows. Описанные в этой главе объекты Delphi сглаживают многие острые углы, но все равно начинающему программисту без синяков и шишек не обойтись. Если у вас есть время и серьезные намерения, посидите над исходным текстом модуля GRAPHICS.PAS — лучшего пособия для самообразования не найти.



ЧАСТЬ III

Приложения баз данных

Глава 11. Архитектура приложений баз данных

Глава 12. Набор данных

Глава 13. Поля и типы данных

Глава 14. Механизмы управления данными

Глава 15. Компоненты отображения данных

ГЛАВА 11

Архитектура приложений баз данных

Приложение баз данных, как следует уже из его названия, предназначено для взаимодействия с некоторым источником данных — базой данных (БД). Взаимодействие подразумевает получение данных, их представление в определенном формате для просмотра пользователем, редактирование в соответствии с реализованными в программе бизнес-алгоритмами и возврат обработанных данных обратно в базу данных.

В качестве источника данных могут выступать как собственно базы данных, так и обычные файлы — текстовые, электронные таблицы и т. д. Но здесь мы будем рассматривать приложения, работающие с базами данных.

Как известно, базы данных обслуживаются специальными программами — системами управления базами данных (СУБД), которые делятся на *локальные*, преимущественно однопользовательские, предназначенные для настольных приложений, и *серверные* — сетевые (часто удаленные), многопользовательские, функционирующие на выделенных компьютерах — серверах. Главный критерий такой классификации — объем базы данных и средняя нагрузка на СУБД.

Тем не менее, несмотря на разнообразие реализаций, общая архитектура приложения баз данных остается неизменной.

Само приложение включает механизм получения и отправки данных, механизм внутреннего представления данных в том или ином виде, пользовательский интерфейс для отображения и редактирования данных, бизнес-логику для обработки данных.

Механизм получения и отправки данных обеспечивает соединение с источником данных (часто опосредованно). Он должен "знать", куда ему обращаться и какой протокол обмена использовать для обеспечения двунаправленного потока данных.

Механизм внутреннего представления данных является ядром приложения баз данных. Он обеспечивает хранение полученных данных в приложении и предоставляет их по запросу других частей приложения.

Пользовательский интерфейс обеспечивает просмотр и редактирование данных, а также управление данными и приложением в целом.

Бизнес-логика приложения представляет собой набор реализованных в программе алгоритмов обработки данных.

Между приложением и собственно базой данных находится специальное программное обеспечение (ПО), связывающее программу и источник данных и управляющее процессом обмена данными. Это ПО может быть реализовано самыми разнообразными способами, в зависимости от объема базы данных, решаемых системой задач, числа пользователей, способами соединения приложения и базы данных. Промежуточное ПО может быть реализовано как окружение приложения, без которого оно вообще не будет работать, как набор драйверов и динамических библиотек, к которым обращается приложение, может быть интегрировано в само приложение. Наконец, это может быть отдельный удаленный сервер, обслуживающий тысячи приложений.

Источник данных представляет собой хранилище данных (саму базу данных) и СУБД, управляющую данными, обеспечивающую целостность и непротиворечивость данных.

В этой и последующих главах *части III* мы подробно остановимся на способах разработки приложений баз данных в Delphi. При разнообразии способов реализации и обилии технических деталей общая архитектура приложений баз данных в Delphi следует описанной выше общей схеме.

В Delphi 7 реализовано достаточно большое число разнообразных технологий доступа к данным (они рассматриваются далее в этой книге). Но последовательность операций при конструировании приложений баз данных остается почти одинаковой. И в работе используются по сути одни и те же компоненты, доработанные для применения с той или иной технологией доступа к данным.

В этой главе рассматриваются общие подходы к разработке приложений баз данных в Delphi, базовые классы и механизмы, которые не изменятся, выберите ли вы для вашего приложения Borland Database Engine (BDE), Microsoft ActiveX Data Objects (ADO) или dbExpress.

Примечание

Главы *части IV* построены на основе материала глав этой части. Излагаемые здесь сведения являются базовыми для понимания процесса разработки и функционирования приложений баз данных в Delphi. Поэтому в последующем материале столь часто встречаются ссылки на главы этой части.

Итак, в этой главе рассматриваются следующие вопросы:

- структура приложения баз данных в Delphi;
- базовые компоненты, используемые при разработке приложений баз данных, и их взаимосвязь;
- понятие набора данных и его участие в основных механизмах приложения баз данных;
- модуль данных;
- программная реализация частей приложения баз данных (см. рис. 11.1).

Как работает приложение баз данных

В Репозитории Delphi отсутствует отдельный шаблон для приложения баз данных. Поэтому, как и любое другое приложение Delphi, приложение баз данных начинается с обычной формы. Безусловно, это оправданный подход, т. к. приложение баз данных имеет *пользовательский интерфейс*. И этот интерфейс создается с использованием стандартных и специализированных визуальных компонентов на обычных формах.

Визуальные компоненты отображения данных расположены на странице **Data Controls** Палитры компонентов. В большинстве они представляют собой модификации стандартных элементов управления, приспособленных для работы с набором данных *(см. гл. 15)*.

Приложение может содержать произвольное число форм и использовать любой интерфейс (MDI или SDI). Обычно одна форма отвечает за выполнение группы однородных операций, объединенных общим назначением.

В основе любого приложения баз данных лежат *наборы данных*, которые представляют собой группы записей (их удобно представить в виде таблиц в памяти), переданных из базы данных в приложение для просмотра и редактирования. Каждый набор данных инкапсулирован в специальном компоненте доступа к данным. В VCL Delphi реализован набор базовых классов, поддерживающих функциональность наборов данных, и практически идентичные по составу наборы дочерних компонентов для технологий доступа к данным. Их общий предок — класс `TDataSet`. *(Подробно наборы данных рассмотрены в гл. 12.)*

Для обеспечения связи набора данных с визуальными компонентами отображения данных используется специальный компонент `TDataSource`. Его роль заключается в управлении потоками данных между набором данных и связанными с ним компонентами отображения данных. Этот компонент обеспечивает передачу данных в визуальные компоненты и возврат результатов редактирования в набор данных, отвечает за изменение состояния визуальных компонентов при изменении состояния набора данных, передает

сигналы управления от пользователя (визуальных компонентов) в набор данных. Компонент `TDataSource` расположен на странице **Data Access** Палитры компонентов.

Таким образом, базовый механизм доступа к данным создается триадой компонентов:

- компоненты, инкапсулирующие набор данных (потомки класса `TDataSet`);
- компоненты `TDataSource`;
- визуальные компоненты отображения данных.

Рассмотрим схему взаимодействия этих компонентов в приложении баз данных (рис. 11.1).

В приложении с источником данных или промежуточным программным обеспечением взаимодействует компонент доступа к данным, который инкапсулирует набор данных и обращается к функциям соответствующей технологии доступа к данным для выполнения различных операций. Компонент доступа к данным представляет собой "образ" таблицы базы данных в приложении. Общее число таких компонентов в приложении не ограничено.

С каждым компонентом доступа к данным может быть связан как минимум один компонент `TDataSource`. В его обязанности входит соединение набора данных с визуальными компонентами отображения данных. Компонент `TDataSource` обеспечивает передачу в эти компоненты текущих значений полей из набора данных и возврат в него сделанных изменений.

Еще одна функция компонента `TDataSource` заключается в синхронизации поведения компонентов отображения данных с состоянием набора данных. Например, если набор данных не активен, то компонент `TDataSource` обеспечивает удаление данных из компонентов отображения данных и их перевод в неактивное состояние. Или, если набор данных работает в режиме "только для чтения", то компонент `TDataSource` обязан передать в компоненты отображения данных запрещение на изменение данных.

С одним компонентом `TDataSource` могут быть связаны несколько визуальных компонентов отображения данных. Эти компоненты представляют собой модифицированные элементы управления, которые предназначены для показа информации из наборов данных.

При открытии набора данных компонент обеспечивает передачу в набор данных записей из требуемой таблицы БД. Курсор набора данных устанавливается на первую запись. Компонент `TDataSource` организует передачу в компоненты отображения данных значений необходимых полей из текущей записи. При перемещении по записям набора данных текущие значения полей в компонентах отображения данных автоматически обновляются.

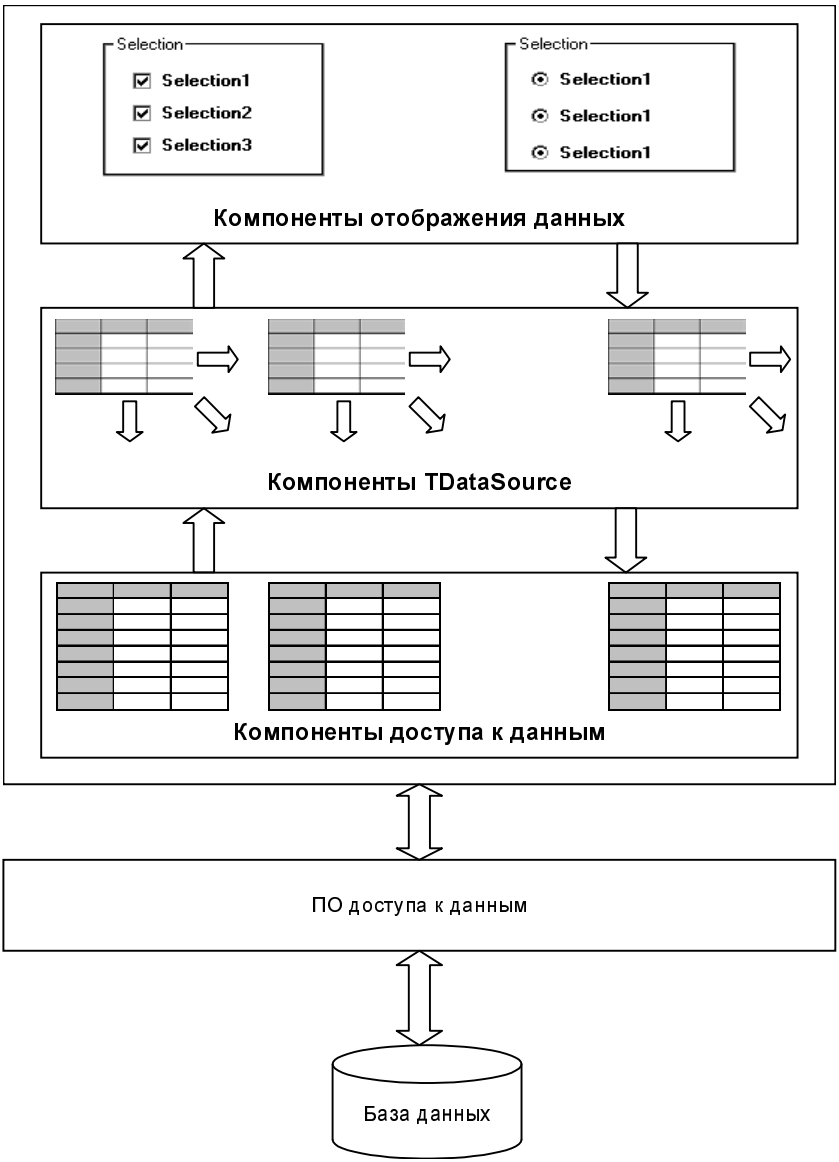


Рис. 11.1. Механизм доступа к данным приложения баз данных

Пользователь при помощи компонентов отображения данных может просматривать и редактировать данные. Измененные значения сразу же передаются из элемента управления в набор данных при помощи компонента `TDataSource`. Затем изменения могут быть переданы в базу данных или отменены.

Теперь, имея общее представление о работе приложения баз данных, перейдем к поэтапному рассмотрению процесса создания такого приложения.

Модуль данных

Для размещения компонентов доступа к данным в приложении баз данных желательно использовать специальную "форму" — модуль данных (класс `TDataModule`). Обратите внимание, что модуль данных не имеет ничего общего с обычной формой приложения, ведь его непосредственным предком является класс `TComponent`. В модуле данных можно размещать только невизуальные компоненты. Модуль данных доступен разработчику, как и любой другой модуль проекта, на этапе разработки. Пользователь приложения не может увидеть модуль данных во время выполнения.

Для создания модуля данных можно воспользоваться Репозиторием объектов или главным меню Delphi. Значок модуля данных **Data Module** расположен на странице **New**.

Как уже говорилось, модуль данных имеет мало общего со стандартной формой, хотя бы потому, что класс `TDataModule` происходит непосредственно от класса `TComponent`. У него почти полностью отсутствуют свойства и методы-обработчики событий, ведь от платформы для других невизуальных компонентов почти ничего не требуется, хотя потомки модуля данных, работающие в распределенных приложениях, выполняют весьма важную работу.

Для создания структуры (модели, диаграммы) данных, с которой работает приложение, можно воспользоваться возможностями, предоставляемыми страницей **Diagram** Редактора кода. Любой элемент из иерархического дерева компонентов модуля данных можно перенести на страницу диаграммы и задать связи между ними.

При помощи управляющих кнопок можно задавать между элементами диаграммы отношения синхронного просмотра и главный/подчиненный. При этом производится автоматическая настройка свойств соответствующих компонентов.

Для создания модуля данных (рис. 11.2) можно воспользоваться Репозиторием объектов или главным меню Delphi. Значок модуля данных **Data Module** расположен на странице **New**.

Для обращения компонентов доступа к данным, расположенным в модуле данных, из других модулей проекта необходимо включить имя модуля в секцию `uses`:

```
unit InterfaceModule;
...
implementation
```

```
uses DataModule;
...
DataModule.Table1.Open;
...
```

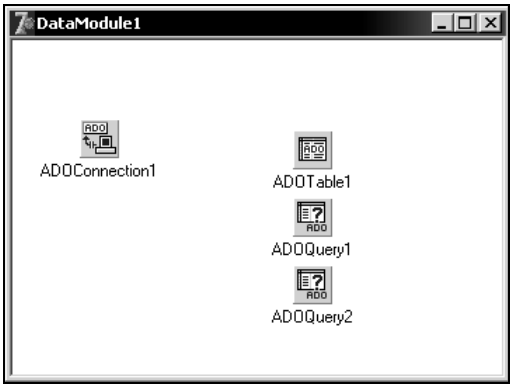


Рис. 11.2. Модуль данных

Преимуществом размещения компонентов доступа к данным в модуле данных является то, что изменение значения любого свойства проявится сразу же во всех обычных модулях, к которым подключен этот модуль данных. Кроме этого, все обработчики событий этих компонентов, т. е. вся логика работы с данными приложения, собраны в одном месте, что тоже весьма удобно.

Подключение набора данных

Компонент доступа к данным является основой приложения баз данных. На основе выбранной таблицы БД он создает набор данных и позволяет эффективно управлять им. В процессе работы такой компонент тесно взаимодействует с функциями соответствующей технологии доступа к данным. Обычно доступ к функциональности технологии доступа к данным осуществляется через совокупность интерфейсов. Все компоненты доступа к данным являются невизуальными.

Для создания нового проекта достаточно выбрать команду **New Application** из меню **File** или воспользоваться Репозиторием объектов, который открывается командой **New** из меню **File**.

Примечание

Здесь рассматривается простейший вариант создания приложения. В реальных проектах для размещения компонентов доступа к данным следует использовать модуль данных.

Затем на форму нового проекта необходимо перенести компонент, инкапсулирующий набор данных, и выполнить следующие действия. Последовательность действий рассмотрим для компонента, инкапсулирующего функции таблицы *(см. гл. 12)*.

1. *Подключить компонент к базе данных.* Для этого, в зависимости от конкретной технологии, используется или специальный компонент, устанавливающий соединение, или прямое обращение к драйверу, интерфейсу или динамической библиотеке. Детально способы соединения рассматриваются в *части IV*.

2. *Подключить к компоненту таблицу БД.* Для этого используется свойство `TableName`, доступное в Инспекторе объектов. После выполнения действий первого этапа в списке этого свойства должны появиться имена всех доступных в подключенной базе данных таблиц. После выбора имени таблицы в свойстве `TableName` компонент оказывается связанным с ней.

3. *Переименовать компонент.* Это не обязательное действие. Тем не менее, в любых случаях желательно присваивать компонентам доступа к данным осмысленные имена, соответствующие названиям подключенных таблиц. Обычно название компонента копирует название таблицы (например, `Orders` или `OrdTable` или `tblOrders`).

4. *Активизировать связь между компонентом и таблицей БД.* Для этого используется свойство `Active`. Если в Инспекторе объектов присвоить этому свойству значение `True`, то связь активизируется. Эту операцию можно выполнить и в исходном коде приложения. Также существует метод `Open`, который открывает набор данных, и метод `Close`, закрывающий его.

В качестве примера попробуем создать простейшее приложение баз данных, работающее с таблицей COUNTRY.DB из стандартной демонстрационной базы данных DBDEMOS через драйвер процессора Borland Database Engine.

На форму нового проекта необходимо перенести компонент `TTable` со страницы **BDE** Палитры компонентов. Свойство `DatabaseName` должно ссылаться на псевдоним DBDEMOS, который создается автоматически при установке Delphi, его можно выбрать из списка свойства `DatabaseName`. Для свойства `TableName` необходимо задать имя таблицы `"COUNTRY.DB"`. Его также можно выбрать из списка. Двойной щелчок на свойстве `Active` в Инспекторе объектов присваивает ему значение `True`. После этого связь компонента с таблицей активизируется. Свойство `Name` имеет значение `"CountryTable"`.

Открытие и закрытие набора данных можно предусмотреть как реакцию на действия пользователя или возникновение события. Чаще всего набор данных должен открываться при первом показе формы и закрываться при ее закрытии.

Листинг 11.1. Секция `Implementation` главного модуля проекта DemoDBApp

```
implementation

{$R *.DFM}

procedure TForm1.FormShow(Sender: TObject);
begin
 try
  CountryTable.Open;
 except
  ShowMessage('Table open error');
 end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 CountryTable.Close;
end;

end.
```

При открытии формы выполняется метод обработчик `FormShow`. В нем набор данных открывается при помощи метода `Open`. Обратите внимание на использование конструкции `try..except`, которая обеспечивает корректное завершение при возникновении исключительных ситуаций.

Так как ошибки в работе приложений баз данных могут привести к серьезным последствиям (потеря или искажение данных), то защитный код должен присутствовать во всех критических местах.

В методе-обработчике `FormClose`, который вызывается при закрытии формы, набор данных закрывается методом `Close`.

Примечание

В принципе, для выполнения рассмотренных операций можно воспользоваться и свойством `Active`. Однако реальные операции выполняют указанные методы. Поэтому использование свойства является лишним этапом, да и с точки зрения ООП все действия должны выполнять методы объекта, а свойства служат только для представления значений. Свойство `Active` сигнализирует о состоянии набора данных.

Настройка компонента *TDataSource*

На втором этапе разработки приложения баз данных необходимо перенести на форму и настроить компонент `TDataSource`. Он обеспечивает взаимодействие набора данных с компонентами отображения данных. Чаще всего

одному набору данных соответствует один компонент `TDataSource`, хотя их может быть несколько.

Для настройки свойств компонента необходимо выполнить следующие действия.

1. *Связать набор данных и компонент* `TDataSource`. Для этого используется свойство `DataSet` компонента `TDataSource`, доступное через Инспектор объектов. Это указатель на экземпляр компонента доступа к данным. В списке этого свойства в Инспекторе объектов перечислены все доступные компоненты наборов данных.
2. *Переименовать компонент.* Это не обязательное действие. Тем не менее желательно присваивать компонентам осмысленные имена, соответствующие названиям связанных наборов данных. Обычно название компонента комбинирует имя набора данных (например `OrdSource` или `dsOrders`).

В приложении DemoDBApp компонент `CountrySource` связан с компонентом `CountryTable`. Поэтому свойство `DataSet` имеет значение `CountryTable`.

Примечание

Компонент `TDataSource` можно подключить не только к набору данных из той же формы, но и любой другой, модуль которой указан в секции `uses`.

Компонент `TDataSource` имеет ряд полезных свойств и методов.

Итак, связывание с компонентом набора данных выполняет свойство

```
property DataSet: TDataSet;
```

а определить текущее состояние набора данных можно, использовав свойство

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc);
property State: TDataSetState;
```

При помощи свойства

```
property Enabled: Boolean;
```

можно включить или отключить все связанные визуальные компоненты. При значении `False` ни один связанный компонент отображения данных не будет работать.

Свойство

```
property AutoEdit: Boolean;
```

при значении `True` всегда будет переводить набор данных в режим редактирования при получении фокуса одним из связанных визуальных компонентов.

Аналогично, метод

```
procedure Edit;
```

переводит связанный набор данных в режим редактирования.

Метод

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

возвращает значение `True`, если компонент, указанный в параметре `DataSet`, действительно связан с данным компонентом `TDataSource`.

Метод-обработчик

```
type TDataChangeEvent = procedure(Sender: TObject; Field: TField)
of object;
property OnDataChange: TDataChangeEvent;
```

вызывается при редактировании данных в одном из связанных визуальных компонентов.

Метод-обработчик

```
property OnUpdateData: TNotifyEvent;
```

вызывается перед сохранением изменений в базе данных.

Метод-обработчик

```
property OnStateChange: TNotifyEvent;
```

вызывается при изменении состояния связанного набора данных *(см. гл. 12)*.

Отображение данных

На третьем этапе создания приложения баз данных необходимо разработать пользовательский интерфейс на основе компонентов отображения данных. Эти компоненты предназначены специально для решения задач просмотра и редактирования данных. Внешне большинство этих компонентов ничем не отличаются от стандартных элементов управления. Более того, многие из компонентов отображения данных являются наследниками стандартных компонентов — элементов управления.

Компоненты отображения данных должны быть связаны с компонентом `TDataSource` и через него с компонентом набора данных. Для этого используется их свойство `DataSource`. Оно присутствует во всех компонентах отображения данных.

Большинство компонентов предназначены для представления данных из одного единственного поля. В таких компонентах имеется еще одно свойство `DataField`, которое определяет поле связанного набора данных, отображаемое в компоненте.

Особое значение для приложений баз данных играет компонент `TDBGrid`, который представляет данные в виде таблицы. В столбцах таблицы размещаются поля набора данных, а в строках — записи. Для этого компонента не имеет смысла определять конкретное поле, но можно задать настраиваемый набор колонок, а для каждой из них определить поле набора данных. *(Подробнее о визуальных компонентах отображения данных см. гл. 15.)*

Таким образом, для каждого визуального компонента отображения данных необходимо выполнить следующие операции:

1. *Связать компонент отображения данных и компонент TDataSource.* Для этого используется свойство `DataSource`, которое должно указывать на экземпляр требуемого компонента `TDataSource`. Один компонент отображения данных можно связать только с одним компонентом `TDataSource`. Необходимый компонент можно выбрать в списке свойств в Инспекторе объектов.
2. *Задать поле данных.* Для этого используется свойство `DataField` типа `TFields`. В нем необходимо указать имя поля связанного набора данных. После задания свойства `DataSource` поле можно выбрать из списка. Этот этап применяется только для компонентов, отображающих единственное поле.

Отдельное место среди компонентов отображения данных занимает компонент `TDBNavigator`. Он предназначен для перемещения по записям набора данных.

В приложении DemoDBApp использованы компоненты `TDBGrid`, `TDBNavigator` и `TDBEdit` (рис. 11.3).

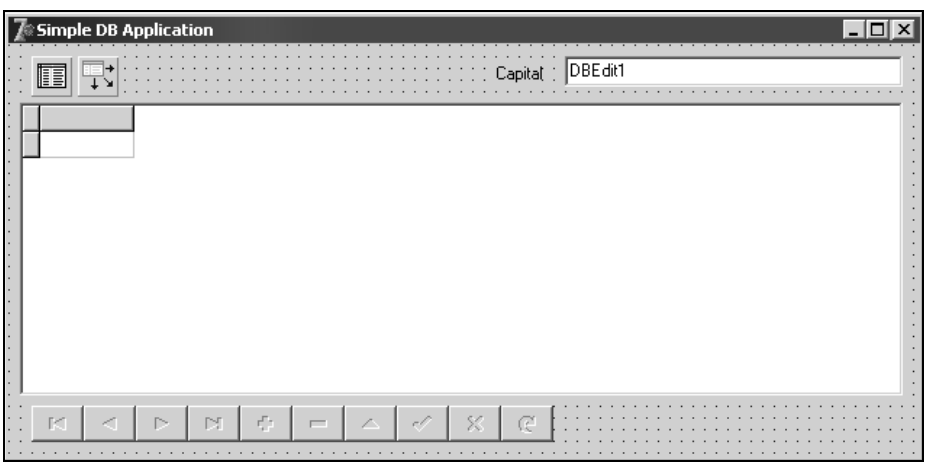


Рис. 11.3. Главная форма приложения DemoDBApp

Все три компонента отображения данных связаны с компонентом `CountrySource` типа `TDataSource` при помощи свойства `DataSource`.

Компонент `TDBEdit` отображает данные из поля `Capital` (столица государства) и позволяет редактировать их.

Компонент `TDBGrid` показывает набор данных целиком, данные в ячейках можно редактировать.

Компонент `TDBNavigator` позволяет перемещаться по записям набора данных `CountryTable`. При этом результат заметен во всех подключенных к набору данных компонентах отображения данных.

Резюме

Приложения баз данных могут получать доступ к источникам данных при помощи разнообразных технологий доступа, многие из которых используются и в приложениях Delphi. Тем не менее любое приложение баз данных в Delphi имеет стандартное ядро, структура которого определена архитектурой приложения баз данных.

Набор базовых компонентов и способов разработки является единой основой, на которой базируются технологии доступа к данным. Это позволило унифицировать процесс разработки приложений баз данных.

В основе процесса разработки лежит триада компонентов:

- невизуальные компоненты набора данных;
- невизуальные компоненты `TDataSource`;
- визуальные компоненты отображения данных.

ГЛАВА 12

Набор данных

Любое приложение баз данных должно уметь выполнять как минимум две операции. *Во-первых*, иметь информацию о местонахождении базы данных, подключаться к ней и считывать имеющуюся в таблицах БД информацию. Эта функция в значительной степени зависит от реализации конкретной технологии доступа к данным.

Во-вторых, обеспечивать представление и редактирование полученных данных. Множество записей одной или нескольких таблиц, переданные в приложение в результате активизации компонента доступа к данным, будем называть *набором данных*. Понятно, что в объектно-ориентированной среде для представления какой-либо группы записей приложение должно использовать возможности некоторого класса. Этот класс должен инкапсулировать набор данных и обладать методами для управления записями и полями.

Таким образом, сам набор данных и класс набора данных является той осью, вокруг которой вращается любая деятельность приложения баз данных.

Пользователь просматривает на экране данные — это результат использования набора данных.

Пользователь решил изменить какое-то число — он изменит содержимое ячейки набора данных.

При закрытии приложение сохраняет все изменения — это набор данных передается в базу данных для сохранения.

При этом, используя одни базовые функции для обслуживания набора данных, компоненты должны обеспечивать доступ к данным в рамках различных технологий. Поэтому не удивительно, что разработчики VCL уделили особое внимание созданию максимально эффективной иерархии классов, обеспечивающих использование наборов данных (рис. 12.1).

Класс `TDataSet` является базовым классом иерархии, он инкапсулирует абстрактный набор данных и реализует максимально общие методы работы

с ним. В него можно передать записи из таблицы базы данных или строки из обычного текстового файла — набор данных будет функционировать одинаково хорошо.

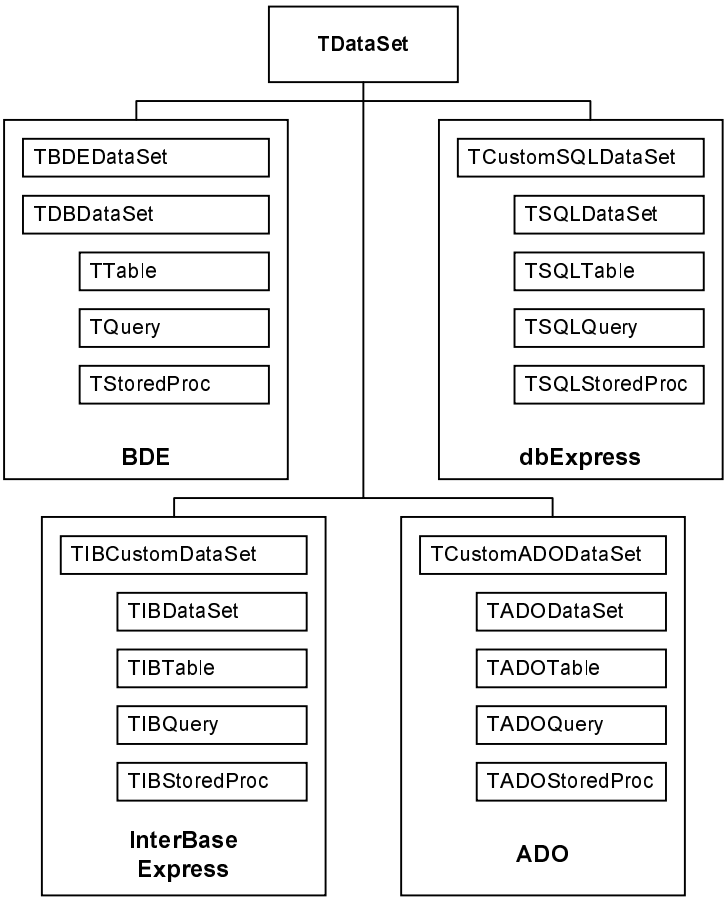


Рис. 12.1. Иерархия классов, обеспечивающих функционирование набора данных

На основе базового класса реализованы специальные компоненты VCL для различных технологий доступа к данным, которые позволяют разработчику конструировать приложения баз данных, используя одни и те же приемы и настраивая одинаковые свойства.

В этой главе рассматриваются следующие вопросы:

- набор данных, инкапсулированный в классе `TDataSet`;
- что такое состояния набора данных;

- индексы, поля, параметры;
- прототипы компонентов для работы с таблицами, запросами и хранимыми процедурами;
- основные механизмы набора данных, реализованные в классе `TDataSet`.

Абстрактный набор данных

В основе иерархии классов, обеспечивающих функционирование наборов данных в приложениях баз данных Delphi, лежит класс `TDataSet`. Хотя он почти не содержит методов, реально обеспечивающих работоспособность основных механизмов набора данных, тем не менее его значение трудно переоценить.

Этот класс задает структурную основу функционирования набора данных. Другими словами, это скелет набора данных, к методам которого необходимо лишь добавить требуемые вызовы соответствующих функций реальных технологий.

При решении наиболее распространенных задач программирования в процессе создания приложений баз данных класс `TDataSet` не нужен. Тем не менее знание основных принципов работы набора данных всегда полезно. Кроме этого, класс `TDataSet` может использоваться разработчиками в качестве основы для создания собственных компонентов. Поэтому рассмотрим основные механизмы, реализованные в наборе данных.

Набор данных открывается и закрывается свойством

```
property Active: Boolean;
```

которому соответственно необходимо присвоить значение `True` или `False`.

Аналогичные действия выполняют методы

```
procedure Open;
procedure Close;
```

После открытия набора данных можно перемещаться по его записям.

На одну запись вперед и назад перемещают курсор соответственно методы

```
procedure Next;
procedure Prior;
```

На первую и последнюю запись можно попасть, используя соответственно методы

```
procedure First;
procedure Last;
```

Признаком того, что достигнута последняя запись набора, является свойство

```
property Eof: Boolean;
```

которое в этом случае имеет значение `True`.

Аналогичную функцию для первой записи выполняет свойство

```
property Bof: Boolean;
```

Перемещение вперед и назад на заданное число записей выполняет метод

```
function MoveBy(Distance: Integer): Integer;
```

Параметр `Distance` определяет число записей. Если параметр отрицательный — перемещение осуществляется к началу набора данных, иначе — к концу.

Для ускоренного перемещения по набору данных можно отключить все связанные компоненты отображения данных. Это делается методом

```
procedure DisableControls;
```

Обратная операция выполняется методом

```
procedure EnableControls;
```

Общее число записей набора данных возвращает свойство

```
property RecordCount: Integer;
```

однако использовать его нужно аккуратно, т. к. каждое обращение к этому свойству приводит к обновлению набора данных, что может вызвать проблемы для больших таблиц или сложных запросов. Если вам нужно определить, не является ли набор данных пустым (часто используемая операция), можно использовать метод

```
function IsEmpty: Boolean;
```

который возвращает значение `True`, если набор данных пуст, или уже упоминавшиеся свойства

```
...
if MyTable.Bof and MyTable.Eof
 then ShowMessage('DataSet is empty');
...
```

Номер текущей записи позволяет узнать свойство

```
property RecNo: Integer;
```

Размер записи в байтах возвращает свойство

```
property RecordSize: Word;
```

Каждая запись набора данных представляет собой совокупность значений полей таблицы. В зависимости от типа компонента и его настройки, число полей в наборе данных может изменяться. И совсем не обязательно набор данных должен содержать все поля таблицы базы данных.

Совокупность полей набора данных инкапсулирует свойство

```
property Fields: TFields;
```

а все необходимые параметры полей содержатся в свойстве

```
property FieldDefs: TFieldDefs;
```

Общее число полей набора данных возвращает свойство

```
property FieldCount: Integer;
```

а общее число полей типа BLOB содержится в свойстве

```
property BlobFieldCount: Integer;
```

Доступ к значениям полей текущей записи предоставляет свойство

```
property FieldValues[const FieldName: string]: Variant; default;
```

где в параметре `FieldName` задается имя поля.

В процессе программирования разработчик очень часто обращается к полям набора данных. Если структура полей набора данных жестко задана и не изменяется, это можно сделать так:

```
for i := 0 to MyTable.FieldCount - 1 do
  MyTable.Fields[i].DiplayFormat := '#.###';
```

Иначе, если порядок следования полей и их состав меняется, можно использовать метод

```
function FieldByName(const FieldName: string): TField;
```

И делается это следующим образом:

```
MyTable.FieldByName('VENDORNO').AsInteger := 1234;
```

Имя поля, передаваемое в параметре `FieldName`, не чувствительно к регистру символов.

Метод

```
procedure GetFieldNames(List: TStrings);
```

вернет в параметр `List` полный список имен полей набора данных.

Более подробная информация о полях и способах работы с ними содержится в *гл. 13*.

Класс `TDataSet` содержит ряд свойств и методов, которые обеспечивают редактирование набора данных.

Но сначала бывает полезно поинтересоваться, можно ли редактировать набор данных вообще. Это можно сделать при помощи свойства

```
property CanModify: Boolean;
```

которое принимает значение `True` для редактируемых наборов.

Перед началом редактирования набор данных нужно перевести в режим редактирования, использовав метод

```
procedure Edit;
```

Для сохранения сделанных изменений применяется метод

```
procedure Post; virtual;
```

Разработчик может вызывать его самостоятельно, или же метод `Post` вызывается самим набором данных при переходе на другую запись.

При необходимости все сделанные после последнего вызова метода `Post` изменения можно отменить методом

```
procedure Cancel; virtual;
```

Новая пустая запись добавляется в конец набора данных методом

```
procedure Append;
```

Новая пустая запись добавляется на место текущей методом

```
procedure Insert;
```

а текущая запись и все нижеследующие смещаются на одну позицию вниз.

Внимание!

При использовании методов `Append` и `Insert` набор данных переходит в режим редактирования самостоятельно.

Дополнительно, у вас есть возможность добавить или вставить новую запись уже с заполненными полями. Для этого применяются методы

```
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
```

А делается это примерно так:

```
MyTable.AppendRecord([2345, 'New customer', '+7(812)4569012', 0, '']);
```

После вызова этих методов и их завершения набор данных автоматически возвращается в состояние просмотра.

Для существующей записи аналогичным образом можно заполнить все поля, использовав метод

```
procedure SetFields(const Values: array of const);
```

Текущая запись удаляется методом

```
procedure Delete;
```

При этом набор данных не выдает никаких предупреждений, а просто делает это.

Очистить содержимое всех полей текущей записи может метод

```
procedure ClearFields;
```

Обратите внимание, что поля становятся пустыми (NULL), а не сбрасываются в нулевое значение.

О том, редактировалась ли текущая запись, сообщает свойство

```
property Modified: Boolean;
```

если оно имеет значение `True`.

Набор данных можно обновить, не закрывая и не открывая его снова. Для этого применяется метод

```
procedure Refresh;
```

Однако он сработает только для таблиц и тех запросов, которые нельзя редактировать.

В каждый момент времени набор данных находится в определенном состоянии *(о состояниях см. ниже в этой главе)*. Свойство

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc, dsOpening);
property State: TDataSetState;
```

дает информацию о текущем состоянии набора.

Методы-обработчики класса `TDataSet` предоставляют разработчику широчайшие возможности по отслеживанию событий, происходящих с набором данных.

По паре методов-обработчиков (до и после события) предусмотрено для следующих событий в наборе данных:

- открытие и закрытие набора данных;
- переход в режим редактирования;
- переход в режим вставки новой записи;
- сохранение сделанных изменений;
- отмена сделанных изменений;
- перемещение по записям набора данных;
- обновление набора данных.

Обратите внимание, что помимо методов-обработчиков режима вставки существует дополнительный метод

```
property OnNewRecord: TDataSetNotifyEvent;
```

который вызывается непосредственно при вставке или добавлении записи.

Дополнительно к этому могут использоваться методы-обработчики возникающих ошибок. Они предусмотрены для ошибок удаления, редактирования и сохранения изменений.

Метод-обработчик

```
property OnCalcFields: TDataSetNotifyEvent;
```

очень важен для задания значений вычисляемых полей. Он вызывается для каждой записи, которая отображается в визуальных компонентах, связанных с набором данных каждый раз, когда необходимо перерисовать значения полей в визуальных компонентах.

Если в методе-обработчике `OnCalcFields` производятся слишком сложные вычисления, частота его вызовов может быть уменьшена за счет свойства

```
property AutoCalcFields: Boolean;
```

По умолчанию оно равно значению `True` и расчет вычисляемых полей производится при каждой перерисовке. При значении `False` метод-обработчик `OnCalcFields` вызывается только при открытии, переходе в состояние редактирования и обновлении набора данных.

Все перечисленные выше обработчики имеют одинаковый тип

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

И метод-обработчик

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

вызывается для каждой записи набора данных при свойстве `Filtered = True`. *(Подробнее об этих свойствах и методе-обработчике см. гл. 14.)*

Помимо перечисленных, класс `TDataSet` содержит еще много свойств и методов, которые обеспечивают работоспособность многих полезных в практическом программировании приложений баз данных функций. Подробно они рассмотрены в *гл. 14*.

Стандартные компоненты

Внимательный читатель заметил, что на рис. 12.1 набор компонентов для каждой из представленных технологий доступа к данным примерно одинаков. Везде есть компонент, инкапсулирующий табличные функции, компо-

нент запроса SQL и компонент хранимой процедуры. И хотя все они имеют разных ближайших предков, тем не менее, функциональность подобных компонентов в различных технологиях почти одинакова.

Поэтому имеет смысл рассмотреть общие для компонентов свойства и методы, представив, что существуют некие виртуальные общие предки для таблицы, запроса и хранимой процедуры.

Примечание

Некоторые из описываемых ниже свойств и методов присутствуют не в каждой реализации компонентов.

Компонент таблицы

Компонент таблицы обеспечивает доступ к таблице базы данных целиком, создавая набор данных, структура полей которого полностью повторяет таблицу БД. За счет этого компонент прост в настройке и обладает многими дополнительными функциями, которые обеспечивают применение табличных индексов.

Но в практике программирования работа с таблицами целиком используется не так часто. А при работе с серверами баз данных промежуточное ПО, используемых технологий доступа к данным, все равно транслирует запрос на получение табличного набора данных в простейший запрос SQL, например:

```
SELECT * FROM Orders
```

В такой ситуации применение табличных компонентов становится менее эффективным, чем использование запросов.

После соединения с источником данных *(процесс подключения для каждой технологии подробно рассматривается в части IV)* необходимо задать имя таблицы в свойстве

```
property TableName: String;
```

Иногда в свойстве `TableType` дополнительно задается тип таблицы.

Если соединение с источником данных настроено правильно, имя таблицы можно выбрать из выпадающего списка свойства `TableName`.

Преимуществом табличного компонента является использование индексов, которые ускоряют работу с таблицей. Все индексы, созданные в базе данных для таблицы, автоматически загружаются в компонент. Их параметры доступны через свойство

```
property IndexDefs: TIndexDefs;
```

Подробно класс `TIndexDefs` рассматривается ниже в этой главе.

При работе с компонентом разработчик имеет возможность управлять индексами.

Существующий индекс можно выбрать в Инспекторе объектов в списке свойств

```
property IndexName: String;
```

или использовать свойство

```
property IndexFieldNames: String;
```

в котором можно задать произвольное сочетание имен индексированных полей таблицы. Имена полей разделяются символом точкой с запятой. Таким образом, при помощи свойства `IndexFieldNames` можно создавать составные индексы.

Свойства `IndexName` и `IndexFieldNames` нельзя использовать одновременно.

Число полей, используемых в текущем индексе табличного компонента, возвращает свойство

```
property IndexFieldCount: Integer;
```

А свойство

```
property IndexFields: [Index: Integer]: TField;
```

представляет собой индексированный список полей, входящих в текущий индекс:

```
...
for i := 0 to MyTable.IndexFieldCount — 1 do
 MyTable.IndexFields[i].Enabled := False;
...
```

Для выполнения операций с таблицами и индексами целиком в табличных компонентах реализовано несколько методов.

Метод

```
procedure CreateTable;
```

создает новую таблицу в базе данных, используя заданное имя и описание полей, и индексов из свойств `TFieldDefs` и `TIndexDefs`. Если таблица с таким именем уже имеется в базе данных, то она будет уничтожена и создана заново с новой структурой и данными.

Метод

```
procedure EmptyTable;
```

удаляет из набора данных и таблицы базы данных все записи.

Метод

```
procedure DeleteTable;
```

уничтожает таблицу базы данных, связанную с компонентом. Набор данных должен быть закрыт.

Метод

```
type
  TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive,
  ixExpression, ixNonMaintained);
  TIndexOptions = set of TIndexOption;
procedure AddIndex(const Name, Fields: String; Options: TIndexOptions,
const DescFields: String='');
```

добавляет к таблице БД новый индекс. Параметр `Name` задает имя индекса. В параметре `Fields` через точку с запятой определяются имена полей, входящих в индекс. Параметр `DescFields` задает описание индекса из констант, объявленных в типе `TIndexOption`.

Метод

```
procedure DeleteIndex(const Name: string);
```

уничтожает индекс.

Кроме этого, табличные компоненты содержат свойства и методы, описываемые в *гл. 14*.

Компонент запроса

Компонент запроса предназначен для создания запроса SQL, подготовки его параметров, передачи запроса на сервер БД и представления результата запроса в наборе данных. При этом набор данных может быть редактируемым или нет.

Любой компонент запроса, каждая строка набора данных которого однозначно связывается с одной строкой таблицы БД, может редактироваться. Например, запрос

```
SELECT * FROM Country
```

редактировать можно. Если же приведенное правило не выполняется, то набор данных можно использовать только для просмотра, и, конечно, возможности компонентов здесь ни при чем. Куда, к примеру, записывать результаты редактирования записей следующего запроса:

```
SELECT CustNo, SUM(AmountPaid)
FROM Orders
GROUP BY CustNo
```

Ведь в таком запросе каждая запись есть результат суммирования неизвестного заранее числа других записей.

Тем не менее компоненты запросов предоставляют разработчику мощный и гибкий механизм работы с данными. С помощью компонентов запросов можно решать гораздо более сложные задачи, чем с табличными компонентами.

В целом компонент запроса работает быстрее, т. к. структура возвращаемых запросом полей может изменяться, то экземпляры класса `TFieldDef`, хранящие информацию о свойствах полей, создаются по необходимости при запуске приложения. Табличный компонент создает все классы для описания полей в любом случае, поэтому в приложениях клиент-сервер табличный компонент может открываться медленнее, чем запрос.

Рассмотрим общие свойства и методы компонентов запросов.

Текст запроса определяется свойством

```
property SQL: TStrings;
```

В свойстве

```
property Text: PChar;
```

содержится окончательно подготовленный текст запроса перед пересылкой его на сервер.

Выполнение запроса возможно тремя способами.

Если запрос возвращает результат в набор данных, то применяется метод

```
procedure Open;
```

или свойство

```
property Active: Boolean;
```

которому присваивается значение `True`. После выполнения запроса открывается набор данных компонента. Закрывается такой запрос методом

```
procedure Close;
```

или тем же свойством `Active`.

Если запрос не возвращает результат в набор данных (например, использует операторы `INSERT`, `DELETE`, `UPDATE`), то используется метод

```
procedure ExecSQL;
```

и после выполнения запроса набор данных компонента не открывается. Попытка использовать для такого запроса метод `Open` или свойство `Active` приведет к ошибке создания указателя на курсор данных.

После выполнения запроса в свойстве

```
property RowsAffected: Integer;
```

возвращается число обработанных при выполнении запроса записей.

Для того чтобы разрешить редактирование набора данных запроса, необходимо свойству

```
property RequestLive: Boolean;
```

присвоить значение `True`. Это свойство устанавливается, но не работает для запроса, результат которого не модифицируется из-за самого запроса.

Для подготовки запроса к выполнению предназначен метод

```
procedure Prepare;
```

который обеспечивает выделение необходимых ресурсов на сервере и проведение оптимизации.

Метод

```
procedure UnPrepare;
```

освобождает занятые при подготовке запроса ресурсы.

Результат выполнения этих двух операций отражается в свойстве

```
property Prepared: Boolean;
```

Значение `True` данного свойства говорит о том, что запрос подготовлен для выполнения.

Вызов методов `Prepare` и `UnPrepare` не является обязательным, т. к. компонент делает это автоматически. Однако если запрос будет выполняться несколько раз подряд, то подготовку необходимо провести перед первым выполнением запроса вручную. Тогда при последующих выполнениях сервер не будет тратить время на проведение бесполезной операции — ведь ресурсы под запрос уже были выделены.

Часто запросы имеют настраиваемые параметры, значения которых определяются непосредственно перед выполнением запроса.

Свойство

```
property Params: TParams;
```

представляет собой список объектов `TParams`, каждый из которых содержит настройки одного параметра. Свойство `Params` обновляется автоматически при изменении текста запроса. Подробнее о классе `TParams` рассказывается ниже в этой главе.

Примечание

В компоненте `TADOQuery` свойство, аналогичное описанному свойству `Params`, называется `Parameters`.

Свойство

```
property ParamCount: Word;
```

возвращает число параметров запроса.

Свойство

```
property ParamCheck: Boolean;
```

определяет, необходимо ли обновлять свойство `Params` при изменении текста запроса во время выполнения. При значении `True` обновление осуществляется.

Кроме этого, компоненты запросов содержат некоторые свойства и методы, описываемые в *гл. 14*.

Компонент хранимой процедуры

Компонент хранимой процедуры предназначен для определения процедуры, установки ее параметров, выполнения процедуры и возвращения результатов в компонент.

В зависимости от выбранной технологии доступа к данным, каждый компонент хранимой процедуры имеет собственный способ соединения с сервером. После подключения к источнику данных имя хранимой процедуры можно выбрать из списка свойства

```
property StoredProcName: String;
```

После этого свойство

```
property Params: TParams;
```

предназначенное для хранения параметров процедуры, автоматически заполняется.

Для хранимых процедур важно деление параметров на входные и выходные. Первые содержат исходные данные, а вторые передают результаты выполнения процедуры.

Детально класс `TParams` описывается ниже.

Общее число параметров возвращает свойство

```
property ParamCount: Word;
```

Для подготовки хранимой процедуры используется метод

```
procedure Prepare;
```

или свойство

```
property Prepared: Boolean;
```

которое должно получить значение `True`.

Метод

```
procedure UnPrepare;
```

или свойство `Prepared := False` выполняют обратное действие.

Кроме того, проверка значения свойства `Prepared` позволяет установить, осуществлялась ли подготовка процедуры к выполнению или нет.

Внимание!

После выполнения хранимой процедуры исходный порядок следования параметров в списке `Params` может измениться. Поэтому для доступа к конкретному параметру рекомендуется использовать метод

```
function ParamByName(const Value: String): TParam;
```

Если хранимая процедура возвращает набор данных, компонент можно открывать методом

```
procedure Open;
```

или свойством

```
property Active: Boolean;
```

В противном случае для выполнения процедуры используется метод

```
procedure ExecProc;
```

и после этого выходные параметры получат вычисленные значения.

Индексы в наборе данных

Важнейшей проблемой для любой БД является достижение максимальной производительности и ее сохранение при дальнейшем увеличении объемов хранимых данных. Использование индексов позволяет решить эту задачу.

Индекс представляет собой часть базы данных, в которой содержится информация об организации данных в таблицах БД.

В отличие от ключей, которые просто идентифицируют отдельные записи, индексы занимают дополнительные объемы памяти (довольно значительные) и могут храниться как совместно с таблицами, так и в виде отдельных файлов. Индексы создаются вместе со своей таблицей и обновляются при модификации данных. При этом работа по обновлению индекса для большой таблицы может отнимать много ресурсов, поэтому имеет смысл ограничить число индексов для таких таблиц, где происходит частое обновление данных.

Индекс содержит в себе уникальные идентификаторы записей и дополнительную информацию об организации данных. Поэтому если при выполнении запроса сервер или локальная СУБД обращается для отбора записи к индексу, то это занимает значительно меньше времени, т. к. понятно, что идентификатор гораздо меньше самой записи. Кроме этого, индекс "знает", как организованы данные и может ускорять обработку за счет группирования записей по сходным значениям параметров.

Создание для БД эффективного набора индексов является нетривиальной задачей.

Во-первых, нужно верно определить оптимальное число индексов для каждой таблицы. Во-вторых, каждый индекс должен содержать только необходимые поля, при этом большую роль играет их упорядочивание.

В большинстве СУБД при создании индексов требуется только задать поля и название индекса, вся остальная работа выполняется автоматически.

Естественно, что в компонентах доступа к данным VCL Delphi используются все возможности такого мощного инструмента, как индексы. Причем свойства и методы для работы с индексами присутствуют только в табличных компонентах, т. к. в компонентах запросов работа с индексами осуществляется средствами SQL.

Набор данных может работать и без применения индексов, но для этого соответствующая таблица БД не должна иметь первичного ключа — случай довольно редкий. Поэтому по умолчанию в наборе данных используется первичный индекс. При открытии набора данных все записи отсортированы в соответствии с первичным ключом.

Механизм подключения индексов

Для того чтобы подключить к набору данных вторичный индекс, необходимо присвоить свойству `IndexName` название индекса. Если свойство не имеет значения, то в наборе данных используется первичный индекс.

Альтернативный способ задания индекса заключается в использовании свойства `IndexFieldNames`, в котором задается перечень имен полей необходимого индекса, разделенных точкой с запятой. При этом в Инспекторе объектов для данного свойства список полей существующих индексов создается автоматически, разработчику остается сделать выбор. При помощи свойства `IndexFieldNames` можно создавать и составные индексы. Для этого необходимо, чтобы все входящие в список поля были индексированы.

Список имен всех индексов можно получить при помощи метода `GetIndexNames`.

Примечание

Изменение текущего индекса можно осуществлять без отключения набора данных, поэтому в приложениях очень удобно делать сортировку данных по индексам. Такой метод смены индексов называется индексацией "на лету".

После установки индекса количество полей в индексе передается в свойство `IndexFieldCount`.

Список описаний индексов

Информация об индексах набора данных содержится в свойстве класса TDataSet

```
property IndexDefs: TIndexDefs;
```

В нем для каждого индекса создается структура TIndexDef. Доступ к информации об индексах осуществляется через свойство

```
property Items[Index: Integer]: TIndexDef; default;
```

являющееся списком объектов TIndexDef.

Объекты типа TIndexDef можно добавлять в список при помощи метода

```
function AddIndexDef: TIndexDef;
```

Поиск объекта описания индекса осуществляет метод

```
function Find(const Name: String): TIndexDef;
```

который возвращает найденный объект по заданному в параметре Name имени индекса.

Пара методов

```
function FindIndexForFields(const Fields: string): TIndexDef;
function GetIndexForFields(const Fields: String;
 CaseInsensitive: Boolean): TIndexDef;
```

находит объект описания индекса по списку полей, входящих в индекс. Если индекс не найден, ищется первый индекс, начинающийся с указанных полей. Первый из этих двух методов в случае неудачного поиска генерирует исключительную ситуацию EDatabaseError, а второй возвращает nil.

Список IndexDefs обновляется автоматически при открытии набора данных. Но метод

```
procedure Update; reintroduce;
```

обновляет список описаний индексов без открытия набора данных.

Описание индекса

Параметры каждого индекса набора данных представлены в классе TIndexDef, а их совокупность для набора данных содержится в свойстве IndexDefs класса TDataSet.

Свойство

```
property Name: String;
```

определяет название индекса.

Список всех полей индекса содержится в свойстве

```
property Fields: String;
```

Поля разделяются точкой с запятой.

Свойство

```
property CaseInsFields: String;
```

содержит список полей, регистр символов в которых при сортировке не учитывается. Поля разделяются точкой с запятой. Все поля из этого списка должны входить в свойство `Fields`. В наборе данных по умолчанию используется сортировка записей с учетом регистра символов. Но некоторые серверы БД допускают комбинированную сортировку по полям с учетом регистра и без.

Свойство

```
property DescFields: String;
```

содержит список полей через точку с запятой, которые сортируются в обратном порядке. Все поля из этого списка должны входить в свойство `Fields`. По умолчанию все поля сортируются в прямом порядке. Некоторые серверы БД поддерживают одновременную сортировку полей в прямом и обратном порядке.

Свойство

```
property GroupingLevel: Integer;
```

позволяет ограничить область применения индекса. Если значение этого свойства равно нулю, индекс упорядочивает все записи набора данных. В противном случае действие индекса распространяется на группы записей, имеющих одинаковые значения для того числа полей, которое задано этим свойством.

Параметры индекса определяются свойством

```
property Options: TIndexOptions;
```

Для индекса возможны сочетания следующих параметров:

- `ixPrimary` — первичный индекс;
- `ixUnique` — значения индекса уникальны;
- `ixDescending` — индекс сортирует записи в обратном порядке;
- `ixCaseInsensitive` — индекс сортирует записи без учета регистра символов;
- `ixExpression` — в индексе используется выражение (для индексов dBASE);
- `ixNonMaintained` — индекс не обновляется при открытии таблицы.

Метод

```
procedure Assign(ASource: TPersistent); override;
```

заполняет свойства объекта значениями аналогичных свойств объекта `ASource`.

Использование описаний индексов

Описания индексов наряду с описаниями полей *(см. гл. 13)* также используются при создании новых таблиц БД. Для каждого планируемого индекса перед вызовом метода `CreateTable` необходимо создать или скопировать из существующего набора данных соответствующее описание. Тогда при создании таблицы индексы будут добавлены автоматически:

```
with Table1 do
begin
 DatabaseName := 'DBDEMOS';
 TableType := ttParadox;
 TableName := 'DemoTable';
...
{Создание описаний полей}
...
 with IndexDefs do
 begin
  Clear;
  AddIndexDef;
  with Items[0] do
  begin
   Name := '';
   Fields := 'Field1';
   Options := [ixPrimary, ixUnique];
  end;
  AddIndexDef;
  with Items[1] do
  begin
   Name := 'SecondIndex';
   Fields := 'Field1;Field2';
   Options := [ixCaseInsensitive];
  end;
 end;
 CreateTable;
end;
```

При создании описаний индексов использован метод `AddIndexDef`, который при каждом вызове добавляет к списку `Items` объекта `TIndexDefs` новый

объект `TIndexDef`. Таким образом сначала создается первичный индекс (в таблицах Paradox он не имеет имени), затем вторичный индекс `SecondIndex`. Для каждого описания обязательно определяются составляющие индекс поля и параметры индекса (свойства `Fields` и `Options`).

Параметры запросов и хранимых процедур

Свойство `Params` представляет собой набор изменяемых параметров запроса или хранимой процедуры, а также набор объектов `TParam`, инкапсулирующих отдельные параметры.

Рассмотрим следующий запрос SQL:

```
SELECT SaleDat, OrderNo
FROM Orders
WHERE SaleDat >= '01.08.2001' AND SaleDat <= '31.08.2001'
```

В нем осуществляется отбор номеров заказов, сделанных в августе 2001 года. Теперь вполне естественно было бы предположить, что пользователю может понадобиться получить подобный отчет за другой месяц или за первые десять дней августа.

В этом случае можно поступить так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 with Query1 do
 begin
  SQL[0] := 'SELECT PartDat, ItemNo, ItemCount, InputPrice';
  SQL[1] := 'FROM Parts';
  SQL[2] := 'WHERE PartDat>= ''01.08.2001'' AND PartDat<=''31.08.2001''';
 end;
end;

procedure TForm1.RunBtnClick(Sender: TObject);
begin
 with Query1 do
 begin
  if Active then Close;
  SQL[2] := 'WHERE PartDat>= '+chr(39)+Date1Edit.Text+chr(39)+
  ' AND PartDat<='+chr(39)+Date2Edit.Text+chr(39);
  Open;
 end;
end;
```

При создании формы в методе `FormCreate` задается текст запроса. Для этого используется свойство `SQL`. При щелчке на кнопке `RunBtn`, в соответствии

с заданными в однострочных редакторах Date1Edit и Date2Edit датах, изменяется текст запроса. Метод FormCreate приведен только для того, чтобы обозначить первоначальный текст запроса, этот текст вполне можно задать в свойстве SQL.

Для решения подобных задач как раз и используются параметры. В этом случае текст запроса будет выглядеть следующим образом:

```
SELECT PartDat, ItemNo, ItemCount, InputPrice
FROM Parts
WHERE PartDat>= :PD1 AND PartDat<= :PD2
```

Двоеточие перед именами PD1 и PD2 означает, что это параметры. Имя параметра выбирается произвольно. В списке свойства Params первым идет тот параметр, который расположен первым по тексту запросу.

После ввода в свойстве SQL текста запроса для каждого параметра автоматически создается объект TParam. Эти объекты доступны в специализированном редакторе, который вызывается при щелчке на кнопке свойства Params в Инспекторе объектов (рис. 12.2). Для каждого параметра требуется установить тип данных, который должен согласовываться с типом данных соответствующего поля.

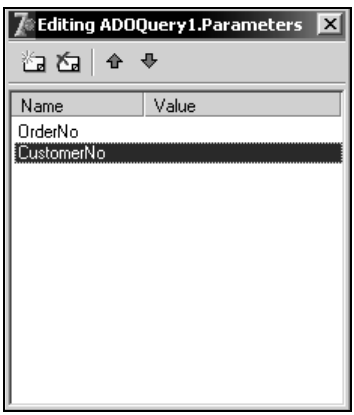


Рис. 12.2. Специализированный редактор параметров запроса

Теперь для задания текущих ограничений по дате поступления можно использовать свойство Params:

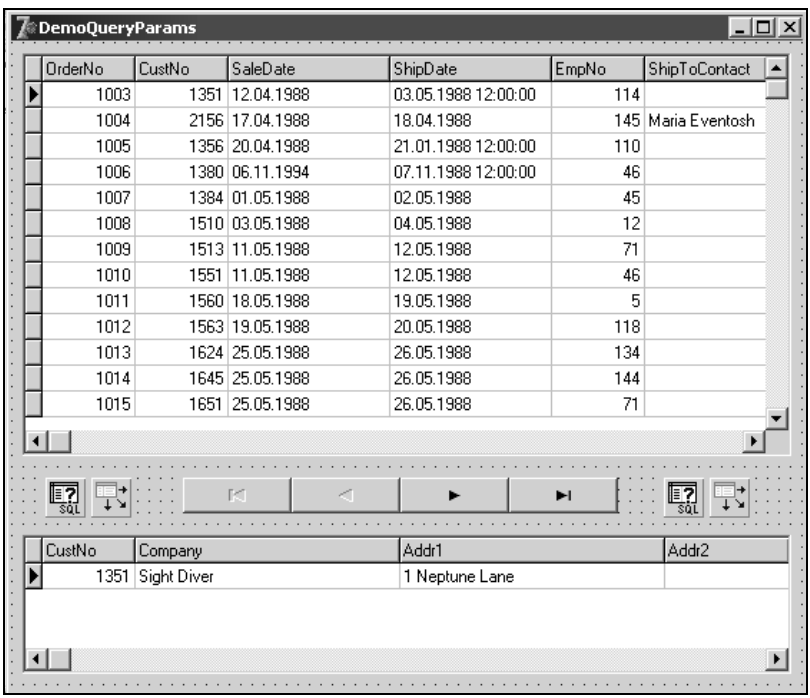
```
procedure TForm1.RunBtnClick(Sender: TObject);
begin
 with Query1 do
 begin
  Close;
```

```
  Params[0].AsDateTime := StrToDate(Date1Edit.Text);
  Params[1].AsDateTime := StrToDate(Date2Edit.Text);
  Open;
 end;
end;
```

При щелчке на кнопке `RunBtn` при помощи параметров в запрос передаются текущие значения ограничений дат.

Значения параметров запроса можно задать и из другого набора данных. Для этого применяется свойство `DataSource` компонента набора данных. Указанный в свойстве компонент `TDataSource` должен быть связан с набором данных, значения полей которого требуется передать в параметры. Названия параметров должны соответствовать названиям полей этого набора данных, тогда свойство `DataSource` начнет работать. При перемещении по записям набора данных текущие значения одноименных параметров полей автоматически передаются в запрос.

Для иллюстрации работы этого свойства рассмотрим простой пример, главная форма которого представлена на рис. 12.3. Этот проект не имеет ни одной строки написанного вручную программного кода.



OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
1003	1351	12.04.1988	03.05.1988 12:00:00	114	
1004	2156	17.04.1988	18.04.1988	145	Maria Eventosh
1005	1356	20.04.1988	21.01.1988 12:00:00	110	
1006	1380	06.11.1994	07.11.1988 12:00:00	46	
1007	1384	01.05.1988	02.05.1988	45	
1008	1510	03.05.1988	04.05.1988	12	
1009	1513	11.05.1988	12.05.1988	71	
1010	1551	11.05.1988	12.05.1988	46	
1011	1560	18.05.1988	19.05.1988	5	
1012	1563	19.05.1988	20.05.1988	118	
1013	1624	25.05.1988	26.05.1988	134	
1014	1645	25.05.1988	26.05.1988	144	
1015	1651	25.05.1988	26.05.1988	71	

CustNo	Company	Addr1	Addr2
1351	Sight Diver	1 Neptune Lane	

Рис. 12.3. Главная форма проекта DemoQueryParams

Верхний компонент TDBGrid отображает данные из таблицы Orders базы данных DBDEMOS и связан через компонент OrdersSource типа TDataSource с компонентом запроса OrdersQuery. Текст запроса выглядит так:

```
SELECT * FROM Orders
```

Нижний компонент TDBGrid отображает данные о покупателе и через компонент CustSource типа TDataSource связан с запросом CustSource, свойство SQL которого имеет следующий вид:

```
SELECT * FROM Customer
WHERE CustNo=:CustNo
```

Обратите внимание, что название параметра соответствует названию поля номера покупателя из таблицы Orders.

Свойство DataSource компонента CustQuery указывает на компонент OrdersSource.

Как только оба набора данных открываются, текущее значение из поля CustNo набора данных Orders автоматически передается в параметр запроса компонента CustQuery.

Благодаря этому, для двух наборов данных реализована связь "один-к-одному" по полю номера поставщика.

И в завершение разговора о параметрах запросов рассмотрим свойства и методы класса TParams, составляющего свойство Params, и класса TParam, инкапсулирующего отдельный параметр.

Класс *TParams*

Класс TParams представляет собой список параметров.

Доступ к элементам списка возможен через индексированное свойство

```
property Items[Index: Word]: TParam;
```

а к значениям параметров — через свойство

```
property ParamValues[const ParamName: String]: Variant;
```

Добавить новый параметр можно методом

```
procedure AddParam(Value: TParam);
```

Но для него необходимо создать объект параметра. Это можно сделать методом

```
function CreateParam(FldType: TFieldType; const ParamName: string;
ParamType: TParamType): TParam;
```

где FldType — тип данных параметра, ParamName — имя параметра и ParamType — тип параметра *(см. ниже)*.

И оба метода можно использовать в связке:

```
MyParams.AddParam(MyParams.CreateParam(ftInteger, 'Param1', ptInput));
```

Вместо того, чтобы заполнять параметры по одному, можно использовать метод

```
function ParseSQL(SQL: String; DoCreate: Boolean): String;
```

который при `DoCreate = True` анализирует текст запроса из свойства `SQL` и создает новый список параметров.

Или же, для присвоения значений сразу всем параметрам используется метод

```
procedure AssignValues(Value: TParams);
```

Для удаления параметра из списка применяется метод

```
procedure RemoveParam(Value: TParam);
```

При работе с параметрами для их идентификации полезно использовать обращение по имени, т. к. при работе с хранимыми процедурами после их выполнения порядок следования может измениться. Также и при использовании динамических запросов (их текст SQL может изменяться во время выполнения).

Для обращения к параметру по имени используется метод

```
function ParamByName(const Value: String): TParam;
```

В сложных запросах SQL или после многочисленных исправлений разработчик может допустить ошибку и создать два разных параметра с одним именем. В этом случае при выполнении запроса одноименные параметры считаются одним и им присваиваются значение первого по порядку запроса. Для контроля повторных имен в списке параметра используется метод

```
function IsEqual(Value: TParams): Boolean;
```

который возвращает значение `True`, если для параметра `Value` найден дубликат.

Класс *TParam*

Класс `TParam` инкапсулирует свойства отдельного параметра.

Имя параметра определяется свойством

```
property Name: String;
```

Тип данных параметра задает свойство

```
property DataType: TFieldType;
```

Тип данных параметра и связанного поля должны совпадать.

Тип параметра определяется множеством

```
type
  TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult);
  TParamTypes = set of TParamType;
```

которое имеет следующие значения:

- `ptUnknown` — тип неизвестен;
- `ptInput` — параметр предназначен для передачи значения из приложения;
- `ptOutput` — параметр предназначен для передачи значения в приложение;
- `ptInputOutput` — параметр предназначен для передачи и приема значения;
- `ptResult` — параметр предназначен для передачи в приложения информации о статусе операции.

Свойство

```
property ParamType: TParamType;
```

определяет тип параметра.

При работе с параметрами довольно часто бывает необходимо определить, имеет ли параметр ненулевое значение. Для этого используется свойство

```
property IsNull: Boolean;
```

Свойство возвращает значение `True`, если параметр не имеет значения или имеет значение `Null`.

Свойство

```
property Bound: Boolean;
```

возвращает значение `True` только тогда, когда параметру не присваивалось значение вообще.

Метод

```
procedure Clear;
```

присваивает параметру значение `Null`.

Само значение параметра задается свойством

```
property Value: Variant;
```

Но использование вариантов не очень эффективно, когда требуется обеспечить максимальную скорость. В таких случаях можно обратиться к целому набору свойств `As...`, которые не только возвращают значение, но и приводят его к некоторому типу. Например, свойство

```
property AsInteger: LongInt;
```

возвращает целочисленное значение поля.

Примечание

Необходимо осторожно использовать свойства с приведением типа, т. к. попытка преобразования неверного значения вызовет исключительную ситуацию.

Для чтения из буфера и записи в буфер значения параметра соответственно используются методы

```
procedure SetData(Buffer: Pointer);
procedure GetData(Buffer: Pointer);
```

а необходимый размер при записи в буфер позволит определить метод

```
function GetDataSize: Integer;
```

Можно скопировать тип данных, имя и значение параметра прямо из поля данных. Для этого применяется метод

```
procedure AssignField(Field: TField);
```

а для присвоения типа данных и значения используется метод

```
procedure AssignFieldValue(Field: TField; const Value: Variant);
```

Общее число знаков для числовых значений определяет свойство

```
property Precision: Integer;
```

А свойство

```
property NumericScale: Integer;
```

задает число знаков после запятой.

Для строковых параметров размер задает свойство

```
property Size: Integer;
```

Состояния набора данных

В процессе своего функционирования (от открытия методом `Open` и до закрытия методом `Close`) набор данных может выполнять самые разнообразные операции. Можно просто перемещаться по записям, можно редактировать данные и удалять записи, можно проводить поиск по различным параметрам и т. д. При этом желательно, чтобы все операции выполнялись как можно быстрее и эффективнее.

Набор данных в любой момент времени находится в некотором состоянии, т. е. подготовлен к выполнению действий строго определенного рода. И для каждой группы операций набор данных выполняет ряд подготовительных действий.

Все состояния набора данных делятся на две группы.

- К первой группе относятся состояния, в которые набор данных переходит автоматически, а также непродолжительные по времени состояния, сопровождающие функционирование полей набора данных (табл. 12.1).
- Во вторую группу входят состояния, которыми можно управлять из приложения, например, перевод набора данных в режим редактирования (табл. 12.2).

Базовый класс `TDataSet`, инкапсулирующий свойства набора данных, позволяет изменять состояние, а также проверять текущее состояние набора данных.

Текущее состояние набора данных передается в свойство `State`, имеющее тип `TDataSetState`:

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc);
```

Для управления состояниями набора данных используются методы `Open`, `Close`, `Edit`, `Insert`.

Таблица 12.1. *Автоматические состояния набора данных*

Константа состояния	Описание
`dsNewValue`	Включается при обращении к свойству `NewValue` поля набора данных
`dsOldValue`	Включается при обращении к свойству `OldValue` поля набора данных
`dsCurValue`	Включается при обращении к свойству `CurValue` поля набора данных
`dsInternalCalc`	Включается при расчете значений полей, для которых `FindKind = fkInternalCalc`
`dsCalcFields`	Включается при выполнении метода `OnCalcFields`
`dsBlockRead`	Включается механизм ускоренного перемещения по набору данных
`dsOpening`	Существует при открытии набора данных методом `Open` или свойством `Active`
`dsFilter`	Включается при выполнении метода `OnFilterRecord`

Таблица 12.2. *Управляемые состояния набора данных*

Константа состояния	Метод	Описание
`dsInactive`	`Close`	Набор данных закрыт
`dsBrowse`	`Open`	Данные доступны для просмотра, но недоступны для редактирования
`dsEdit`	`Edit`	Данные можно редактировать
`dsInsert`	`Insert`	К набору данных можно добавлять новые записи
`dsSetKey`	`SetKey`	Включается механизм поиска по ключу. Также могут использоваться диапазоны

Рассмотрим, как изменяется состояние набора данных при выполнении стандартных операций.

Закрытый набор данных всегда имеет неактивное состояние `dsInactive`.

При открытии набор данных переходит в состояние просмотра данных `dsBrowse`. В этом состоянии по записям набора данных можно перемещаться и просматривать их содержимое, но редактировать данные нельзя. Это основное состояние открытого набора данных, из него можно перейти в другие состояния, но любое изменение состояния происходит через просмотр данных (рис. 12.4).

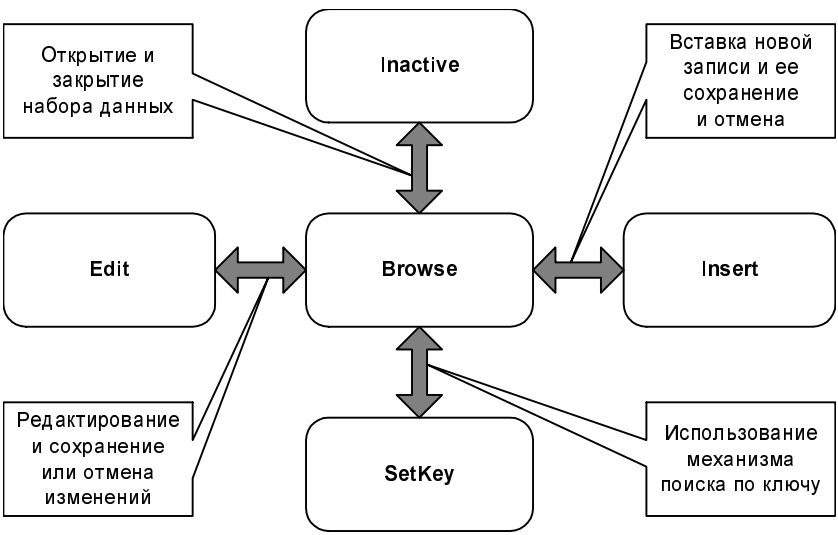


Рис. 12.4. Схема изменения состояний набора данных

При необходимости редактирования данных набор должен быть переведен в состояние редактирования dsEdit, для этого используется метод Edit. После выполнения метода можно изменять значения полей для текущей записи. При перемещении на следующую запись набор данных автоматически переходит в состояние просмотра.

Для того чтобы вставить в набор данных новую запись, необходимо использовать состояние вставки dsInsert. Метод Insert переводит набор данных в это состояние и добавляет на месте текущего курсора новую пустую запись. При переходе на другую запись, после проверки на уникальность первичного ключа (если он есть) набор данных возвращается в состояние просмотра.

Состояние установки ключа dsSetKey используется только в табличных компонентах при необходимости поиска методами FindKey и FindNext, а также при использовании диапазонов (метод SetRange). Это состояние сохраняется до момента вызова одного из методов поиска по ключу или метода отмены диапазона. После этого набор данных возвращается в состояние просмотра.

Состояние просмотра по блокам dsBlockRead используется набором данных при реализации быстрого перемещения по большим массивам записей без показа промежуточных записей в компонентах отображения данных и без вызова обработчика события перемещения по записям. Для реализации быстрого перемещения по набору данных можно использовать методы DisableControls и EnableControls.

Резюме

Набор данных является образом таблицы базы данных в приложении. Он содержит группу записей и обеспечивает их использование.

Класс TDataSet, инкапсулирующий функциональность набора данных, является базовым классом для всех технологий доступа к данным. На его основе созданы все основные компоненты, применяемые при разработке приложений баз данных. Условно их можно разделить на три группы:

- компоненты таблиц;
- компоненты запросов;
- компоненты хранимых процедур.

В этой главе рассмотрены важнейшие свойства, методы и структуры, реализованные в компонентах, инкапсулирующих набор данных.

ГЛАВА 13

Поля и типы данных

Каждая таблица БД и, следовательно, каждый набор данных приложения имеет собственную структуру, которая определяется совокупностью полей. Каждое *поле* набора данных представляет собой объект, содержащий описание типа данных, которому должно соответствовать значение, находящееся в записи на определенном месте. Иначе, полем можно назвать совокупность ячеек с данными конкретного типа, расположенных в одном и том же месте каждой записи набора данных, или попросту — это столбец в таблице.

В наборе данных приложения баз данных Delphi каждому полю соответствует собственный объект. Основой объектов полей является класс `TField`, который инкапсулирует основные свойства абстрактного поля, не зависящего от типа данных. От этого базового класса порождены другие класса, обеспечивающие функционирование реальных объектов полей, зависящих от типа данных.

Программист, грамотно использующий возможности полей, может решать существенно более сложные задачи и создавать эффективные и гибкие приложения баз данных.

Эта глава посвящена изучению объектов полей набора данных и приемов работы с ними. В ней рассматриваются следующие вопросы:

- объект поля в наборе данных;
- динамические и статические объекты полей;
- способы использования объектов полей в наборе данных;
- класс `TField` — основа использования полей в наборах данных;
- типы объектов полей и типы данных;
- ограничения.

Объекты полей

Объекты полей инкапсулируют свойства и методы полей различных типов данных. Они функционируют совместно с набором данных и очень тесно связаны с ним. Например, для того чтобы получить значения полей из текущей записи набора данных, разработчик должен создать примерно такой код:

```
Edit1.Text := Table1.Fields[0].AsString;
```

Свойство `Fields` представляет собой индексированный список объектов полей набора данных *(см. гл. 12)*. Если разработчик не изменяет порядок следования полей в наборе данных, то расположение объектов полей в списке `Fields` соответствует структуре таблицы базы данных.

Каждый объект полей хранит ряд параметров, определяющих поле. Например, в наборе данных к объекту поля можно обратиться, зная только название поля:

```
Edit1.Text := Table1.FieldByName('SomeField').AsString;
```

Для того чтобы присвоить значение полю в текущей записи, можно воспользоваться приведенными выше способами или, если тип данных поля неизвестен, свойством `FieldValues`:

```
Table1.FieldValues['SomeField'] := Edit1.Text;
```

Знание имени поля дает самый простой способ обращения к текущему значению поля:

```
Table1['SomeField'] := Edit1.Text;
Edit1.Text := Table1['SomeField'];
```

Примечание

При присваивании значений полям набора данных необходимо контролировать состояние, в котором находится набор данных *(см. гл. 12)*.

В основе классов, описывающих иерархию типизированных полей, лежит класс `TField`. От него порождены другие классы, обеспечивающие работу целых групп полей, объединенных по типам данных.

Что же такое объект поля и какие возможности он предоставляет разработчику?

Во-первых, назначение класса `TField`, как базового класса поля, заключается в умении взаимодействовать с компонентом отображения данных для обеспечения правильной визуализации данных. Например, объект поля хранит способ выравнивания, параметры шрифта, текст заголовка и т. д.

Во-вторых, с точки зрения набора данных объект поля является хранилищем текущего значения этого поля (а не всего столбца данных, как это можно себе представить по названию).

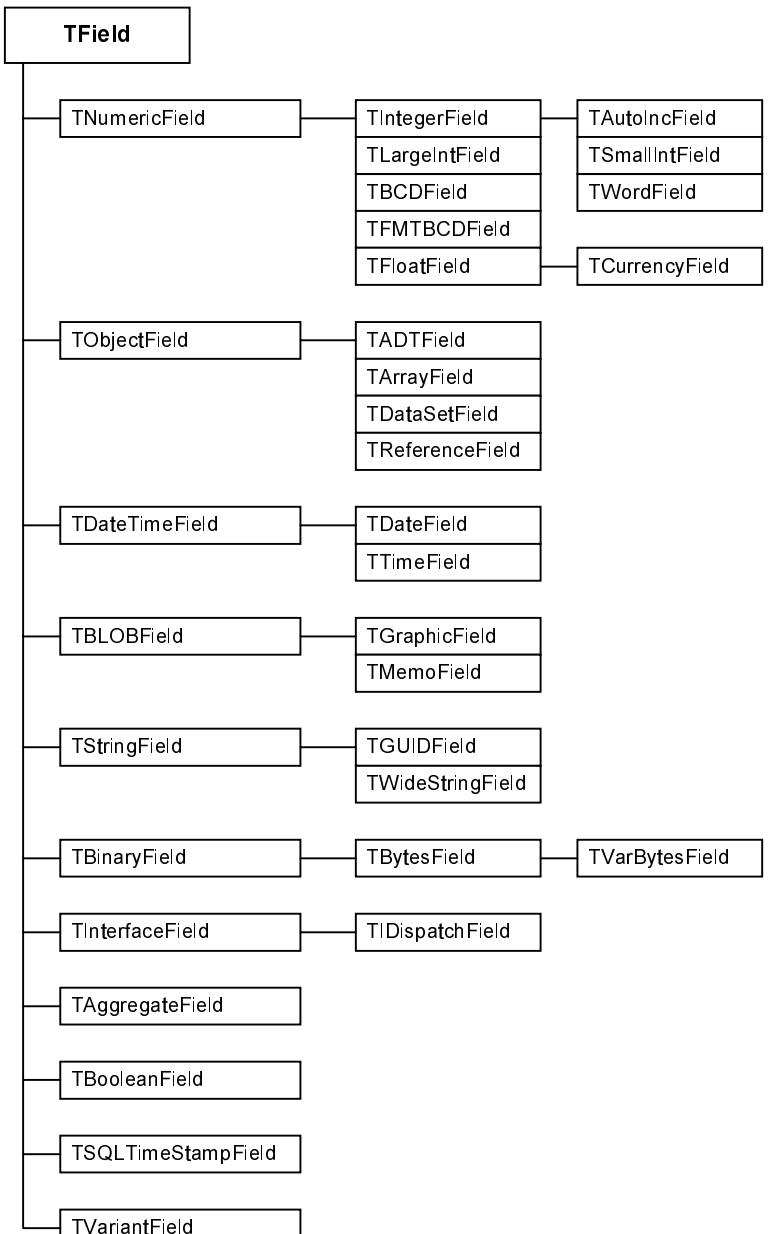


Рис. 13.1. Иерархия классов полей

Компоненты отображения данных при работе с набором данных взаимодействуют именно с полями. Например, колонки компонента `TDBGrid` при отсутствии дополнительных настроек соответствуют расположению объектов полей в связанном наборе данных.

Статические и динамические поля

В Delphi предусмотрено два способа создания объектов полей.

Динамические поля используются программой в случае, если разработчик не создал для них объекты явным образом на этапе разработки. Каждый не заданный явно объект поля автоматически создается при открытии набора данных в соответствии со структурой связанной таблицы БД. Любой объект поля является прямым наследником класса `TField`, а его конкретный тип зависит от типа данных поля таблицы. При этом свойства динамического поля устанавливаются в соответствии с параметрами поля таблицы базы данных.

Компонент набора данных после подключения к таблице БД без дополнительных настроек использует только динамические поля. К свойствам и методам динамических полей можно обратиться программно, для этого следует использовать индексированное свойство `Fields` компонента доступа к данным, которое объединяет все поля набора данных *(см. выше)* или метод `FieldByName`.

Динамические поля используются в случаях, когда заданные характеристики полей в таблице базы данных полностью удовлетворяют разработчика и нет необходимости рассматривать какое-либо поле вне набора данных.

Статические поля создаются программистом на этапе разработки, их свойства доступны в Инспекторе объектов, а их названия можно выбрать из списка объектов активной формы в верхней части Инспектора объектов. Название статического объекта поля обычно складывается из названий таблицы и поля, например `OrdersCUSTNO`.

Создаются статические объекты полей при помощи специализированного Редактора полей, который вызывается двойным щелчком на компоненте набора данных на форме или командой **Fields Editor** из всплывающего меню этого компонента.

Редактор полей представляет собой простой список уже созданных статических полей. Все управление осуществляется командами из всплывающего меню. В верхней части окна Редактора расположены кнопки навигатора для перемещения по набору данных, которые активны только при открытом наборе данных. Если набор данных имеет агрегатные поля данных *(см. ниже)*, то они размещаются в отдельном списке в нижней части окна Редактора полей (рис. 13.2).

Рис. 13.2. Редактор полей с отдельным списком агрегатных полей

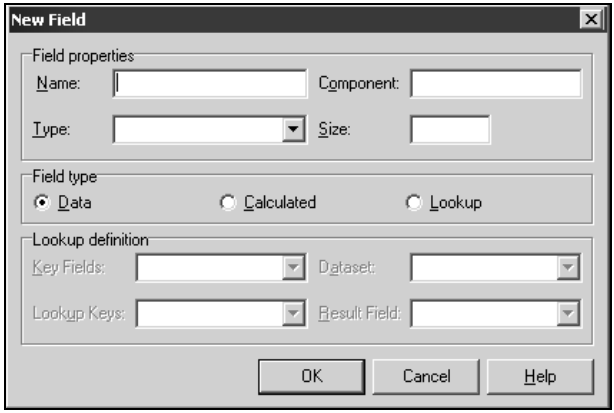


Рис. 13.3. Диалог создания нового статического поля Редактора полей набора данных

Добавить к списку статических полей новое поле, существующее в таблице БД, можно при помощи команды **Add fields** из всплывающего меню Редактора. Удаление элемента из списка осуществляется клавишей <Delete>. Перетаскиванием элементов списка при помощи мыши можно изменить их расположение. Таким образом можно создавать произвольные комбинации статических полей.

Примечание

Как только для набора данных создан хотя бы один статический объект поля, считается, что набор данных содержит только те поля, которые имеются в списке статических полей. Эту особенность можно использовать для искусственного ограничения структуры данных таблиц. Все поля набора данных расположены в том порядке, как это указано в списке Редактора полей, независимо от их реального положения в таблице базы данных.

Команда **New field** из всплывающего меню Редактора полей позволяет создать статическое поле, которое реально не существует в структуре данных таблицы (рис. 13.3). Для выбора типа поля используется группа радиокнопок **Field Type**:

- **Data** — поле данных;
- **Calculated** — вычисляемое поле;
- **Lookup** — поле синхронного просмотра.

Реже создаются поля данных, которые обязательно должны базироваться на реальных полях таблицы. Если для такого объекта соответствующее поле в таблице будет удалено или его тип данных будет изменен, то при открытии набора данных генерируется исключительная ситуация.

Примечание

Для клиентских наборов данных многоуровневых приложений диалог создания нового поля позволяет выбрать два дополнительных типа поля — это агрегатные поля (радиокнопка **Aggregate**) и внутренние вычисляемые поля (радиокнопка **InternalCalc**).

Класс *TField*

Как уже говорилось выше, в большой иерархии классов для полей различных типов данных класс `TField` является базовым (см. рис. 13.1), он инкапсулирует свойства и методы абстрактного поля данных. Именно от него происходят все классы типизированных полей. В реальной работе класс `TField` не используется, но его значение трудно переоценить. Практически все основные свойства классов типизированных полей унаследованы от класса `TField` без каких-либо изменений, а дополнительные свойства и методы обеспечивают работу конкретного типа данных.

Что касается методов-обработчиков событий, то четыре метода, определенные в классе `TField`, наследуются всеми потомками без изменения и дополнения.

Ниже приведены свойства и методы класса `TField`.

Имя объекта содержит свойство

```
property Name: TComponentName;
```

При создании объекта поля на этапе разработки имя объекта складывается из имени соответствующего компонента набора данных и имени поля.

Свойство

```
property FieldName: String;
```

возвращает имя поля таблицы базы данных.

Свойство

```
property FullName: string;
```

используется, если текущее поле является дочерним для другого поля. В этом случае свойство содержит имена всех родительских полей.

Название поля в таблице базы данных содержится в свойстве

```
property Origin: String;
```

Свойство

```
property FieldNo: Integer;
```

возвращает исходный порядковый номер поля в наборе данных. Если объекты полей являются статическими, их фактический порядок может быть изменен в Редакторе полей.

Свойство

```
property Index: Integer;
```

содержит индекс объекта поля в списке `Fields`.

Функциональное назначение поля определяется свойством

```
type TFieldKind = (fkData, fkCalculated, fkLookup, fkInternalCalc,
fkAggregate);
property FieldKind: TFieldKind;
```

В большинстве случаев его значение определяется автоматически в момент создания объекта поля. Да и впоследствии вряд ли возникнет необходимость сделать реальное поле данных вычисляемым. Обычно попытка изменить значение свойства `FieldKind` вызывает ошибку. Рассмотрим возможные значения этого свойства:

- `fkData` — поле данных;
- `fkCalculated` — вычисляемое поле;
- `fkLookup` — поле синхронного просмотра;
- `fkInternalCalc` — внутреннее вычисляемое поле;
- `fkAggregate` — агрегатное поле.

Если поле является вычисляемым, свойство

```
property Calculated: Boolean;
```

принимает значение `True`.

На связанный набор данных указывает свойство

```
property DataSet: TDataSet;
```

которое при создании объекта средствами среды разработки заполняется автоматически.

Свойство

```
property DataType: TFieldType;
```

возвращает тип данных поля, а свойство

```
property DataSize: Integer;
```

содержит объем памяти, необходимый для хранения значения поля.

Одной из важнейших задач класса `TField` является обеспечение доступа к текущему значению поля. В этом случае класс взаимодействует с буфером текущей записи набора данных, а значение можно получить при помощи нескольких свойств.

Свойство

```
property Value: Variant
```

всегда содержит значение, которое сохранено после последнего выполнения метода `Post` набора данных:

```
with Table1 do
begin
 Open;
 while Not EOF do
 begin
  if Fields[0].Value > 10
  then Fields[1].Value := Fields[1].Value*2;
  Next;
 end;
 Close;
end;
```

В этом примере при помощи метода `Next` осуществляется перебор всех записей набора данных. Если значение первого поля больше 10, то значение второго поля удваивается. Для этого применяется свойство `Value` объектов полей набора данных.

Однако из-за использования вариантов свойство `Value` является относительно медленным. И для преобразования текущего значения поля к необходимому виду можно применять целую группу быстрых свойств `As...`, которые содержат значение в определенном типе данных. Чаще всего используется свойство `AsString`, например, оно может применяться для представления числовых значений полей в элементах управления:

```
Edit1.Text := Table1.Fields[0].AsString;
```

Примечание

При работе со статическими объектами полей при передаче значений желательно использовать свойства из группы `As...`, т. к. неявное задание типа свойством `Value` может привести к ошибке преобразования данных типа `Variant`.

Если свойство

```
property CanModify: Boolean;
```

имеет значение `False`, значение поля нельзя редактировать. Однако это свойство является только средством для определения возможности редактирования.

Свойство

```
property ReadOnly: Boolean;
```

позволяет запретить редактирование (`ReadOnly := True`) или разрешить его (`ReadOnly := False`).

Большая группа свойств отвечает за представление и форматирование значения поля.

Свойство

```
property DisplayText: String;
```

содержит значение поля в строковом формате до начала редактирования.

Свойство

```
property Text: String;
```

предназначено для использования компонентами отображения данных при редактировании. Поэтому эти два свойства могут иметь разные значения в случае, если значение поля в строковом формате при редактировании и просмотре различно. У классов-наследников `TField` для этого достаточно задать шаблон отображения данных для поля (свойство `DisplayFormat`) и шаблон редактирования данных (свойство `EditFormat`). Например, вещественное число при просмотре может иметь разделители тысяч, а при редактировании нет. В этом случае рассматриваемые свойства будут иметь следующий вид:

```
DisplayText = '1 452,32'
Text = '1452,32'
```

Свойства `Text` и `DisplayText` влияют на использование метода-обработчика `OnGetText`. Если параметр `DisplayText` имеет значение `True`, то параметр `Text` содержит значение свойства `DisplayText`, в противном случае в метод передается значение поля в строковом формате.

Если поле не имеет значения, то при помощи свойства `DefaultExpression` можно задать некоторое постоянное значение, которое будет появляться в компоненте отображения данных при пустом поле. Если постоянное значение содержит какие-либо символы кроме цифр, то все выражение нужно обязательно брать в кавычки.

В случае возникновения исключительных ситуаций во время использования поля генерируется соответствующее сообщение, в котором в качестве имени поля применяется значение свойства `DisplayName`. Если задано свойство `DisplayLabel`, то `DisplayName` приравнивается к нему, в противном случае для задания свойства `DisplayName` используется свойство `FieldName`. Другим способом задать значение свойства `DisplayName` невозможно.

Свойство

```
property DisplayWidth: Integer;
```

определяет число символов для отображения значений поля в визуальных компонентах отображения данных.

Свойство

```
property Visible: Boolean;
```

отвечает за видимость поля в визуальных компонентах отображения данных. При этом компоненты, отображающие одно поле, перестают показывать его значения, а компоненты типа `TDBGrid` не отображают колонки, связанные с полем.

Примечание

Еще несколько групп свойств класса `TField`, а также его методы-обработчики рассматриваются ниже в этой главе.

Виды полей

Теперь рассмотрим классификацию полей набора данных в зависимости от их функционального назначения. Самыми распространенными полями являются поля данных, базирующиеся на реальных полях таблицы БД. Свойства объектов таких полей устанавливаются в соответствии с параметрами полей таблицы БД.

Кроме этого, в практике программирования часто применяются *поля синхронного просмотра* и *вычисляемые поля*.

Процесс создания всех типов полей набора данных практически не отличается *(см. выше)*. Тем не менее такое разнообразие позволяет успешно решать самые сложные задачи программирования приложений БД.

Ниже мы рассмотрим только поля синхронного просмотра и вычисляемые поля, т. к. поля данных не содержат каких-либо существенных особенностей в применении.

С точки зрения набора данных большой разницы между этими двумя видами полей нет. Однако значения для всех полей синхронного просмотра рассчитываются раньше, чем для вычисляемых полей. Поэтому вы можете использовать поля синхронного просмотра в выражениях вычисляемых полей и не можете сделать наоборот.

Поля синхронного просмотра

При создании для исходного набора данных нового поля синхронного просмотра необходимо использовать перечисленные ниже свойства.

- Свойство

  `property LookupDataSet: TDataSet;`

  задает набор данных синхронного просмотра.

- Свойство

  `property LookupResultField: String;`

  представляет поле синхронного просмотра из набора данных `LookupDataSet`, данные из которого будут появляться в созданном поле.

- Свойство

  `property LookupKeyFields: String;`

  содержит поле (или поля) из набора данных синхронного просмотра, по значению которого выбирается значение из поля `LookupResultField`.

- Свойство

  `property KeyFields: String;`

  определяет поле (или поля) из исходного набора данных, для которого создается поле синхронного просмотра.

Для просмотра данных из поля синхронного просмотра можно использовать любые компоненты отображения данных, но естественно будет применить специальные компоненты синхронного просмотра, о которых рассказывается в *гл. 15*.

Кроме этого, очень удобно использовать поля синхронного просмотра в компоненте `TDBGrid`. Если такое поле связать с одной из колонок компонента, то для него автоматически заполняется список синхронного просмотра. Его элементы хранятся в свойстве `PickList`, которое имеется в любой колонке. Теперь пользователю достаточно выбрать нужную колонку в сетке и, щелкнув на появившейся в текущей ячейке кнопке, получить возможные значения для замены. Одновременно с изменением поля синхронного просмотра изменяется и ключевое поле (свойство `KeyFields`) исходного набора данных.

Примечание

При использовании полей синхронного просмотра в компоненте `TDBGrid` открывать набор данных синхронного просмотра необязательно. При этом свойство `LookupCache`, о котором речь пойдет ниже, обязательно должно иметь значение `False`.

Для идентификации полей синхронного просмотра можно использовать булевское свойство `Lookup` базового класса `TField`, которое принимает истинное значение для таких полей.

Свойство

```
property LookupCache: Boolean;
```

определяет режим использования специального буфера значений синхронного просмотра. Если это свойство истинно, то буфер работает.

Буфер основан на свойстве `LookupList`. При открытии исходного набора данных каждое поле синхронного просмотра получает свое значение, одновременно с этим, в соответствии со всеми имеющимися в исходном наборе данных значениями ключевого поля, заполняется и буфер синхронного просмотра. Впоследствии, при перемещении на другую запись, значение синхронного просмотра берется не из набора данных синхронного просмотра, а из буфера. Этот механизм при небольших объемах значений синхронного просмотра позволяет увеличить скорость работы с исходным набором данных, особенно в режиме удаленного доступа при низкоскоростных сетях.

При изменениях в наборе данных синхронного просмотра можно использовать метод

```
procedure RefreshLookupList;
```

который обновляет текущее значение поля и список значений в буфере.

Специально для разработчиков в базовый класс `TField` включено свойство

```
property Offset: Integer;
```

которое возвращает размер буфера в байтах.

Для создания поля синхронного просмотра удобнее всего воспользоваться Редактором полей компонента доступа к данным. После выбора команды **New field** из всплывающего меню в одноименном диалоге (см. рис. 13.3), помимо обычных действий, соответствующих созданию поля данных, необходимо задать значения свойств в группе **Lookup definition**. Элементы управления группы становятся доступны после выбора типа поля (радиокнопка **Lookup** в группе **Field type**). Группа **Lookup definition** включает следующие элементы:

- в списке **Dataset** представлены все доступные в модуле наборы данных, из которых нужно выбрать набор данных синхронного просмотра (свойство `Lookup DataSet`);
- список **Result Field** позволяет выбрать поле синхронного просмотра (свойство `LookupResultField`);
- список **Lookup Keys** задает ключевое поле в наборе данных синхронного просмотра (свойство `LookupKeyFields`);
- список **Key Fields** определяет ключевое поле исходного набора данных (свойство `KeyFields`).

Вычисляемые поля

Вычисляемые поля существенно облегчают разработку приложений баз данных, т. к. позволяют получать новые данные на основе существующих, не изменяя при этом структуру таблиц БД. Выражения для получения значений вычисляемых полей разработчик должен разместить в методе-обработчике `OnCalcFields` набора данных. Здесь можно использовать любые арифметические, логические операции и функции, любые операторы языка, свойства и методы любых компонентов, в том числе запросы SQL:

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
 with Table1 do
 Table1CalcField1.Value := Fields[0].Value + Fields[1].Value;
 with Query1 do
 begin
  Params[0].AsInteger := Table1.Fields[0].AsInteger;
  Open;
  Table1CalcField1.Value := Fields[0].AsString;
  Close;
 end;
end;
```

Метод `OnCalcFields` выполняется при открытии набора данных, при переходе в режим редактирования, при передаче фокуса между компонентами отображения данных или колонок сетки, при удалении записи. Но для этого нужно, чтобы свойство `AutoCalcFields` набора данных было равно значению `True`.

Примечание

Необходимо учитывать, что сложные вычисляемые поля могут существенно замедлить работу набора данных (особенно при использовании при этом запросов SQL). Кроме того, в процессе редактирования набора данных (при изменении значения поля, сохранении изменений и переходе на следующую запись) вычисляемые поля рассчитываются несколько раз подряд. Для уменьшения числа автоматических обращений к методу `OnCalcFields` нужно использовать свойство `AutoCalcFields := False`.

Для создания вычисляемого поля достаточно в диалоге создания нового поля Редактора полей в качестве типа поля задать "вычисляемое", в остальном процесс совпадает с созданием поля данных.

В выражениях вычисляемых полей можно использовать другие вычисляемые поля, но они обязательно должны быть определены в методе `OnCalcFields` до этого.

Вычисляемые поля нельзя использовать при фильтрации набора данных при помощи метода-обработчика `OnFilterRecord`, т. к. он вызывается до метода-обработчика `OnCalcFields`, а вычисляемые поля не сохраняются.

Внутренние вычисляемые поля

Помимо простых вычисляемых полей существуют *внутренние вычисляемые поля* (`FieldKind = fkInternalCalc`). Они используются в клиентских наборах данных (компоненты `TClientDataSet`) и отличаются тем, что их значения сохраняются в наборе данных.

Внутренние вычисляемые поля могут быть использованы для фильтрации методом-обработчиком `OnFilterRecord`.

Агрегатные поля

Агрегатные поля предназначены для выполнения вычислительных операций со значениями полей набора данных с использованием агрегатных функций SQL. К таким функциям относятся:

- `AVG` — вычисляет среднее значение;
- `COUNT` — возвращает число записей;
- `MIN` — вычисляет минимальное значение;
- `MAX` — вычисляет максимальное значение;
- `SUM` — вычисляет сумму.

Агрегатные поля не входят в структуру полей набора данных, т. к. агрегатные функции подразумевают объединение записей таблицы для получения результата. Следовательно, значение агрегатного поля нельзя связать с какой-то одной записью, оно относится ко всем или группе записей.

Использование агрегатных полей возможно только в компоненте `TClientDataSet` и его аналогах, т. к. он обеспечивает кэширование данных, необходимое для проведения вычислений *(см. гл. 22)*.

Агрегатные поля не отображаются вместе со всеми полями в компонентах `TDBGrid`, в Редакторе полей они расположены в отдельном списке, а их свойство `Index` *(см. выше)* всегда имеет значение −1. Для представления значения агрегатного поля можно воспользоваться одним из компонентов отображения данных, который визуализирует значение одного поля (например, `TDBText` или `TDBEdit`), или свойствами самого поля:

```
Label1.Caption := MyDataSetAGGRFIELD1.AsString;
```

Для создания агрегатного поля необходимо использовать команду **New field** из всплывающего меню Редактора полей.

Для представления агрегатных полей имеется специальный класс `TAggregateField`.

Его свойство

```
property Expression: string;
```

задает вычисляемое выражение.

В его состав могут входить агрегатные функции, имена полей набора данных и простейшие арифметические операции:

```
SUM(Pirce*ItemCount) — SUM(Balance)
```

Вычисление значения проводится только для тех агрегатных полей, свойство

```
property Active: Boolean;
```

которых имеет значение `True`.

Вычисление включенных свойством `Active` агрегатных полей выполняется только в том случае, если булевское свойство `AggregatesActive` клиентского компонента набора данных имеет значение `True`.

По умолчанию экземпляр класса `TAggregateField` создается со свойством

```
Visible = False.
```

Свойство

```
property GroupingLevel: Integer;
```

задает уровень группировки полей набора данных при вычислении. При значении 0 расчет проводится для всех записей набора данных. При значении 1 записи группируются по первому полю набора данных и расчет осуществляется для каждой группы. При значении 2 записи разбиваются на группы по первому и второму полям и т. д.

Однако группировка по уровням выше нулевого возможна, только если в наборе данных используется индекс по группирующим полям. Например, если свойство `GroupingLevel = 2` и набор данных начинаются с полей `CustNo` и `OrderNo`, в свойстве `IndexName` компонента набора данных и свойстве

```
property IndexName: String;
```

объекта агрегатного поля должно быть имя индекса, включающего оба эти поля.

По причине необходимости подключения индексов, уровень группировки выше нулевого возможен только в табличных компонентах.

Объектные поля

Наряду с обычными типами данных (строковым, целочисленным и т. д.), при работе с полями набора данных можно использовать более сложные типы, представляющие собой совокупность более простых типов.

В Delphi существуют четыре класса *объектных* полей. Это — `TADTField`, `TArrayField`, `TDataSetField`, `TReferenceField`. Их общим предком является класс `TObjectField`. Классы `TADTField`, `TArrayField` обеспечивают доступ к набору дочерних полей одного типа из родительского. Эти типы полей

можно использовать, если сервер БД поддерживает объектные типы и соответствующие поля имеются в наборе данных. Поэтому объектные поля можно создавать статически и динамически, так же, как и простые поля.

Для доступа к дочерним полям в этих классах имеются свойства:

- `property Fields: TFields;`

  которое представляет собой индексированный список объектов дочерних полей *(см. гл. 12)*;

- `property FieldValues[Index: Integer]: Variant;`

  которое содержит значения дочерних полей;

- `property FieldCount: Integer;`

  которое возвращает количество дочерних полей.

Классы `TDataSetField` и `TReferenceField` предоставляют доступ к данным из связанных наборов данных.

Ссылка на используемый набор данных задается свойством

```
property DataSet: TDataSet;
```

Более подробно классы `TDataSetField` и `TReferenceField` рассматриваются в *части V*.

Типы данных

В среде разработки Delphi можно создавать приложения для работы с самыми разными базами данных. Такая универсальность подразумевает необходимость применения средств, которые бы обеспечили возможность работы со многими типами данных, используемыми в этих базах данных.

Естественно, что существует большая группа типов данных, конкретная реализация которых практически не отличается от платформы к платформе. Это, например, строки, символы, целые и вещественные числа и т. д.

Есть типы данных, которые реализованы далеко не на каждой платформе. Есть, наконец, просто уникальные типы данных.

Для удовлетворения потребностей разработчиков в Delphi применен следующий способ работы с типами данных.

Тип данных однозначно связан с конкретным полем таблицы базы данных. Без этого поля само понятие типа данных не имеет практического смысла. В Delphi свойства абстрактного поля инкапсулирует класс `TField`, который не имеет заранее определенного типа данных. Уже от этого класса порождено целое семейство классов для типизированных полей (см. рис. 13.1), каждый из которых умеет обращаться со своим типом данных.

Примечание

В классе `TField` имеется свойство `DataType`, которое отвечает за тип данных, но оно не может быть изменено.

Весь список доступных типов данных содержится в типе `TFileldType`:

```
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);
```

В табл. 13.1 рассматриваются все типы данных, которые можно использовать при разработке приложений для работы с базами данных.

Таблица 13.1. *Типы данных*

Тип	Класс	Описание
Неизвестный (`ftUnknown`)		Неопределенный тип данных
Строковый (`ftString`)	`TStringField`	Строка длиной до 8192 символов
Целый короткий (`ftSmallint`)	`TSmallIntField`	16-битное целое в диапазоне от –32 768 до 32 767
Целый (`ftInteger`)	`TIntegerField`	32-битное целое в диапазоне от –2 147 483 648 до 2 147 483 647
Целый положительный (`ftWord`)	`TWordField`	16-битное целое в диапазоне от 0 до 65535
Логический (`ftBoolean`)	`TBooleanField`	Значения `True` и `False`
Вещественный (`ftFloat`)	`TFloatField`	Вещественные положительные и отрицательные числа с точностью 15 цифр после запятой в диапазоне от $5{,}0 \times 10^{-324}$ до $1{,}7 \times 10^{308}$
Денежный (`ftCurrency`)	`TCurrencyField`	Вещественные положительные и отрицательные числа с точностью 15 цифр после запятой в диапазоне от $5{,}0 \times 10^{-324}$ до $1{,}7 \times 10^{308}$. Дополнительно вставляется символ валюты

Таблица 13.1 *(продолжение)*

Тип	Класс	Описание
Десятичный с двоичным кодированием (ftBCD)	TBCDField	Вещественные числа с повышенной точностью (до 4 знаков перед запятой и до 20 знаков после запятой). Могут храниться в двоичном и десятичном форматах
Дата (ftDate)	TDateField	Дата
Время (ftTime)	TDateTimeField	Время
Календарный (ftDateTime)	TDateTimeField	Комбинированный формат с одновременным хранением даты и времени
Фиксированный буфер (ftBytes)	TBytesField	Набор байтов фиксированного размера. Для работы с этим типом требуется выделять и освобождать память (методы GetMem и FreeMem)
Переменный буфер (ftVarBytes)	TVarBytesField	Набор байтов переменного размера. Текущий размер буфера хранится в первых двух байтах. Для работы с этим типом требуется выделять и освобождать память (методы GetMem и FreeMem)
Автоинкрементный (ftAutoInc)	TAutoIncField	Значение поля в каждой новой записи автоматически увеличивается на 1. Целое число в диапазоне от –2 147 483 648 до 2 147 483 647. Применяется для обеспечения уникальности значений ключей
BLOB (ftBlob)	TBLOBField	Большой двоичный массив. Используется для хранения любых данных, которые можно преобразовать в цифровой массив (Memo, Graphic). В базах данных такие данные хранятся в отдельных файлах, а поле содержит лишь ссылки на них
Memo (ftMemo)	TMemoField	Набор строк произвольной длины
Графический (ftGraphic)	TGraphicField	Формат для хранения изображений
Форматированный Memo (ftFmtMemo)		Форматированный набор строк произвольной длины

Таблица 13.1 *(продолжение)*

Тип	Класс	Описание
OLE Paradox (`ftParadoxOle`)		Поле OLE для таблиц Paradox
OLE dBASE (`ftDBaseOle`)		Поле OLE для таблиц dBASE
Типизированный двоичный (`ftTypedBinary`)		Типизированный двоичный
Курсор Oracle (`ftCursor`)		Курсор для хранимых процедур сервера Oracle
Фиксированный символьный (`ftFixedChar`)	`TStringField`	Строка символов с нулевым символом в конце
Расширенный строковый (`ftWideString`)		Динамически выделяемая строка 16-битных символов в кодировке Unicode
Целый большой (`ftLargeint`)	`TLargeIntField`	64-битное целое число
Абстрактный (`ftADT`)	`TADTField`	Произвольный тип данных, создаваемый пользователем на сервере БД и используемый в приложении
Массив (`ftArray`)	`TArrayField`	Массив полей любого типа, кроме `TarrayField`
Ссылочный (`ftReference`)	`TReferenceField`	Указатель на объект, содержащийся в другой таблице
Набор данных (`ftDataSet`)	`TDataSetField`	Содержит набор данных, интегрированный в текущий набор данных
BLOB Oracle 8 (`ftOraBlob`)		Тип BLOB для сервера Oracle 8
CLOB Oracle 8 (`ftOraClob`)		Тип CLOB для сервера Oracle 8
Вариант (`ftVariant`)	`TVariantField`	Вариант
Интерфейс (`ftInterface`)	`TInterfaceField`	Ссылка на интерфейс (потомок от `IUnknown`)
Ссылка на интерфейс IDispatch (`ftIDispatch`)	`TIDispatchField`	Ссылка на интерфейс (потомок от `IDispatch`)

Таблица 13.1 *(окончание)*

Тип	Класс	Описание
Глобальный идентификатор (`ftGuid`)	`TGuidField`	Глобальный идентификатор GUID
Календарный (`ftTimeStamp`)		Календарный тип для наборов данных dbExpress
Десятичный с двоичным кодированием (`ftFMTBcd`)	`TFMTBCDField`	Тип BCD повышенной точности

Примечание

В Delphi тип данных BCD напрямую не поддерживается. Его использование обеспечивает денежный тип (`ftCurrency`). Поэтому точность BCD ограничена 20 цифрами после запятой. Для устранения этого ограничения используется тип FMTBcd, который обладает требуемой точностью.

Как видно из таблицы, наряду с традиционными типами данных, в Delphi имеются и специальные типы, использование которых значительно расширяет функциональность приложений. В частности, типы `ftInterface`, `ftIDispatch`, `ftGuid` позволяют создавать полноценные приложения БД для COM и OLE DB.

Практически на всех серверах БД пользователь имеет возможность создавать собственные типы данных. Для их использования в приложениях Delphi имеется *абстрактный тип данных* и класс `TADTField`. Абстрактный тип может включать любой *скалярный тип данных* (числа, даты, ссылки, массивы, наборы данных).

Автоинкрементный тип данных давно используется в СУБД. Поле автоинкрементного типа для каждой новой записи автоматически увеличивает свое значение на единицу. Тем самым, каждая запись имеет собственный уникальный идентификатор, который очень часто используется в качестве первичного ключа.

Данные типа BLOB (Binary Large OBject) представляют собой двоичные массивы произвольной длины. В самом поле содержится лишь ссылка на отдельный файл базы данных, в котором хранится двоичный массив. Таким образом, поля типа BLOB являются универсальным носителем любых данных, которые имеют скалярную и нескалярную структуру и которые можно преобразовать в двоичное представление.

Тип Memo представляет собой набор строк произвольной длины (его разновидность — форматированный Memo), основан на формате BLOB. Исполь-

зуется при необходимости сохранить текст из компонента `TMemo` или из текстового редактора.

Графический тип данных используется для хранения в базе данных изображений, основан на формате BLOB. Поле `TGraphicField` непосредственно взаимодействует с компонентом отображения данных (например `TDBImage`). Изображения должны храниться в формате BMP.

Типы данных ParadoxOle и *dBaseOle* разработаны специально для использования возможностей СУБД Paradox и dBASE по работе с данными OLE. В Delphi эти типы данных основаны на формате BLOB.

Специально для работы с сервером Oracle 8 предназначены *типы CLOB* и *BLOB*.

Тип `ftArray` организует массив из данных любой структуры, за исключением таких же массивов. Для каждого элемента массива может создаваться собственный объект `TField`. Для управления этим механизмом используется свойство `SparseArrays` в классе `TDataSet`.

В качестве отдельного поля в набор данных можно включить и любой другой произвольный набор данных. Для этого используется специальный тип данных и класс `TDataSetField`. Причем каждым полем из интегрированного набора данных тоже можно управлять.

Ссылочный тип данных также использует внешние наборы данных, но в этом случае можно подключать и использовать только отдельные поля.

Ограничения

Контроль вводимых в поля набора данных в Delphi возложен на объекты полей, а не на компоненты отображения данных. Именно в рамках этого вопроса мы рассмотрим работу методов-обработчиков событий базового класса `TField`.

Перед сохранением значения поля в БД всегда вызывается метод-обработчик `OnValidate`, при этом автоматически проводится проверка на выполнение задаваемых ограничений и ограничений типа данных. Кроме этого, здесь можно предусмотреть свою дополнительную обработку:

```
procedure TForm1.Table1SomeFieldValidate(Sender: TField);
begin
 if (Sender as TField).Value < 0 then
 begin
  ShowMessage('Значение поля не может быть отрицательным');
  (Sender as TField).Value := Null;
 end;
end;
```

Различают контроль значения в целом и посимвольный контроль. Метод `OnValidate` проверяет значение поля целиком. Если при проверке обнаружена ошибка, то выдается сообщение и фокус формы устанавливается на соответствующем компоненте отображения данных.

Если метод `OnValidate` не вызвал исключительной ситуации, то при сохранении значения поля в БД вызывается обработчик события `OnChange`. В принципе, можно предусмотреть операции по контролю данных и в этом методе, но тогда в случае ошибки возникает нежелательная исключительная ситуация, которая может привести к серьезным сбоям в работе приложения.

Проверить текущее значение поля перед его появлением в компоненте отображения данных можно в методе-обработчике `OnGetText`. Если параметр `DisplayText` принимает истинное значение, то в параметре `Text` передается значение свойства `DisplayText` (значение в строковом формате в таком виде, как оно будет показано в компоненте отображения данных — с символами форматирования). В противном случае в параметре `Text` передается текущее значение в строковом формате.

Примечание

При использовании метода-обработчика `OnGetText` на разработчика ложится обязанность самостоятельно предусмотреть передачу значения в компонент отображения данных, в противном случае компонент останется незаполненным.

В методе-обработчике `OnSetText` можно осуществлять текущий контроль значения в строковом формате в том виде, как оно представлено в компоненте отображения данных. Напомним, что этот обработчик вызывается при каждом изменении свойства `Text` класса `TField`.

Рассмотренные методы-обработчики удобнее всего использовать для проверки текущего значения с точки зрения программной логики. Например, чтобы отпускная цена не была ниже закупочной или чтобы остаток не был больше первоначального количества товара в партии. Для проверки правильности самого значения класс `TField` имеет несколько полезных свойств.

Если на сервере БД задано ограничение на некоторое поле, его можно использовать в приложении Delphi при помощи свойства `ImportedConstraint`

Для создания собственного ограничения можно использовать свойство `CustomConstraint`, в котором применяется синтаксис SQL:

```
Value>10
```

или

```
OutputPrice>InputPrice×1.25
```

При возникновении ошибки совсем не лишним будет, если программа выдаст некое осмысленное сообщение, которое поможет пользователю испра-

вить оплошность. При работе с методами-обработчиками это можно предусмотреть в программном коде.

Для встроенного контроля предусмотрено специальное свойство — `ConstraintErrorMessage`, которое выводится в виде сообщения при возникновении ошибки. Согласитесь, что это гораздо проще, чем исправлять и перекомпилировать соответствующие файлы ресурсов. Если приложение работает с сервером БД и возникла ошибка ограничения поля, то выводится сообщение, определяемое сервером, а не этим свойством.

Если для поля заданы ограничения, то свойство `HasConstraints` принимает истинное значение.

Посимвольный контроль данных осуществляется свойством `ValidChars`, в котором можно определить допустимые в строковом представлении значения поля символы, и методом `IsValidChar`, который определяет допустимость использования переданного в параметре символа.

Еще один мощный инструмент контроля данных предоставляет свойство `EditMask`, которое позволяет создавать шаблоны ввода данных, облегчая тем самым работу пользователя и уменьшая возможность ошибки. Рассмотрим правила создания шаблонов.

Шаблон состоит из трех частей.

Первая часть содержит управляющие символы собственно шаблона. Доступные для создания шаблона символы приведены в табл. 13.2.

Таблица 13.2. *Управляющие символы шаблона*

Символ	Описание
!	
>	Все символы после этого преобразуются в заглавные
<	Все символы после этого преобразуются в строчные
<>	Все символы после этого остаются в том регистре, как это было задано пользователем
\	Символ, следующий за этим, считается алфавитным, а не управляющим
L	В позиции этого символа обязательно должен находиться только алфавитный символ
I	В позиции этого символа может находиться алфавитный символ
A	В позиции этого символа обязательно должен находиться алфавитный символ или цифра
a	В позиции этого символа может находиться алфавитный символ или цифра

Таблица 13.2 *(окончание)*

Символ	Описание
C	В позиции этого символа обязательно должен находиться знак препинания
c	В позиции этого символа может находиться знак препинания
0	В позиции этого символа обязательно должна находиться цифра
9	В позиции этого символа может находиться цифра
#	В позиции этого символа может находиться цифра, плюс или минус
:	Символ разделения часов, минут и секунд (зависит от системных установок)
/	Символ разделения дней, месяцев, годов (зависит от системных установок)
;	Символ разделения частей шаблона
_	Символ автоматического ввода в текст пробела

В первую часть шаблона можно включать любые алфавитные символы (для создания поясняющих надписей, слов и сокращений), если их нет среди управляющих символов. Также можно использовать в качестве алфавитных и управляющие символы, для этого перед ними нужно помещать символ "\".

Вторая часть состоит из одного символа и определяет, могут ли не арифметические символы быть частью вводимого текста. Если здесь расположен ноль, то можно вводить только цифры, если любой другой символ — можно использовать и алфавитные символы.

В третьей части содержится символ, используемый для обозначения мест, запрещенных для ввода.

Части шаблона разделяются точкой с запятой.

Например, шаблон для ввода телефонного номера выглядит следующим образом:

```
!\(999\)000-0000;1;_
```

Резюме

Работа с полями является важным этапом в процессе разработки приложения баз данных. Для этого используются специальные объекты, которые инкапсулируют возможности полей таблицы БД. В Delphi имеется целая

иерархия классов, обеспечивающая применение полей самых различных типов. В основе этой иерархии лежит класс `TField`.

По способу создания объекты полей делятся на *статические* и *динамические*.

По функциональным возможностям объекты полей бывают *полями данных*, *вычисляемыми*, *синхронного просмотра*, *агрегатными*.

Объекты полей играют важную роль в работе наборов данных. С их помощью можно получить доступ к текущим значениям, задать ограничения на вводимые величины и проверить их правильность.

ГЛАВА 14

Механизмы управления данными

Наряду с описываемыми в предыдущих главах свойствами и методами, стандартный набор данных Delphi инкапсулирует ряд дополнительных механизмов, облегчающих управление записями и полями.

К ним относятся такие полезные функции, как быстрое перемещение по записям, поиск нужной записи по значениям полей, дополнительная фильтрация записей набора данных без использования возможностей СУБД и т. д. Большинство этих механизмов применяют в своей работе индексы таблиц БД.

Абстрактные методы, обеспечивающие управление данными, реализованы в базовом классе `TDataSet` *(см. гл. 12)*. А классы-потомки, в свою очередь, реализуют механизмы управления данными в соответствии с возможностями технолигий доступа к данным *(см. часть IV)*.

Все рассматриваемые в этой главе методы управления данными в полном объеме доступны только в компонентах, инкапсулирующих таблицу БД. Это связано с тем, что компоненты запросов SQL и хранимых процедур не обеспечивают полноценное использование индексов.

В этой главе рассматриваются следующие вопросы:

- связанные таблицы;
- методы поиска данных;
- диапазоны;
- быстрая навигация по набору данных;
- фильтрация записей в наборе данных.

Связанные таблицы

В рамках одного проекта таблицы БД можно связывать отношениями "один-ко-многим" и "многие-ко-многим", при этом отношения обязательно устанавливаются между индексированными полями двух таблиц.

При создании отношений в качестве главной таблицы можно использовать любой компонент, инкапсулирующий набор данных. Для задания подчиненной таблицы можно использовать только табличные компоненты *(см. гл. 12)*.

Отношение "один-ко-многим"

Для установления отношения "один-ко-многим" в наборе данных предназначены два свойства — `MasterSource` и `MasterFields`, которые задаются для подчиненной таблицы. Набор данных главной таблицы не требует никаких дополнительных настроек и заданная связь будет работать только при перемещениях по записям главной таблицы.

Свойство `MasterSource` определяет компонент `TDataSource`, который связан с главной таблицей.

Затем при помощи свойства `MasterFields` необходимо установить отношения между полями главной и подчиненной таблицы. В нем содержится имя индексированного поля, по которому устанавливается связь. Если таких полей несколько, их имена разделяются точкой с запятой. При этом не все поля, входящие в индекс, обязаны участвовать в создании отношения.

Для задания свойства `MasterFields` можно использовать Редактор связей полей (Field Link Designer), который вызывается щелчком на кнопке в поле редактирования этого свойства в Инспекторе объектов (рис. 14.1).

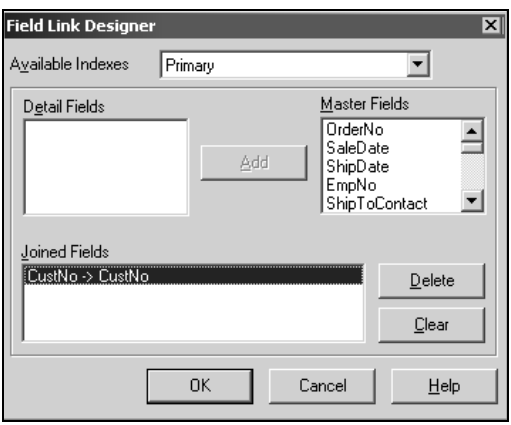


Рис. 14.1. Редактор связей полей

Здесь в разворачивающемся списке **Available Indexes** выбирается требуемый индекс для подчиненной таблицы. После этого в списке **Detail Fields** появляются имена всех полей, входящих в этот индекс. В списке **Master Fields** отображаются все поля главной таблицы.

Теперь требуется создать связи между полями. Для этого в левом списке выбирается поле подчиненной таблицы, а затем соответствующее ему поле главной таблицы в правом списке. После этого активизируется кнопка **Add**, щелчок на которой создает отношение по двум полям главной и подчиненной таблиц. Созданная связь отображается в списке **Joined Fields**.

Примечание

После создания связи по индексированным полям данный индекс становится текущим для набора данных. При этом в зависимости от типа СУБД автоматически заполняется свойство `IndexName` или `IndexFieldNames`.

Уже созданные связи можно удалить. Кнопка **Delete** удаляет выбранную связь, кнопка **Clear** — все связи.

После создания связей между полями отношение "один-ко-многим" считается установленным. Теперь достаточно открыть оба набора данных, чтобы увидеть работу отношения.

В качестве примера рассмотрим проект DemoJoins, в котором связываются таблицы из демонстрационной базы данных DBDEMOS. Для этого использованы компоненты ADO, подробнее о которых вы можете узнать из *гл. 19*.

Таблица Customers представлена в наборе данных компонента `CustTable`, она содержит данные о покупателях. Таблица Orders представлена в наборе данных компонента `OrdTable`, она содержит данные о заказах. Таблица Employee представлена в наборе данных компонента `EmpTable`, она содержат данные о продавцах (табл. 14.2).

Примечание

Приложение DemoJoins не содержит дополнительного исходного кода. Все отношения между таблицами заданы при помощи Инспектора объектов.

Отношение "один-ко-многим" задано между таблицами покупателей (Customers) и заказов (Orders). Таблица покупателей является главной. Для создания отношения установлены следующие значения свойств компонента `OrdTable` (подчиненная таблица).

Свойство `MasterSource` должно указывать на компонент `CustSource`, связанный с набором данных `CustTable`.

Свойство `MasterFields` указывает на поле `CustNo` таблицы Customers.

В наборе данных `OrdTable` включен вторичный индекс на основе поля `CustNo` (`IndexName = 'CustNo'`).

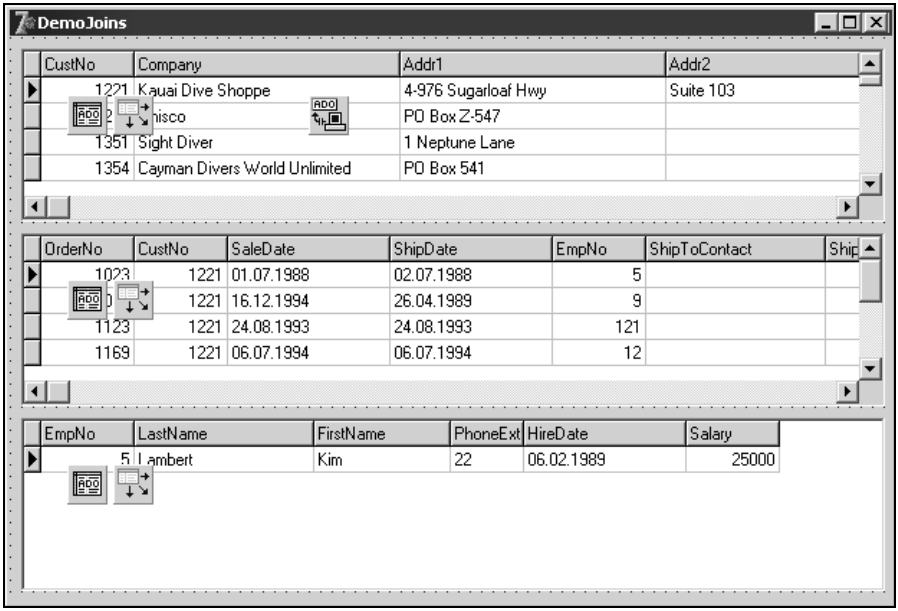


Рис. 14.2. Главная форма проекта DemoJoins

Таким образом, две таблицы связаны отношением "один-ко-многим" по индексированным полям `CustNo` (номер покупателя). В результате, при перемещении по записям таблицы покупателей, в таблице заказов будут показаны только те заказы, которые относятся к текущему покупателю

Отношение "многие-ко-многим"

Отношение "многие-ко-многим" отличается тем, что подчиненная таблица еще раз связывается в качестве главной с другой подчиненной таблицей аналогичной последовательностью действий, как и в отношении "один-ко-многим".

В приложении DemoJoins отношением "многие-ко-многим" связаны таблицы заказов (Orders) и продавцов (Employee). Таблица заказов уже работает в отношении "один-ко-многим" в качестве подчиненной.

В наборе данных `EmpTable` заданы следующие свойства:

- свойство `MasterSource` указывает на компонент `EmpSource`;
- свойство `MasterFields` содержит имя поля `EmpNo`, по которому осуществляется связь между таблицами. Для подчиненной таблицы поле `EmpNo` является первичным.

Поиск данных

В наборе данных реализованы два способа поиска записей по заданным значениям полей. Один способ основан на использовании индексов и является более быстрым, но поиск проводится только по индексированным полям. Второй способ применяет специальные методы классов наборов данных и позволяет проводить поиск по любому сочетанию полей, но он более медленный.

Поиск по индексам

Для организации индексного поиска к набору данных должен быть подключен индекс (свойства `IndexName` или `IndexFieldNames`).

Метод `FindKey` проводит поиск записи по заданным в параметре значениям ключевых полей текущего индекса набора данных. В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`, в противном случае — `False`.

Если индекс состоит из нескольких полей, значения для поиска записываются в виде множества, причем отсутствующие значения приравниваются к `Null`.

Рассмотрим простейший пример, в котором реализован поиск по вторичному индексу в таблице CUSTOLY.DB демонстрационной базы данных DBDEMOS. Индекс основан на полях `Last_Name` и `First_Name` (рис. 14.3).

В компоненте `Table1`, помимо стандартных настроек на таблицу, при помощи свойства `IndexName` задан и вторичный индекс (его имя `Names`). Значения для поиска задаются в компонентах `Edit1` и `Edit2`.

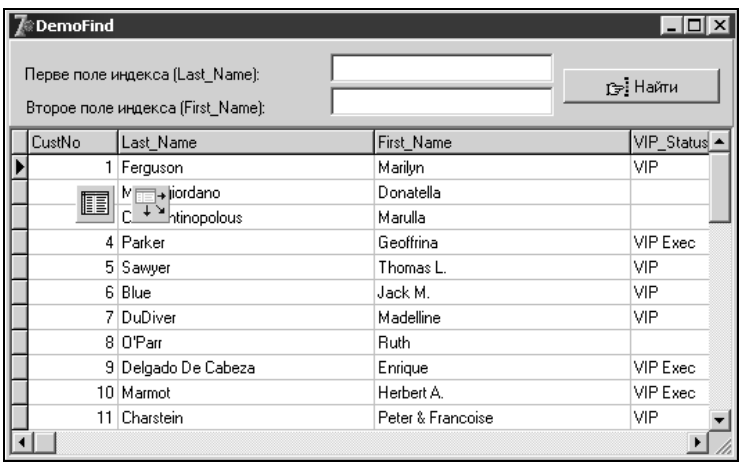


Рис. 14.3. Главная форма проекта DemoFind

Листинг 14.1. Секция `Implementation` главного модуля `Main` проекта DemoFind

```
implementation

{$R *.DFM}

procedure TForm1.FormShow(Sender: TObject);
begin
 try
  Cust.Open;
 except
  on E: EDBEngineError do ShowMessage('Ошибка при открытии таблицы');
 end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Cust.Close;
end;

procedure TForm1.FindBtnClick(Sender: TObject);
begin
 try
 if not Cust.FindKey([Edit1.Text, Edit2.Text])
  then ShowMessage('Запись не найдена');
 except
  on E: EDatabaseError
   do ShowMessage('Ошибка поиска');
 end;
end;

end.
```

Набор данных открывается в методе-обработчике `FormShow` при открытии формы и закрывается в методе-обработчике `FormClose`. При щелчке на кнопке `FindBtn` в метод `FindKey` передаются значения для поиска из компонентов `Edit1` и `Edit2`.

Поиск в диапазоне

Индексный поиск можно организовать группой методов, подобно созданию диапазонов. Метод `SetKey` переводит набор данных в состояние `dsSetKey`, затем должно следовать присваивание ключевым полям значений для поиска. Сам поиск осуществляется методом `GotoKey`:

```
with Table1 do
begin
 SetKey;
 Fields[0].Value := '428';
 GotoKey;
end;
```

В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`. Вместо этого метода можно применять метод `GotoNearest`, который в случае неудачного поиска ищет запись, минимально отличающуюся от критерия поиска.

Изменение параметров поиска осуществляется методом `EditKey`.

Поиск по произвольным полям

Для поиска по произвольной выборке полей можно использовать методы `Locate` и `Lookup`.

```
function Locate(const KeyFields: string; const KeyValues: Variant;
Options: TLocateOptions): Boolean;
function Lookup(const KeyFields: string; const KeyValues: Variant;
const ResultFields: string): Variant;
```

В метод `Locate` необходимо передать список полей, по которым будет идти поиск (параметр `KeyFields`, имена полей разделяются точкой с запятой), их требуемые значения (параметр `KeyValues`, значения разделяются запятой) и настройки поиска (параметр `Options`). В настройках можно задать опцию `loCaseInsensitive`, которая отключает проверку на регистр символов, и опцию `loPartialKey`, которая включает поиск с минимальными отличиями. В случае успеха поиска курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`.

```
Table1.Locate('Last_Name;First_Name', VarArrayOf(['Edit1.Text',
'Edit2.Text']), []);
```

В метод `Lookup` передается список полей для поиска (параметр `KeyFields`, имена полей разделяются точкой с запятой) и их требуемые значения (параметр `KeyValues`, значения разделяются запятой). В случае успешного поиска функция возвращает массив значений типа вариант для полей, названия которых содержатся в параметре `ResultFields`.

```
Table1.Lookup('Last_Name;First_Name', VarArrayOf(['Edit1.Text',
'Edit2.Text']), 'Last_Name;First_Name');
```

Оба эти метода автоматически используют быстрый индексный поиск в случае, если в параметре `KeyFields` задать поля индекса.

Фильтры

Наиболее эффективным способом отбора записей в набор данных (особенно из больших таблиц) является создание и выполнение соответствующего запроса SQL. Но что делать, если набор данных функционирует на базе табличного компонента? В этом случае на помощь приходит встроенный в набор данных механизм фильтрации данных.

Применение фильтра основано всего на двух основных свойствах и одном вспомогательном. Текст фильтра должен содержаться в свойстве `Filter`, а свойство `Filtered` включает и выключает фильтр. Параметры фильтра определяются свойством `FilterOptions`.

Примечание

Компонент `TQuery` также может использовать фильтры. Эта возможность подчас позволяет легко и изящно решать довольно сложные проблемы, которые иначе требуют изменения текста запроса или создания нового компонента запроса.

При использовании фильтра его текст транслируется в синтаксис SQL и передается для выполнения на сервер или через соответствующий драйвер в локальную СУБД.

Фильтры можно создавать двумя способами:

- при помощи свойства `Filter` создаются довольно простые фильтры, для которых достаточно предоставляемого механизмом фильтрации синтаксиса;
- для создания более сложных фильтров с применением всех возможных средств языка программирования применяется метод-обработчик набора данных `OnFilterRecord`.

Фильтры можно разделить на статические и динамические.

Статические фильтры создаются во время разработки приложения и могут использовать как свойство `Filter`, так и метод `OnFilterRecord`.

Динамические фильтры можно создавать и редактировать во время выполнения приложения, для них используется только свойство `Filter`.

При создании текста фильтра для свойства `Filter` используются имена полей соответствующей таблицы БД, а для задания отношений применяются все операторы сравнения (`>`, `>=`, `<`, `<=`, `=`, `<>`) и логические операторы (`AND`, `OR`, `NOT`):

```
Field1>100 AND Field2=20
```

Сравнивать между собой два поля нельзя. Следующий фильтр вызовет ошибку при попытке использования:

```
ItemCount=Balance AND InputPrice>OutputPrice
```

При создании динамических фильтров можно изменять как выражение фильтра целиком, так и его части. Например, ограничивающее значение для поля можно задавать при помощи элементов управления формы, что позволяет пользователю приложения управлять фильтрацией набора данных:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 with Table1 do
 begin
  Filtered := False;
  Filter := 'Field1>=' + TEdit(Sender).Text;
  Filtered := True;
 end;
end;
```

В фильтрах можно производить отбор по частям строк для строковых полей, для этого используется символ звездочка:

```
ItemName='A*'
```

Фильтр начинает работать только после того, как свойству `Filtered` присваивается истинное значение. Перед изменением текста динамического фильтра или для отключения фильтра свойству `Filtered` присваивается значение `False`.

Параметры фильтра определяются свойством `FilterOptions`:

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

Параметр `foCaseInsensitive`, будучи включенным в свойстве, отключает сравнение строковых значений с учетом регистра символов.

Параметр `foNoPartialCompare` отключает отбор строковых значений по части строки.

Метод-обработчик `OnFilterRecord` имеет следующее объявление:

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

Если этот метод создан для набора данных, то он вызывается для каждой его записи. Программный код метода должен присваивать параметру `Accept` истинное или ложное значение. В результате запись передается в набор данных или отсекается:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
```

```
begin
 Accept := ArchOrdersArchDat.AsString >= DateEdit1.Text;
end;
```

Важнейшее преимущество метода `OnFilterRecord`, по сравнению со свойством `Filter`, заключается в том, что в этом методе-обработчике можно сравнивать поля и производить вычисления над их значениями.

Недостатком метода является недостаточная гибкость, хотя такой фильтр можно модифицировать путем присвоения методу процедурной переменной, содержащей ссылку на новый метод.

Быстрый переход к помеченным записям

Закладки, как инструмент работы с записями набора данных, позволяют осуществлять быстрое перемещение на нужную запись. Набор данных может содержать неограниченное число закладок, каждая из которых представляет собой указатель. Закладку можно создать только для текущей записи набора данных.

При работе с закладками используются три основных метода:

- метод `GetBookmark` создает новую закладку для текущей записи;
- метод `GotoBookmark` осуществляет переход к закладке, переданной в параметре;
- метод `FreeBookmark` удаляет закладку, переданную в параметре.

Кроме этого, можно использовать метод `BookmarkValid`, который проверяет, указывает ли закладка на реально существующую запись. Метод `CompareBookmark` позволяет сравнить между собой две закладки:

```
var Bookmark1, Bookmark2: TBookmark;
...
if Table1.CompareBookmark(Bookmark1, Bookmark2) = 1
then ShowMessage('Закладки одинаковы');
```

В наборе данных имеется свойство `Bookmark`, которое содержит название текущей закладки.

Рассмотрим небольшой пример, где право управлять закладками предоставлено пользователю (рис. 14.4). На форме, помимо других элементов управления (среди которых есть компонент `TDBGrid`), имеются две кнопки. Кнопка `StartBookmark` помечает текущую запись, кнопка `StopBookmark` переходит к закладке, а затем уничтожает ее.

Рис. 14.4. Главная форма проекта DemoBookmark

Листинг 14.2. Пример использования закладок

```
implementation

{$R *.DFM}

var SaveRecPos: TBookMark;

procedure TMainForm.FormShow(Sender: TObject);
begin
 try
  Cust.Open;
  BookmarkControl.Brush.Color := clBtnFace;
 except
  ShowMessage('Ошибка открытия набора данных');
 end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Cust.Close;
end;

procedure TMainForm.StartBookmarkClick(Sender: TObject);
begin
 if Not Cust.BookmarkValid(SaveRecPos)
  then SaveRecPos := Cust.GetBookmark;
```

```
 BookmarkControl.Brush.Color := clLime
end;

procedure TMainForm.StopBookmarkClick(Sender: TObject);
begin
 with Cust do
 begin
  if Not BookmarkValid(SaveRecPos)
   then Exit;
  GotoBookmark(SaveRecPos);
  FreeBookmark(SaveRecPos);
  SaveRecPos := Nil;
 end;
 BookmarkControl.Brush.Color := clBtnFace;
end;

end.
```

Использование метода `BookmarkValid` позволяет корректно переопределять закладку, если она уже установлена, и избежать ошибок при произвольных нажатиях кнопок. Компонент `BookmarkControl` типа `TShape` сигнализирует о том, что закладка установлена или удалена.

Примечание

Закладки также используются в компоненте `TDBGrid`. Он имеет свойство `SelectedRows` типа `TBookmarkList`, которое представляет собой список закладок, указывающих на одновременно выделенные записи.

Диапазоны

В наборе данных, помимо фильтров, имеется еще одно средство отбора записей. Группа методов позволяет на основе использования индексов отбирать в набор данных только те записи, значения индексированных полей которых (для текущего индекса) соответствуют диапазону заданных величин.

Диапазоны работают быстрее фильтров, но менее гибки и не так удобны в работе. При использовании диапазонов набор данных обязательно должен находиться в состоянии `dsSetKey` *(см. ниже)*.

Для того чтобы включить диапазон, необходимо задать стартовое и конечное значение диапазона для ключевых полей, затем применить созданный диапазон к набору данных. Работающий диапазон можно модифицировать.

Примечание

Все методы работы с диапазонами используют те поля, которые заданы в текущем индексе. Для таблиц Paradox и dBASE это свойство `IndexName`. Для таблиц серверов SQL это свойство `IndexFieldNames`.

Метод SetRangeStart переводит набор данных в режим dsSetKey, следующее за этим присваивание ключевым полям значений означает задание начальной границы диапазона.

Метод SetRangeEnd переводит набор данных в режим dsSetKey, следующее за этим присваивание ключевым полям значений означает задание конечной границы диапазона.

После этого необходимо использовать метод ApplyRange, который применяет созданный диапазон к набору данных:

```
with Table1 do
begin
 SetRangeStart;
 Fields[0].Value := '439';
 SetRangeEnd;
 Fields[1].Value := '522';
 ApplyRange;
end;
```

Работающий диапазон можно модифицировать аналогичным образом: после вызова методов EditRangeStart и EditRangeEnd необходимо задать новые границы для ключевых полей и снова вызвать метод ApplyRange:

```
with Table1 do
begin
 EditRangeStart;
 Fields[0].Value := '500';
 EditRangeEnd;
 Fields[1].Value := '522';
 ApplyRange;
end;
```

Отмена диапазона осуществляется методом CancelRange.

Если индекс содержит несколько полей, то перед вызовом метода ApplyRange необходимо задать значения для всех ключевых полей.

Для одновременного задания верхней и нижней границы диапазона можно использовать метод SetRange.

```
with Table1 do
begin
 SetRange(['500'], ['522']);
 ApplyRange;
end;
```

Тем, какая граница будет у диапазона — открытая или закрытая, управляет свойство KeyExclusive. Если оно имеет значение True, граничные значения в диапазон не включаются, в противном случае — включаются.

Резюме

Разработчик приложений БД в Delphi может использовать ряд полезных механизмов набора данных, которые реализованы для компонентов всех технологий доступа к данным.

К этим механизмам относятся методы быстрого поиска и перехода к найденным записям; связывания наборов данных по индексированным полям; метод дополнительной фильтрации записей набора данных.

ГЛАВА 15

Компоненты отображения данных

До этого момента мы рассмотрели аспекты создания приложений баз данных, касающиеся организации доступа к данным и создания в приложениях наборов данных. Теперь более подробно остановимся на вопросах отображения данных в приложениях (интерфейс приложений).

Отображение данных обеспечивает достаточно представительный набор компонентов VCL Delphi. Многие из них унаследованы от компонентов, инкапсулирующих стандартные элементы управления. Для связи с набором данных эти компоненты используют компонент `TDataSource`.

Механизмы управления данными реализованы в компонентах наборов данных и активно взаимодействуют с компонентами отображения данных.

В этой главе рассматриваются следующие вопросы:

- использование стандартных компонентов отображения данных;
- навигация по данным;
- механизм синхронного просмотра данных;
- использование графиков для представления данных.

Классификация компонентов отображения данных

Все компоненты отображения данных можно разделить на группы по нескольким критериям (рис. 15.1).

Большинство компонентов предназначены для работы с отдельным полем, т. е. при перемещении по записям набора данных такие компоненты показывают текущие значения только одного поля. Для соединения с набором

данных через компонент TDataSource предназначено свойство DataSource. Поле задается свойством DataField.

Компоненты TDBGrid и TDBCtrlGrid обеспечивают просмотр наборов данных целиком или в произвольном сочетании полей. В них присутствует только свойство DataSource.

Особенную роль среди компонентов отображения данных играет компонент TDBNavigator. Он не показывает данные и не предназначен для их редактирования, зато обеспечивает навигацию по набору данных.

Наиболее часто в практике программирования используются компоненты TDBGrid, TDBEdit и TDBNavigator.

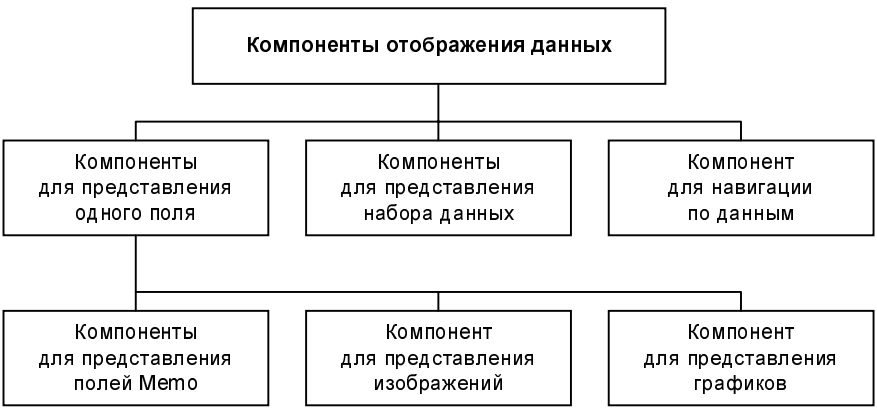


Рис. 15.1. Классификация компонентов отображения данных

Для представления и редактирования информации, содержащейся в полях типа Memo, используются специальные компоненты TDBMemo и TDBRichEdit.

Для просмотра (без редактирования) изображений предназначен компонент TDBImage.

Отдельную группу составляют компоненты синхронного просмотра данных. Они обеспечивают показ значений поля из одной таблицы в соответствии со значениями поля из другой таблицы.

Наконец, данные можно представить в виде графика. Для этого предназначен компонент TDBChart.

Как видите, набор компонентов отображения данных весьма разнообразен и позволяет решать задачи по созданию любых интерфейсов для приложений баз данных.

Ввиду общности решаемых задач, компоненты отображения данных имеют несколько важных общих свойств, которые представлены в табл. 15.1 и в дальнейшем изложении опущены.

Таблица 15.1. *Общие свойства компонентов отображения данных*

Объявление	Описание
`property DataField: string;`	Поле связанного с компонентом набора данных
`property DataSource: TDataSource;`	Связываемый с компонентом компонент `TDataSource`
`property Field: Tfield;`	Обеспечивает доступ к классу `TField`, который соответствует полю набора данных, заданному свойством `DataField`
`property ReadOnly: Boolean;`	Управляет работой режима "только для чтения"

Табличное представление данных

Компонент *TDBGrid*

Этот компонент инкапсулирует двумерную таблицу, в которой строки представляют собой записи, а столбцы — поля набора данных.

Компонент `TDBGrid` является потомком классов `TDBCustomGrid` и `TCustomGrid`. От класса `TCustomGrid` наследуются все функции отображения и управления работой двумерной структуры данных. Класс `TDBCustomGrid` обеспечивает визуализацию и редактирование полей из набора данных, причем `TDBGrid` только публикует свойства и методы класса `TDBCustomGrid`, не добавляя собственных.

В компоненте `TDBGrid` можно отображать произвольное подмножество полей используемого набора данных, но число записей ограничить нельзя — в компоненте всегда присутствуют все записи связанного набора данных. Требуемый набор полей можно составить при помощи специального Редактора столбцов, который открывается при двойном щелчке на компоненте, перенесенном на форму, или кнопкой свойства `Columns` в Инспекторе объектов.

Новая колонка добавляется при помощи кнопки **Add New**, после этого ее название появляется в списке колонок (рис. 15.2). Для выбранной в списке колонки доступные для редактирования свойства появляются в Инспекторе объектов. Колонки в списке можно редактировать, удалять, менять местами.

При помощи кнопки **Add All Fields** в сетку можно добавить все поля набора данных.

Каждая колонка компонента `TDBGrid` описывается специальным классом `TColumn`, а совокупность колонок доступна через свойство `Columns` компонента, оно имеет тип `TDBGridColumns` и представляет собой индексированный список объектов колонок. Поле набора данных связывается с конкрет-

ной колонкой при помощи свойства `FieldName` класса `TColumn`. При этом в колонку автоматически переносятся все необходимые параметры поля, в частности заголовок поля, настройки шрифтов, ширина поля. После ручного изменения параметров первоначальные значения восстанавливаются методами соответствующих объектов `TColumn`.

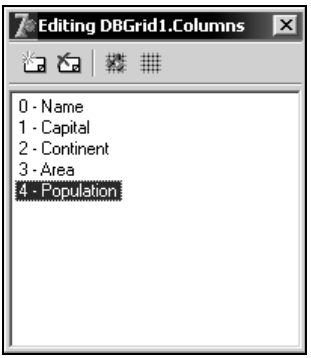


Рис. 15.2. Редактор колонок компонента `TDBGrid`

При помощи метода `DefaultDrawColumnCell` и метода-обработчика `OnDrawColumnCell` можно управлять процессом отображения данных в ячейках.

Метод `DefaultDrawDataCell` предназначен только для обеспечения обратной совместимости по коду с более ранними версиями.

Настройка параметров компонента `TDBGrid`, от которых зависит его внешний вид и некоторые функции, осуществляется при помощи свойства `Options` (табл. 15.2). Текущая позиция в двумерной структуре данных может быть определена свойствами `SelectedField`, `SelectedRows`, `SelectedIndex`.

При необходимости разработчик может использовать разнообразные методы-обработчики событий. Среди них есть как стандартные методы, присущие всем элементам управления, так и специфические.

Например, при помощи метода-обработчика `OnEditButtonClick` можно предусмотреть вызов специализированной формы при щелчке на кнопке в ячейке:

```
procedure TForm1.DBGrid1EditButtonClick(Sender: TObject);
begin
 if DBGrid1.SelectedIndex = 2 then SomeForm.ShowModal;
end;
```

Примечание

Объект колонки `TColumn` имеет свойство `ButtonStyle`. Если ему присвоить значение `cbsEllipsis`, то при активизации ячейки этой колонки в правой части ячейки появляется кнопка.

Таблица 15.2. *Свойства и методы компонента* `TDBGrid`

Объявление	Тип	Описание
Свойства		
`property Columns: TDBGridColumns;`	Pb	Содержит коллекцию объектов `TColumn`, описывающих колонки компонента
`property DefaultDrawing: Boolean;`	Pb	Определяет способ визуализации данных в сетке. При значении `True` данные отображаются автоматически. При значении `False` используется метод-обработчик `OnDrawColumnCell`
`property FieldCount: Integer;`	Ro	Возвращает число видимых колонок сетки
`property Fields[Index: Integer]: TField;`	Ro	Массив объектов полей набора данных, отображаемых в компоненте
`TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgRowSelect, dgAlwaysShowSelection, dgConfirmDelete, dgCancelOnExit, dgMultiSelect);` `TDBGridOptions = set of TDBGridOption;`	Pb	Определяет особенности визуализации и поведения компонента: • `dgEditing` — данные можно редактировать; • `dgAlwaysShowEditor` — данные в сетке всегда в режиме редактирования; • `dgTitles` — видны заголовки колонок; • `dgIndicator` — в начале строки виден номер текущей колонки; • `dgColumnResize` — колонки можно перемещать и менять их ширину; • `dgColLines` — видны линии между колонками; • `dgRowLines` — видны линии между строками; • `dgTabs` — для перемещения по строкам можно использовать клавиши <Tab> и <Shift>+<Tab>; • `dgRowSelect` — можно выделять целые строки, при этом игнорируются установки `dgEditing` и `dgAlwaysShowEditor`; • `dgAlwaysShowSelection` — выделение текущей ячейки сохраняется, даже если сетка не активна; • `dgConfirmDelete` — при удалении строк появляется запрос о подтверждении операции;

Таблица 15.2 *(продолжение)*

Объявление	Тип	Описание
(прод.)		• `dgCancelOnExit` — созданные пустые строки при уходе из сетки не сохраняются; • `dgMultiSelect` — можно выделять несколько строк одновременно
`property SelectedField: TField;`	Pu	Содержит объект текущего поля
`property SelectedIndex: Integer;`	Pu	Содержит номер текущей колонки в массиве свойства `Columns`
`property SelectedRows: TBookmarkList;`	Ro	Набор закладок на записи набора данных, соответствующих выделенным строкам сетки
`property TitleFont: TFont;`	Pb	Шрифт заголовков колонок
`property EditorMode: Boolean;`	Pu	Показывает, можно ли редактировать текущую ячейку
`property FixedColor: TColor;`	Pb	Цвет фона неподвижных ячеек сетки
Методы		
`procedure DefaultDrawColumnCell(const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState);`	Pu	Перерисовывает текст в ячейке колонки с номером `DataCol`. Ячейка задается прямоугольником `Rect` на канве сетки. Параметр `State` определяет состояние ячейки после перерисовки. Параметр `Column` содержит экземпляр класса колонки, которой принадлежит ячейка
`procedure DefaultDrawDataCell(const Rect: TRect; Field: TField; State: TGridDrawState);`	Pu	Перерисовывает текст в ячейке колонки, определяемой параметром `Field`, содержащим связанный с колонкой объект поля. Ячейка задается прямоугольником `Rect` на канве сетки. Параметр `State` определяет состояние ячейки после перерисовки
`procedure DefaultHandler(var Msg); override;`	Pu	Вызывает всплывающее меню для колонки, которой соответствуют текущие координаты мыши. Компонент должен обрабатывать сообщение `WM_RBUTTONUP`
`function ExecuteAction(Action: TBasicAction): Boolean; override;`	Pu	Выполняет действие, заданное параметром `Action`, по отношению к данному компоненту

Таблица 15.2 *(продолжение)*

Объявление	Тип	Описание
`procedure ShowPopupEditor(Column: TColumn; X: Integer = Low(Integer); Y: Integer = Low(Integer)); dynamic;`	Pu	Открывает набор данных, связанный с передаваемой параметром `Column` колонкой в новом окне. Работает только для типов данных абстрактный и набор данных. Параметры X и Y определяют положение нового окна
`function ValidFieldIndex(FieldIndex: Integer): Boolean;`	Pu	Возвращает значение `True`, если колонка с номером `FieldIndex` связана с полем набора данных
`type TGridCoord = record X: Longint; Y: Longint; end;` `function MouseCoord(X, Y: Integer): TGridCoord;`	Pu	Возвращает номера строки и столбца, соответствующие ячейке, которой принадлежат экранные координаты X и Y
Методы-обработчики событий		
`type TDBGridClickEvent = procedure (Column: TColumn) of object;` `property OnCellClick: TDBGridClickEvent;`	Pb	Вызывается при щелчке мышью на ячейке. Параметр `Column` содержит колонку, которой принадлежит ячейка
`property OnColEnter: TNotifyEvent;`	Pb	Вызывается при переносе фокуса на новую колонку сетки
`property OnColExit: TNotifyEvent;`	Pb	Вызывается перед переносом фокуса из текущей колонки
`type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;` `property OnColumnMoved: TMovedEvent;`	Pb	Вызывается при переносе колонки в сетке на новое место при помощи мыши. Параметр `FromIndex` возвращает номер старого положения колонки. Параметр `ToIndex` возвращает номер нового положения колонки
`type TDrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;` `property OnDrawColumnCell: TDrawColumnCellEvent;`	Pb	Вызывается при перерисовке ячейки. Параметр `Rect` определяет ячейку по координатам прямоугольника на канве. Параметр `DataCol` возвращает номер колонки в сетке. Параметр `Column` содержит объект колонки. Параметр `State` возвращает состояние колонки

Таблица 15.2 *(окончание)*

Объявление	Тип	Описание

`type`
`TDrawDataCellEvent = procedure (Sender: TObject; const Rect: TRect; Field: TField; State: TGridDrawState) of object;`
`property OnDrawDataCell: TDrawDataCellEvent;`
	Pb	Вызывается при перерисовке ячейки перед обработчиком `OnDrawColumnCell`, если свойство `Columns.State = csDefault`.
		Этот метод лучше не применять, т. к. он используется только для обеспечения обратной совместимости с ранними версиями

`property OnEditButtonClick: TNotifyEvent;`	Pb	Вызывается при щелчке мышью на кнопке в ячейке
`type TDBGridClickEvent = procedure (Column: TColumn) of object;` `property OnTitleClick: TDBGridClickEvent;`	Pb	Вызывается при щелчке мышью на заголовке колонки. Колонка определяется параметром `Column`

В работе компонента `TDBGrid` важную роль играет класс `TColumn`, который инкапсулирует свойства колонки или столбца сетки (табл. 15.3). Его основным назначением является правильное отображение данных из поля набора данных, связанного с этой колонкой. Поэтому объект колонки обладает свойствами и методами, которые позволяют произвольным образом задавать параметры отображения данных (цвет, шрифт, ширину и т. д.). Первоначальные значения берутся из связанных с колонками полей. Измененные свойства можно восстановить при помощи группы специальных методов (`DefaultColor`, `DefaultFont` и др.).

Свойство `AssignedValues` позволяет в любой момент определить, какие первоначальные настройки были изменены.

За отображение заголовка колонки отвечает свойство `Title`, представляющее собой ссылку на экземпляр объекта `TColumnTitle`. Здесь можно задать текст заголовка, параметры шрифта текста заголовка и цвет фона заголовка. По умолчанию текст заголовка берется из свойства `DisplayLabel` объекта `TField` *(см. гл. 13)*.

Каждой колонке можно придать список, который разворачивается при щелчке на кнопке в активной ячейке колонки. Выбранное в списке значение автоматически заносится в ячейку. Для реализации этой возможности применяется свойство `PickList` типа `TStrings`. Достаточно лишь заполнить список значениями во время разработки или выполнения (рис. 15.3).

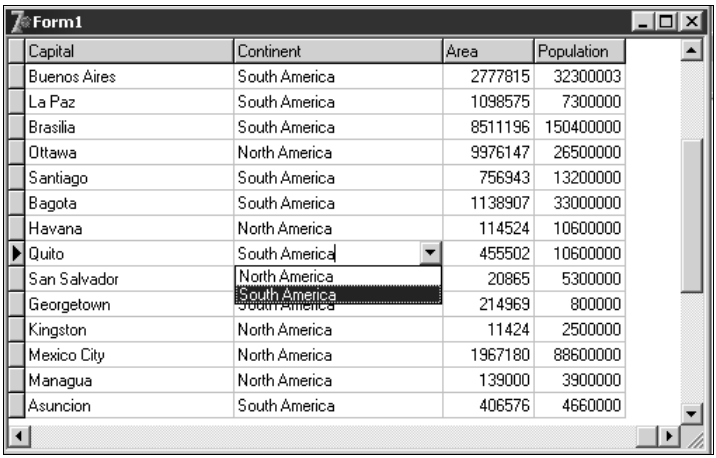


Рис. 15.3. Список колонки в компоненте `TDBGrid`

Таблица 15.3. *Свойства и методы класса* `TColumn`

Объявление	Тип	Описание
Свойства		
`property Alignment: TAlignment;`	Pb	Определяет выравнивание данных в колонке

`type`
`TColumnValue = (cvColor, cvWidth, cvFont, cvAlignment, cvReadOnly, cvTitleColor, cvTitleCaption, cvTitleAlignment, cvTitleFont, cvImeMode, cvImeName);`
`TColumnValues = set of TColumnValue;`
`property AssignedValues: TColumnValues;`
	Ro	Возвращает набор атрибутов колонки, которые были изменены по сравнению с первоначальными

`type TColumnButtonStyle = (cbsAuto, cbsEllipsis, cbsNone);` `property ButtonStyle: TColumnButtonStyle;`	Pb	Задает способ редактирования данных в колонке: • `cbsAuto` — кнопка в редактируемой ячейке появляется, если связанное поле является полем синхронного просмотра; • `cbsEllipsis` — кнопка в редактируемой ячейке появляется всегда, щелчок на кнопке вызывает обработчик `OnEditButtonClick`;

Таблица 15.3 *(продолжение)*

Объявление	Тип	Описание
(прод.)		• `cbsNone` — при редактировании ячейки кнопка не появляется
`property Color: TColor;`	Pb	Цвет фона колонки
`property DisplayName: string;`	Pu	Название колонки в списке Редактора столбцов
`property DropDownRows: Cardinal;`	Pb	Определяет число строк разворачивающегося списка ячейки
`property Expandable: Boolean;`	Pu	В значении `True` разрешает показ связанных с полем колонки дочерних полей абстрактного, ссылочного типов и массивов
`property Expanded: Boolean;`	Pb	При значении `True` каждое дочернее поле отображается в новой колонке. При значении `False` дочерние поля отображаются через точку с запятой и не доступны для редактирования
`property FieldName: string;`	Pb	Название поля, связанного с колонкой
`property Font: TFont;`	Pb	Шрифт данных в колонке
`property Grid: TCustomDBGrid;`	Ro	Определяет сетку, содержащую эту колонку
`property ParentColumn: TColumn;`	Ro	Определяет колонку-владельца текущей колонки. Используется для дочерних полей
`property PickList: TStrings;`	Pb	Содержит разворачивающийся список, используемый при редактировании данных
`property PopupMenu: TPopupMenu;`	Pb	Связывает с колонкой всплывающее меню
`property Showing: Boolean;`	Ro	Возвращает значение `True`, если колонка видима
`property Title: TColumnTitle;`	Pb	Задает текст заголовка и его параметры
`property Visible: Boolean;`	Pb	Задает видимость колонки
`property Width: Integer;`	Pb	Задает ширину колонки в пикселах

Таблица 15.3 *(окончание)*

Объявление	Тип	Описание
Методы		
`procedure Assign(Source: TPersistent); override;`	Pu	Копирует колонку `Source` в текущую колонку
`function DefaultAlignment: TAlignment;`	Pu	Возвращает первоначальное значение выравнивания колонки
`function DefaultColor: TColor;`	Pu	Возвращает первоначальный фоновый цвет колонки
`function DefaultFont: TFont;`	Pu	Возвращает первоначальный шрифт данных в колонке
`type TImeMode = (imDisable, imClose, imOpen, imDontCare, imSAlpha, imAlpha, imHira, imSKata, imKata, imChinese, imSHanguel, imHanguel);` `function DefaultImeMode: TImeMode;`	Pu	Возвращает первоначальный способ ввода символов
`type TImeName = type string;` `function DefaultImeName: TImeName;`	Pu	Возвращает первоначальное имя редактора способа ввода символов
`function DefaultReadOnly: Boolean;`	Pu	Возвращает первоначальный режим редактирования данных
`function DefaultWidth: Integer;`	Pu	Возвращает первоначальную ширину колонки в пикселах
`function Depth: Integer;`	Pu	Возвращает число непосредственных предков колонки
`procedure RestoreDefaults; virtual;`	Pu	Восстанавливает первоначальные настройки колонки

При работе с компонентом `TDBGrid` все операции с отдельными колонками осуществляются при помощи экземпляра класса `TDBGridColumns`, который инкапсулирует список объектов колонок (свойство `Columns` компонента `TDBGrid`). Доступ к колонкам осуществляется при помощи свойства `Items`. Нумерация колонок начинается с нуля.

При помощи свойств и методов класса `TDBGridColumns` можно изменять настройки полей компонента `TDBGrid` во время выполнения (табл. 15.4).

Свойство `State` определяет способ создания колонок. Его значение устанавливается автоматически. При создании колонок для всех полей сразу (кнопка **Add All Fields** Редактора столбцов) устанавливается значение `csDefault`.

При любом ручном изменении свойств устанавливается значение `csCustomized`. При программном изменении значения свойства во время выполнения все существующие колонки удаляются.

Все данные из существующих колонок можно сохранить в файле или потоке при помощи методов `SaveToFile` и `SaveToStream`, а затем загрузить их обратно методами `LoadFromFile` и `LoadFromStream`.

Таблица 15.4. *Свойства и методы класса* `TDBGridColumns`

Объявление	Тип	Описание
Свойства		
`property Grid: TCustomDBGrid;`	Ro	Возвращает ссылку на сетку, владеющую данным объектом
`property Items[Index: Integer]: TColumn default;`	Pu	Индексный список объектов колонок сетки:
`type TDBGridColumnsState = (csDefault, csCustomized);` `property State: TDBGridColumnsState;`	Pu	Определяет способ создания колонок сетки: • `csDefault` — колонки создаются динамически с параметрами, соответствующими связанным полям; • `csCustomized` — параметры колонок определены разработчиком и могут отличаться от параметров полей
`property Count: Integer;`	Pu	Возвращает общее число колонок
Методы		
`function Add: TColumn;`	Pu	Добавляет новый объект `TColumn`
`procedure LoadFromFile(const Filename: string);`	Pu	Загружает данные в объект из файла `FileName`
`procedure LoadFromStream(S: TStream);`	Pu	Загружает данные в объект из потока `S`
`procedure RebuildColumns;`	Pu	Удаляет существующие колонки и создает новые, основываясь на параметрах полей набора данных
`procedure RestoreDefaults;`	Pu	Восстанавливает первоначальные настройки колонок
`procedure SaveToFile(const Filename: string);`	Pu	Сохраняет данные из колонок в файле `FileName`
`procedure SaveToStream(S: TStream);`	Pu	Сохраняет данные из колонок в потоке `S`

Компонент *TDBCtrlGrid*

Компонент TDBCtrlGrid внешне напоминает компонент TDBGrid, но никак не связан с классом TCustomDBGrid, а наследуется напрямую от класса TWinControl.

Этот компонент позволяет отображать данные в строках в произвольной форме. Компонент представляет собой набор панелей, каждая из которых служит платформой для размещения данных отдельной записи набора данных. На панели могут размещаться любые компоненты отображения данных, предназначенные для работы с отдельным полем. С каждым таким компонентом можно связать нужное поле набора данных. При открытии набора данных в компоненте TDBCtrlGrid на каждой новой панели создается набор компонентов отображения данных, аналогичный тому, который был создан на одной панели во время разработки.

На панель можно переносить только те компоненты отображения данных, которые показывают значение одного поля для единственной записи набора данных. Нельзя использовать компоненты TDBGrid, TDBCtrlGrid, TDBRichEdit, TDBListBox, TDBRadioGroup, TDBLookupListBox.

После того, как для компонента TDBCtrlGrid задано значение свойства DataSource, все переносимые на панель компоненты отображения данных автоматически связываются с указанным компонентом TdataSource (табл. 15.5). Самостоятельное задание свойства DataSource для дочерних компонентов отображения данных не допускается. В них требуется определить только поля.

Компонент может отображать панели в одну или несколько колонок. Для задания числа колонок панелей используется свойство ColCount. Число видимых строк панелей определяется свойством RowCount. Вертикальное или горизонтальное размещение колонок панелей зависит от значения свойства Orientation.

При использовании нескольких колонок панелей курсор перемещается по колонке сверху вниз с последующим переходом на следующую колонку. Направление движения курсора не зависит от значения свойства Orientation.

Размеры одной панели определяются свойствами PanelHeight и PanelWidth. Они взаимосвязаны с размерами самого компонента. При изменении значений свойств PanelHeight и PanelWidth размеры компонента изменяются таким образом, чтобы в нем помещалось указанное в свойствах ColCount и RowCount число панелей и наоборот.

Не рекомендуется размещать на панели компоненты TDBMemo и TDBImage, т. к. это может привести к значительному снижению производительности.

Таблица 15.5. *Свойства и методы компонента* `TDBCtrlGrid`

Объявление	Тип	Описание
Свойства		
`property AllowDelete: Boolean;`	Pb	Разрешает или запрещает удаление текущей записи
`property AllowInsert: Boolean;`	Pb	Разрешает или запрещает вставку новой записи
`property Canvas: TCanvas;`	Ro	Канва компонента
`property ColCount: Integer;`	Pb	Определяет число колонок с панелями
`property EditMode: Boolean;`	Pu	Разрешает или запрещает редактирование данных
`type TDBCtrlGridOrientation = (goVertical, goHorizontal);` `property Orientation: TDBCtrlGridOrientation;`	Pb	Определяет порядок следования записей — по горизонтали или по вертикали
`type TDBCtrlGridBorder = (gbNone, gbRaised);` `property PanelBorder: TDBCtrlGridBorder;`	Pb	Определяет способ отображения границы панели
`property PanelCount: Integer;`	Ro	Содержит число видимых одновременно панелей
`property PanelHeight: Integer;`	Pb	Определяет высоту панелей в пикселах
`property PanelIndex: Integer;`	Pu	Определяет индекс панели текущей записи
`property PanelWidth: Integer;`	Pb	Определяет ширину панелей в пикселах
`property RowCount: Integer;`	Pb	Определяет число строк видимых панелей
`property SelectedColor: TColor;`	Pb	Определяет фоновый цвет панели текущей записи
`property ShowFocus: Boolean;`	Pb	Разрешает или запрещает выделение вокруг панели текущей записи

Таблица 15.5 *(окончание)*

Объявление	Тип	Описание
Методы		
`type TDBCtrlGridKey = (gkNull, gkEditMode, gkPriorTab, gkNextTab, gkLeft, gkRight, gkUp, gkDown, gkScrollUp, gkScrollDown, gkPageUp, gkPageDown, gkHome, gkEnd, gkInsert, gkAppend, gkDelete, gkCancel);` `procedure DoKey(Key: TDBCtrlGridKey);`	—	Выполняет операцию, заданную при помощи параметра `Key`. Доступны операции навигации по записям, перевода в режим редактирования, вставки, удаления записей, отмены изменений
`procedure KeyDown(var Key: Word; Shift: TShiftState); override;`	—	Используется при нажатии клавиши для трансляции кодов клавиш
Методы-обработчики событий		
`type TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid; Index: Integer) of object;` `property OnPaintPanel: TPaintPanelEvent;`	—	Вызывается при перерисовке панели. Параметр `Index` соответствует индексу панели

Навигация по набору данных

Перемещение или навигация по записям набора данных может осуществляться несколькими путями. Например, в компонентах `TDBGrid` и `TDBCtrlGrid`, которые отображают сразу несколько записей набора данных, можно использовать клавиши вертикального перемещения курсора или вертикальную полосу прокрутки.

Но что делать, если на форме находятся только компоненты, отображающие одно поле только текущей записи набора данных (`TDBEdit`, `TDBComboBox` и т. д.)? Очевидно, что в этом случае на форме должны быть расположены дополнительные элементы управления, отвечающие за перемещение по записям.

Аналогично, ни один компонент отображения данных не имеет встроенных средств для создания и удаления записей целиком.

Для решения указанных задач и предназначен компонент TDBNavigator, который представляет собой совокупность управляющих кнопок, выполняет операции навигации по набору данных и модификации записей целиком.

Компонент TDBNavigator при помощи свойства DataSource связывается с компонентом TDataSource и через него с набором данных. Такая схема позволяет обеспечить изменение текущих значений полей сразу во всех связанных с TDataSource компонентах отображения данных. Таким образом, TDBNavigator только дает команду на выполнение перемещения по набору данных или другой управляющей операции, а всю реальную работу выполняют компонент набора данных и компонент TDataSource. Компонентам отображения данных остается только принять новые данные от своих полей.

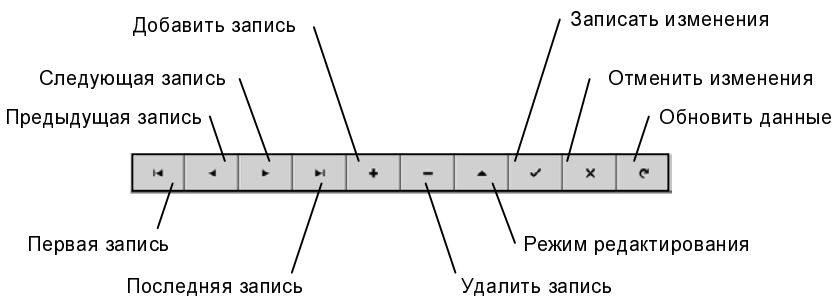


Рис. 15.4. Назначение кнопок компонента TDBNavigator

Компонент TDBNavigator содержит набор кнопок, каждая из которых отвечает за выполнение одной операции над набором данных. Всего имеется 10 кнопок, разработчик может оставить в наборе любое количество кнопок в любом сочетании. Видимостью кнопок управляет свойство VisibleButtons:

```
type
  TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete,
nbEdit, nbPost, nbCancel, nbRefresh);
  TButtonSet = set of TNavigateBtn;
property VisibleButtons: TButtonSet;
```

Каждый элемент типа TNavigateBtn представляет одну кнопку, их назначение описывается ниже:

nbFirst — перемещение на первую запись набора данных;

nbPrior — перемещение на предыдущую запись набора данных;

nbNext — перемещение на следующую запись набора данных;

nbLast — перемещение на последнюю запись набора данных;

nbInsert — вставка новой записи в текущей позиции набора данных;

`nbDelete` — удаление текущей записи, курсор перемещается на следующую запись;

`nbEdit` — набор данных переводится в режим редактирования;

`nbPost` — в базу данных переносятся все изменения в текущей записи;

`nbCancel` — все изменения в текущей записи отменяются;

`nbRefresh` — восстанавливаются первоначальные значения текущей записи, сделанные после последнего переноса изменений в базу данных.

Самой критичной к возможной потере данных вследствие ошибки является операция удаления записи, поэтому при помощи свойства `ConfirmDelete` можно включить механизм контроля удаления. При каждом удалении записи нужно будет дать подтверждение выполняемой операции.

Нажатие любой кнопки можно эмулировать программно при помощи метода `BtnClick`.

В случае необходимости выполнения дополнительных действий при щелчке на любой кнопке можно воспользоваться обработчиками событий `BeforeAction` и `OnClick`, в которых параметр `Button` определяет нажатую кнопку. Свойства и методы компонента `TDBNavigator` представлены в табл. 15.6.

Таблица 15.6. *Свойства и методы компонента* `TDBNavigator`

Объявление	Тип	Описание
Свойства		
`property ConfirmDelete: Boolean;`	Pb	Включает или отключает подтверждение удаления записи
`property Hints: TStrings;`	Pb	Содержит список подсказок для каждой кнопки
`property Flat: Boolean;`	Pb	Определяет внешний вид кнопок компонента

`type`
`TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost, nbCancel, nbRefresh);`
`TButtonSet = set of TNavigateBtn;`
`property VisibleButtons: TButtonSet;`
	Pb	Список видимых кнопок

Таблица 15.6 *(окончание)*

Объявление	Тип	Описание
Методы		
`procedure BtnClick(Index: TNavigateBtn);`	Pu	Эмулирует щелчок на кнопке `Index`
`procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);`	Pu	Задает положение (параметры `ALeft`, `ATop`) и размер компонента (параметры `AWidth`, `AHeight`)
Методы-обработчики событий		
`ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;` `property BeforeAction: ENavClick;`	Pb	Выполняется при щелчке на кнопке `Button` перед выполнением операции, связанной с кнопкой
`ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;` `property OnClick: ENavClick;`	Pb	Выполняется при щелчке на кнопке `Button` после выполнения операции, связанной с кнопкой

Представление отдельных полей

Большинство компонентов отображения данных предназначено для представления данных из отдельных полей. Для этого все они имеют свойство `DataField`, которое указывает на требуемое поле набора данных.

В зависимости от типа данных поля могут использовать различные компоненты. Для большинства стандартных полей используются компоненты `TDBText`, `TDBEdit`, `TDBComboBox`, `TDBListBox`.

Данные в формате Memo отображаются компонентами `TDBMemo` и `TDBRichEdit`.

Для показа изображений предназначен компонент `TDBImage`.

Компонент *TDBText*

Этот компонент представляет собой статический текст, который отображает текущее значение некоторого поля связанного набора данных. При этом данные можно просматривать в режиме "только для чтения".

Непосредственным предком компонента является класс `TCustomLabel`, поэтому он очень похож на компонент `TLabel`.

При использовании компонента следует обратить внимание на возможную длину отображаемых данных. Для предотвращения обрезания текста можно использовать свойства `AutoSize` и `WordWrap`.

Компонент *TDBEdit*

Компонент представляет собой стандартный однострочный текстовый редактор, в котором отображаются и изменяются данные из поля связанного набора данных.

Прямой предок компонента — класс `TCustomMaskEdit`, который также является прямым предком компонента `TEdit`.

Компонент может осуществлять проверку редактируемых данных по заданной для поля маске. Непосредственно для редактора задать маску нельзя, т. к. содержащее маску свойство `EditMask` в классе `TCustomMaskEdit` является защищенным, а в `TDBEdit` не перекрыто. Тем не менее механизм контроля полностью унаследован. Саму же маску можно задать в связанном с редактором поле. Объект `TField` имеет собственное свойство `EditMask`, которое и используется при проверке данных в редакторе *(см. гл. 13)*.

Проверка редактируемого текста на соответствие маске осуществляется методом `ValidateEdit` после каждого введенного или измененного символа. В случае ошибки генерируется исключение `ValidateError` и курсор устанавливается на первый ошибочный символ.

В компоненте можно использовать буфер обмена. Это делается средствами операционной системы пользователем или программно при помощи методов `CopyToClipboard`, `CutToClipboard`, `PasteFromClipboard`.

Компонент *TDBCheckBox*

Компонент представляет собой почти полный аналог обычного флажка (компонент `TCheckBox`) и предназначен для отображения и редактирования любых данных, которые могут иметь только два значения. Это может быть логический тип данных или любые строковые значения, но поле может принимать значения только из двух строк.

Предопределенные значения задаются свойствами `ValueChecked` и `ValueUnchecked`. По умолчанию они имеют значения `True` и `False`. Этим свойствам можно также присваивать любые строковые значения, причем одному свойству можно назначить несколько возможных значений, разделенных точкой с запятой.

Включение флажка происходит, если значение поля набора данных совпадает со значением свойства `ValueChecked` (единственным или любым из списка). Если же флажок включил пользователь, то значение поля данных при-

равнивается к единственному или первому в списке значению свойства `ValueChecked`.

Аналогичные действия происходят и со свойством `ValueUnchecked`.

Компонент *TDBRadioGroup*

Компонент представляет собой стандартную группу переключателей, состояние которых зависит от значений поля связанного набора данных. В поле можно передавать фиксированные значения, связанные с отдельными переключателями в группе.

Если текущее значение связанного поля соответствует значению какого-либо переключателя, то он включается. Если пользователь включает другой переключатель, то связанное с переключателем значение заносится в поле.

Возможные значения, на которые должны реагировать переключатели в группе, заносятся в свойство `Values` при помощи специального редактора в Инспекторе объектов или программно посредством методов класса `TStrings`. Каждому элементу свойства `Values` соответствует один переключатель (порядок следования сохраняется).

Свойство `Items` содержит список поясняющих надписей для переключателей группы. Если для какого-либо переключателя нет заданного значения, но есть поясняющий текст, то такой переключатель включается при совпадении значения связанного поля с поясняющим текстом.

Текущее значение связанного поля содержится в поле `Value`.

Компонент *TDBListBox*

Компонент отображает текущее значение связанного с ним поля набора данных и позволяет изменить его на любое фиксированное из списка. Функционально компонент ничем не отличается от компонента `TListBox`. Значение поля должно совпадать с одним из элементов списка.

Специальных методов компонент не содержит.

Компонент *TDBComboBox*

Компонент отображает текущее значение связанного с ним поля набора данных в строке редактирования, при этом значение поля должно совпадать с одним из элементов разворачивающегося списка. Текущее значение можно изменить на любое фиксированное из списка компонента. Функционально компонент ничем не отличается от компонента `TComboBox`, представляющего собой комбинированный список.

Компонент может работать в пяти различных стилях, которые определяются свойством `Style`.

Специальных методов компонент не содержит.

Компонент *TDBMemo*

Компонент представляет собой обычное поле редактирования, к которому подключается поле с типом данных Memo или BLOB. Основное его преимущество — возможность одновременного просмотра и редактирования нескольких строк переменной длины. Компонент может отображать только строки, которые целиком видны по высоте.

В компоненте можно использовать буфер обмена при помощи стандартных средств операционной системы или унаследованными от предка `TCustomMemo` методами `CopyToClipBoard`, `CutToClipBoard`, `PasteFromClipBoard`.

Для ускорения навигации по набору данных при отображении полей типа `BLOB` можно использовать свойство `AutoDisplay`. При значении `True` любое новое значение поля автоматически отображается в компоненте. При значении `False` новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Метод `LoadMemo` используется автоматически при загрузке значения поля, если свойство `AutoDisplay = False`.

Поведением компонента при работе со слишком длинными строками можно управлять при помощи свойства `WordWrap`. При значении `True` слишком длинная строка сдвигается влево при перемещении текстового курсора за правую границу компонента. При значении `False` остаток длинной строки переносится на новую строку, при этом реально новая строка в данных не создается.

Компонент *TDBImage*

Компонент предназначен для просмотра изображений, хранящихся в базах данных в графическом формате.

Редактировать изображения можно только в каком-либо графическом редакторе, перенося исходное и измененное изображение при помощи буфера обмена. Это делается средствами операционной системы пользователем или программно при помощи методов `CopyToClipboard`, `CutToClipboard`, `PasteFromClipboard`.

Визуализация изображения осуществляется при помощи свойства `Picture`, которое представляет собой экземпляр класса `TPicture`.

Также можно полностью заменить существующее изображение или сохранить новое в новой записи набора данных. Для этого используются методы свойства `Picture`.

Свойство `AutoDisplay` позволяет управлять процессом загрузки новых изображений из набора данных в компонент. При значении `True` любое новое значение поля автоматически отображается в компоненте. При значении `False` новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Для ускорения просмотра изображений можно применять свойство `QuickDraw`, которое задает используемую изображением палитру. При значении `True` применяется стандартная системная палитра. В результате уменьшается время загрузки изображения, но может ухудшиться и качество изображения, в некоторых случаях до полного искажения. При значении `False` используется собственная палитра изображения и процесс загрузки замедляется.

Компонент *TDBRichEdit*

Компонент предоставляет возможности полноценного текстового редактора для просмотра и изменения текстовых данных, хранящихся в связанном поле набора данных. Поле должно содержать информацию о форматировании текста.

Внешне компонент ничем не отличается от поля редактирования, поэтому о реализации доступа к гораздо более богатым возможностям редактора через интерфейс пользователя должен позаботиться разработчик. Для этого можно использовать дополнительные элементы управления.

Синхронный просмотр данных

При разработке приложений для работы с базами данных часто возникает необходимость в связывании двух наборов данных по ключевому полю. Например, в таблице Orders (содержит данные о заказах) демонстрационной базы данных DBDEMOS имеется поле `CustNo`, которое содержит идентификационный номер покупателя. Под этим же номером в таблице Customers хранится информация о покупателе (адрес, телефон, реквизиты и т. д.). При разработке пользовательского интерфейса приложения баз данных необходимо, чтобы при просмотре перечня заказов в форме приложения отображались не идентификационные номера покупателей, а их параметры.

Таким образом, в наборе данных заказов вместо поля номера покупателя должно появиться поле имени покупателя из таблицы Customers. Механизм связывания полей из различных наборов данных по ключевому полю называется *синхронным просмотром*. В рассмотренном примере ключевым является поле `CustNo` из таблицы Customers, а выбор конкретного наименования производится по совпадению значений ключевого поля и заменяемого поля

из исходного набора данных — Orders. Причем необходимо, чтобы в таблице Customers поле `CustNo` было уникальным (составляло первичный или вторичный ключ).

Таблицу, в которой расположено поле, значения которого замещаются на синхронные, будем называть *исходной таблицей* (это таблица Orders).

Таблицу, содержащую ключевое поле и поле данных для синхронного просмотра, будем называть *таблицей синхронного просмотра* (таблица Customers).

В Delphi механизм синхронного просмотра реализован на уровне отдельных полей и компонентов.

В наборе данных динамически можно создать специальное поле синхронного просмотра, которое будет автоматически замещать одно значение другим в зависимости от значения ключевого поля. Такое поле можно связать с любым рассмотренным выше компонентом отображения данных *(см. гл. 13)*.

Помимо простого синхронного просмотра данных может возникнуть задача редактирования данных в аналогичной ситуации. Для этого предназначены специальные компоненты синхронного просмотра данных, которые позволяют, например, выбирать покупателя из списка, а изменится при этом номер покупателя в наборе данных заказов. Использование таких компонентов делает пользовательский интерфейс значительно более удобным и наглядным. В VLC Delphi есть два таких компонента: `TDBLookupListBox` и `TDBLookupComboBox`.

Примечание

На странице **Win 3.1** Палитры компонентов имеются еще два компонента: `TDBLokupList` и `TDBLookupCombo`. Они обладают тем же набором функций, используются для обеспечения совместимости с приложениями, созданными в среде разработки Delphi 1, и поэтому здесь не рассматриваются.

Механизм синхронного просмотра

Непосредственным предком компонентов синхронного просмотра данных является класс `TDBLookupControl`, который инкапсулирует список значений для просмотра и сам механизм синхронного просмотра.

Как и в любом другом компоненте отображения данных, в компонентах синхронного просмотра должны присутствовать средства связывания с требуемым полем некоторого набора данных (табл. 15.7). Это уже известные свойства: `DataSource` — применяется для задания набора данных через компонент `TDataSource` и `DataField` — для определения требуемого поля набора данных. Для синхронного просмотра следует выбирать такое поле, значения которого не дают пользователю полной информации об объекте и совпадают с ключевым полем в таблице синхронного просмотра. Название этого

поля может не совпадать с названием ключевого поля, но типы данных должны быть одинаковыми.

Примечание

При проектировании баз данных желательно, чтобы такие поля все же носили одинаковые названия.

Теперь необходимо задать таблицу синхронного просмотра, ключевое поле и поле синхронного просмотра.

Набор данных, содержащий указанные поля, определяется через соответствующий компонент `TDataSource` в свойстве `ListSource`.

Ключевое поле задается свойством `KeyField`. Во время работы компонента в свойстве `KeyValue` содержится текущее значение, которое связывает между собой два набора данных.

Поле синхронного просмотра определяется свойством `ListField`. Здесь можно задавать сразу несколько полей, которые будут отображаться в компоненте синхронного просмотра. Названия полей разделяются точкой с запятой. Если свойство не определено, то в компоненте будут отображаться значения ключевого поля. Свойство `ListFieldIndex` служит для выбора основного поля из списка. Дело в том, что компоненты синхронного просмотра поддерживают механизм наращиваемого поиска, который позволяет быстро находить нужное значение в больших списках. Свойство `ListFieldIndex` определяет, какое поле используется при наращиваемом поиске. В компоненте `TDBlookupComboBox` свойство `ListFieldIndex` также определяет, какое поле будет передано в строку редактирования.

Таблица 15.7. *Основные свойства, включающие механизм синхронного просмотра*

Объявление	Тип	Описание
`property KeyField: string;`	Pb	Ключевое поле таблицы синхронного просмотра
`property KeyValue: Variant;`	Pu	Текущее значение ключевого поля
`property ListField: string;`	Pb	Поле или список полей синхронного просмотра в таблице синхронного просмотра
`property ListFieldIndex: Integer;`	Pb	Номер основного поля синхронного просмотра (используется, когда свойство `ListField` содержит список полей)
`property ListSource: TDataSource;`	Pb	Указывает на компонент `TDataSource`, связанный с таблицей синхронного просмотра

Таблица 15.7 *(окончание)*

Объявление	Тип	Описание
`property NullValueKey: TShortCut;`	`Pb`	Определяет комбинацию клавиш, нажатие которых задает нулевое значение поля

В качестве примера рассмотрим приложение DemoLookup (рис. 15.5), в котором с набором данных таблицы Orders из базы данных DBDEMOS связаны компоненты `TDBGrid` и `TDBLookupComboBox`. Во втором компоненте при перемещении по записям набора данных отображается имя покупателя, оформившего текущий заказ.

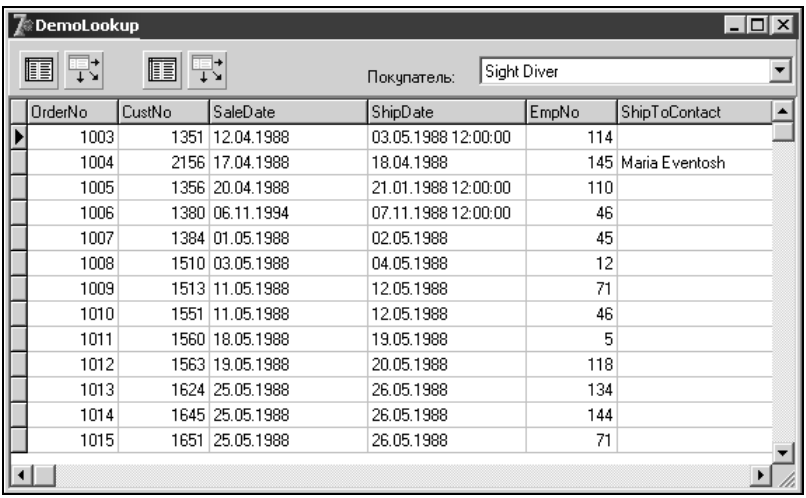


Рис. 15.5. Главная форма проекта DemoLookup

Ключевые свойства компонента настроены следующим образом.

Свойство `ListSource` указывает на компонент `CustSource` типа `TDataSource`, который связан с набором данных синхронного просмотра `CustTable`.

Свойство `ListField` указывает на поле `Company`, все значения которого доступны в списке компонента.

Свойство `KeyField` указывает на поле `CustNo`, которое имеется в двух таблицах и по которому осуществляется связь.

Рассмотрим основные свойства и методы самих компонентов отображения данных, за исключением тех, которые представлены в табл. 15.7 и полностью идентичны для двух компонентов.

Компонент *TDBLookupListBox*

Компонент представляет собой список значений поля синхронного просмотра для поля, заданного свойством `DataField`, из набора данных `DataSource`. Его основное назначение — автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом `TDBListBox`.

Компонент *TDBLookupComboBox*

Компонент представляет собой комбинированный список значений поля синхронного просмотра для поля, заданного свойством `DataField`, из набора данных `DataSource`. Его основное назначение — автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом `TDBComboBox`.

Графическое представление данных

Для представления данных из некоторого набора данных в виде графиков различных видов предназначен компонент `TDBChart` (табл. 15.8). В нем можно одновременно показывать графики для нескольких полей данных. Графики строятся на основе всех имеющихся в наборе данных значений полей. Функционально компонент ничем не отличается от компонента `TChart`.

Настройка параметров компонента осуществляется специальным редактором, который можно открыть двойным щелчком на перенесенном на форму компоненте.

Здесь мы не будем подробно останавливаться на богатейших изобразительных возможностях этого компонента, рассмотрим только процесс подключения к нему набора данных и построение графиков.

Основой любого графика в компоненте `TDBChart` является так называемая серия, свойства которой представлены классом `TchartSeries`. Для того чтобы построить график значений некоторого поля набора данных, необходимо

выполнить следующие действия, большинство из которых выполняется в специализированном редакторе компонента.

1. Создать новую серию и определить ее тип.
2. Задать для серии набор данных.
3. Связать с осями координат нужные поля набора данных и, в зависимости от типа серии, задать дополнительные параметры.
4. Открыть набор данных.

Редактор имеет две главные страницы — **Chart** и **Series**. Страница **Chart** содержит многостраничный блокнот и предназначена для настройки параметров самого графика. Страница **Series** также содержит многостраничный блокнот и используется для настройки серий значений данных.

Для создания новой серии необходимо в редакторе перейти на главную страницу **Chart**, а на ней открыть страницу **Series** (рис. 15.6). На этой странице нужно щелкнуть на кнопке **Add**, а затем в появившемся диалоге выбрать тип серии. После этого в списке на странице **Series** появляется строка новой серии. Здесь можно переопределить тип, цвет и видимость серии, щелкнув на соответствующей зоне строки.

Все остальные страницы блокнота на главной странице **Chart** предназначены для настройки параметров графика.

Теперь необходимо перейти на главную страницу **Series** и на ней из списка названий серий выбрать необходимую. После этого на странице **Data Source** из списка выбирается строка **DataSet**. Далее в появившемся списке **DataSet** выбирается нужный набор данных.

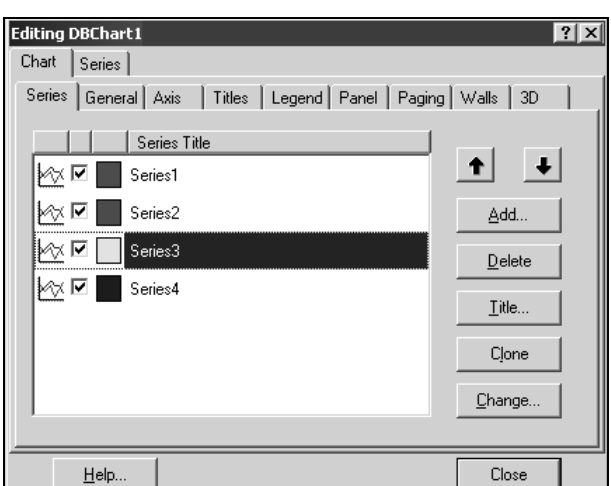


Рис. 15.6. Специализированный редактор компонента `TDBChart`

Список **X** позволяет выбрать поле набора данных, значения которого будут последовательно откладываться по оси абсцисс. Список **Y** позволяет выбрать поле набора данных, значения которого будут отложены по оси ординат. Соответствие между значениями полей по двум осям определяется принадлежностью к одной записи набора данных. Выбор поля в списке **Labels** привязывает его значения в виде меток к оси абсцисс.

Примечание

Здесь описан набор элементов управления для линейного типа серии. Для других типов элементы управления могут отличаться.

Теперь осталось только открыть набор данных и компонент `TDBChart` построит график.

Аналогичным образом на этот же компонент можно поместить и другие графики.

Таблица 15.8. *Свойства и методы компонента* `TDBChart`

Объявление	Описание
Свойства	
`property AutoRefresh : Boolean;`	Разрешает или запрещает обновление данных в серии при открытии связанного набора данных
`property RefreshInterval : LongInt;`	Задает временной интервал в секундах между обновлениями данных в сериях из связанных наборов данных
`property ShowGlassCursor : Boolean;`	Разрешает показ курсора "песочные часы" при обновлении данных
Методы	
`procedure CheckDataSource;`	Обновляет данные в сериях
`function IsValidDataSource(ASeries: TChartSeries; AComponent: TComponent): Boolean; virtual;`	Проверяет, связан ли набор данных `AComponent` с серией `ASeries`. В случае успеха проверки возвращает `True`
`procedure RefreshData;`	Обновляет данные во всех сериях
`procedure RefreshDataSet(ADataSet: TDataSet; ASeries: TChartSeries);`	Считывает все записи в наборе данных `AdataSet` и переносит их в серию `ASeries`
Методы-обработчики событий	
`property OnProcessRecord : TProcessRecordEvent;`	Вызывается при переносе данных из отдельной записи набора данных в серию

Резюме

Компоненты отображения данных играют важную роль при создании интерфейсов приложений баз данных. Разнообразие предлагаемых элементов управления позволяет решать любые задачи по организации взаимодействия пользователя с базой данных. Все они взаимодействуют с набором данных через компонент `TDataSource`.

ЧАСТЬ IV

Технологии доступа к данным

Глава 16. Процессор баз данных Borland Database Engine

Глава 17. Технология dbExpress

Глава 18. Сервер баз данных InterBase и компоненты InterBase Express

Глава 19. Использование ADO средствами Delphi

ГЛАВА 16

Процессор баз данных Borland Database Engine

Любое приложение баз данных имеет в своем составе или использует сторонний механизм доступа к данным, который берет на себя подавляющее большинство стандартных низкоуровневых операций работы с базами данных. Например, любое такое приложение при открытии таблицы БД должно выполнить примерно одинаковый набор операций.

- поиск местоположения базы данных;
- поиск таблицы, ее открытие и чтение служебной информации;
- чтение данных в соответствии с форматом хранения данных

  и т. д.

Очевидно, что если все стандартные функции доступа к данным реализовать в виде специальной программы, сервиса или динамической библиотеки, то это существенно упростит разработку приложений баз данных, которым для выполнения той или иной операции достаточно будет вызвать готовую процедуру.

Одним из традиционных способов взаимодействия приложения, созданного в среде разработки Delphi, и базы данных является использование процессора баз данных Borland Database Engine 5. Он представляет собой набор динамических библиотек, функции которых позволяют не только обращаться к данным, но и эффективно управлять ими на стороне приложения.

Для работы с источниками данных при посредстве BDE в Delphi имеется специальный набор компонентов, расположенных на странице BDE Палитры компонентов. Эти компоненты для работы с базами данных используют возможности BDE, обращаясь к его функциям и процедурам. Механизм доступа к BDE инкапсулирован в базовом классе `TBDEDataSet`. (Подробно базовые классы компонентов доступа к данным рассмотрены далее в этой части.) Поэтому в процессе программирования у вас не будет необходи-

мости использовать функции BDE напрямую. Почти все, что можно сделать путем прямого обращения, можно сделать и через компоненты — это проще и надежнее.

Тем не менее внутреннюю организацию механизма доступа к данным всегда полезно знать. Кроме этого, всегда полезно знать и уметь использовать дополнительные возможности, которые BDE может предоставить разработчику.

BDE взаимодействует с базами данных при посредстве драйверов. Для особенно распространенных локальных СУБД разработан набор стандартных драйверов. Работа с наиболее распространенными серверами БД осуществляется при помощи драйверов системы SQL Links. Кроме этого, если для базы данных существует драйвер ODBC, то можно использовать и его. Достаточно зарегистрировать этот драйвер в BDE.

Однако BDE не претендует на всеобъемлющую универсальность и имеет некоторые недостатки. Это, например, снижение скорости работы приложения, недостатки реализации некоторых драйверов и т. д. В документации к Delphi 7 содержится предупреждение, что после 2002 года фирма Borland перестанет поддерживать BDE и рекомендует использовать технологию dbExpress, которая также рассматривается в настоящей книге.

В этой главе обсуждаются следующие вопросы.

- архитектура и составные части BDE;
- что такое псевдонимы БД и настройка драйверов BDE;
- использование утилиты BDE Administrator;
- способы прямого использования функций API BDE;
- компоненты доступа к данным BDE.

Архитектура и функции BDE

BDE представляет собой набор динамических библиотек, которые "умеют" передавать запросы на получение или модификацию данных из приложения в нужную базу данных и возвращать результат обработки. В процессе работы библиотеки используют вспомогательные файлы языковой поддержки и информацию о настройках среды.

В составе BDE поставляются стандартные драйверы, обеспечивающие доступ к СУБД Paradox, dBASE, FoxPro и текстовым файлам. Локальные драйверы (рис. 16.1) устанавливаются автоматически совместно с ядром процессора. Один из них можно выбрать в качестве стандартного драйвера, который имеет дополнительные настройки, влияющие на функционирование процессора БД.

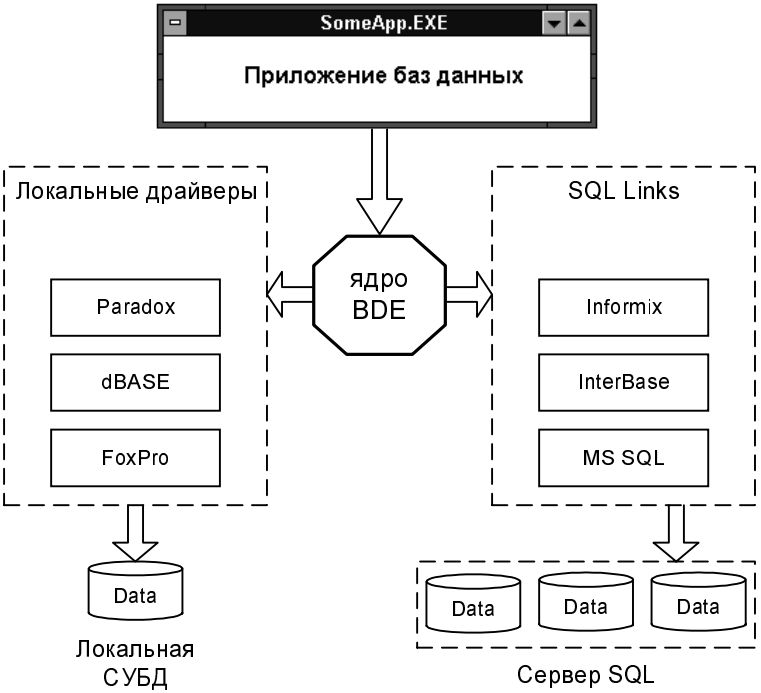


Рис. 16.1. Структура процессора баз данных BDE

Доступ к данным серверов SQL обеспечивает отдельная система драйверов — SQL Links. С их помощью в Delphi можно без особых проблем разрабатывать приложения для серверов Oracle 8, Informix, Sybase, DB2 и, естественно, InterBase. Эти драйверы необходимо устанавливать дополнительно.

Помимо этого, в BDE имеется очень простой механизм подключения любых драйверов ODBC (к примеру, Microsoft Access) и создания на их основе сокетов ODBC.

Примечание

С точки зрения пользователя процесс подключения локального драйвера и драйвера SQL Links практически не отличается, за исключением деталей настройки. Настройка драйверов и собственных параметров BDE осуществляется при помощи специальной утилиты — BDE Administrator и рассматривается далее в этой главе.

В состав BDE входят следующие функциональные подсистемы.

- *Администратор системных ресурсов* управляет процессом подключения к данным — при необходимости устанавливает нужные драйверы, а при завершении работы автоматически освобождает занятые ресурсы. Поэтому BDE всегда использует ровно столько ресурсов, сколько необходимо.

- *Система обработки запросов* обеспечивает выполнение запросов SQL или QBE от приложения к любым базам данных, для которых установлен драйвер, даже если сама СУБД не поддерживает прямое использование запросов SQL.
- *Система сортировки* является запатентованной технологией и обеспечивает очень быстрый поиск по запросам SQL и через стандартные драйверы для Paradox и dBASE.
- *Система пакетной обработки* представляет собой механизм преобразования данных из одного формата в другой при выполнении операций над целыми таблицами. Эта система использована в качестве основы для компонента `TBatchMove` и утилиты DataPump (автоматического переноса структур данных между базами данных), входящей в стандартную поставку BDE.
- *Менеджер буфера* управляет единой для всех драйверов буферной областью памяти, которую одновременно могут использовать несколько драйверов. Это позволяет существенно экономить системные ресурсы.
- *Менеджер памяти* взаимодействует с ОС и обеспечивает эффективное использование выделяемой памяти. Ускоряет работу драйверов, которые для получения небольших фрагментов памяти обращаются к нему, а не к ОС. Дело в том, что менеджер памяти выделяет большие объемы оперативной памяти и затем распределяет ее небольшими кусками между драйверами согласно их потребностям.
- *Транслятор данных* обеспечивает преобразование форматов данных для различных типов БД.
- *Кэш BLOB* используется для ускорения работы с данными в формате BLOB.
- *SQL-генератор* транслирует запросы в формате QBE в запросы SQL.
- *Система реструктуризации* обеспечивает преобразование наборов данных в таблицы Paradox или dBASE.
- *Система поддержки драйверов SQL* повышает эффективность механизма поиска при выполнении запросов SQL.
- *Таблицы в памяти.* Этот механизм позволяет создавать таблицы непосредственно в оперативной памяти. Используется для ускорения обработки больших массивов данных, сортировки, преобразования форматов данных.
- *Связанные курсоры* обеспечивают низкоуровневое выполнение межтабличных соединений. Позволяют разработчику не задумываться над реализацией подобных связей при работе на уровне VCL — для этого достаточно установить значения нескольких свойств.

- *Менеджер конфигурации* обеспечивает разработчику доступ к информации о конфигурации драйверов.

Перечисленные функции реализованы в динамических библиотеках, которые, собственно, и называются процессором БД (табл. 16.1).

Таблица 16.1. *Ядро процессора баз данных BDE 5*

Имя файла	Назначение
IDAPI32.DLL	Базовая динамическая библиотека BDE
IDPROV.DLL	Динамическая библиотека, отвечающая за работу серверной части приложения
BLW32.DLL	Динамическая библиотека, обеспечивающая поддержку драйверов национальных языков
IDBAT32.DLL	Динамическая библиотека с функциями межтабличного переноса данных
IDQBE32.DLL	Динамическая библиотека, обеспечивающая работу *запросов по примеру* (Query By Example, QBE)
IDSQL32.DLL	Динамическая библиотека, обеспечивающая обработку запросов SQL
IDASCI32.DLL	Динамическая библиотека, обеспечивающая работу драйвера текстовых файлов
IDPDX32.DLL	Динамическая библиотека, обеспечивающая работу драйвера Paradox
IDDBAS32.DLL	Динамическая библиотека, обеспечивающая работу драйвера dBASE
DODBC32.DLL	Динамическая библиотека, обеспечивающая работу драйвера сокета ODBC
IDR20009.DLL	Динамическая библиотека ресурсов, содержащая сообщения об ошибках
IDDAO32.DLL	Динамическая библиотека, обеспечивающая работу драйверов Microsoft Access 95 и Jet Engine 3.0
IDDA3532.DLL	Динамическая библиотека, обеспечивающая работу драйверов Microsoft Access 97 и Jet Engine 3.5
IDDR32.DLL	Динамическая библиотека для работы с Репозиторием данных

Кроме этого имеется шесть дополнительных DLL, обеспечивающих работу BDE с серверами Oracle и Microsoft SQL Server.

Псевдонимы баз данных и настройка BDE

Для успешного доступа к данным приложение и BDE должны обладать информацией о местоположении файлов требуемой базы данных. Задание маршрута входит в обязанности разработчика.

Самый простой способ заключается в явном задании полного пути к каталогу, в котором хранятся файлы БД. Но в случае изменения пути, что случается не так уж редко, например, при переносе готового приложения на компьютер заказчика, разработчик должен перекомпилировать проект с учетом будущего местонахождения БД или предусмотреть специальные элементы управления, в которых можно задать путь к БД.

Для решения такого рода проблем разработчик может использовать *псевдоним* базы данных, который представляет собой именованную структуру, содержащую путь к файлам БД и некоторые дополнительные параметры. В первом приближении можно сказать, что вы просто присваиваете маршруту произвольное имя, которое используется в приложении. Тогда при переносе приложения на компьютере заказчика достаточно создать стандартными средствами BDE одноименный псевдоним и настроить его на нужный каталог. При этом само приложение не требует переделок, т. к. оно обращается к псевдониму с одним именем, а вот BDE уже "знает" куда отправить запрос приложения, использовавшего этот псевдоним.

Помимо маршрута к файлам базы данных, псевдоним BDE обязательно содержит информацию о драйвере БД, который используется для доступа к данным. Наличие других параметров зависит от типа драйвера, а значит, от типа СУБД.

Для управления псевдонимами баз данных, настройки стандартных и дополнительных драйверов в составе BDE имеется специальная утилита — BDE Administrator (исполняемый файл BDEADMIN.EXE).

Стандартная конфигурация BDE сохраняется в файле IDAPI.CFG. При необходимости текущую конфигурацию можно сохранить в новом файле с расширением cfg или загрузить заново при помощи команд **Save As Configuration** и **Open Configuration** из меню **Object**.

В верхней части окна утилиты расположена Панель инструментов, кнопки которой используются при работе с конкретным элементом настройки BDE.

Рабочая область утилиты BDE Administrator представляет собой двухстраничный блокнот.

Страница **Databases** (рис. 16.2) содержит иерархическое дерево, в узлах которого расположены установленные в системе на данный момент псевдонимы БД. При выборе какого-либо псевдонима в правой части панели появляется путь к файлам базы данных и перечень параметров драйвера, соответствующего псевдониму, которые можно настраивать вручную.

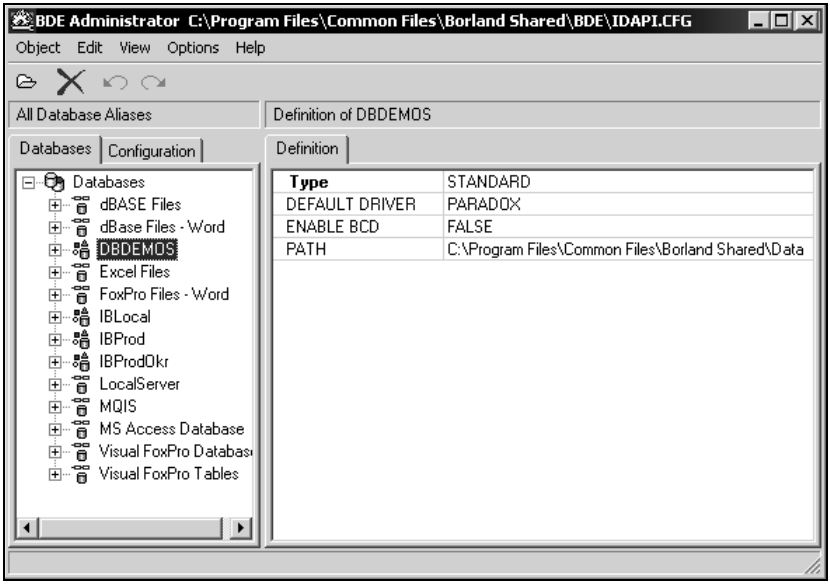


Рис. 16.2. Окно утилиты BDE Administrator с открытой страницей **Databases**

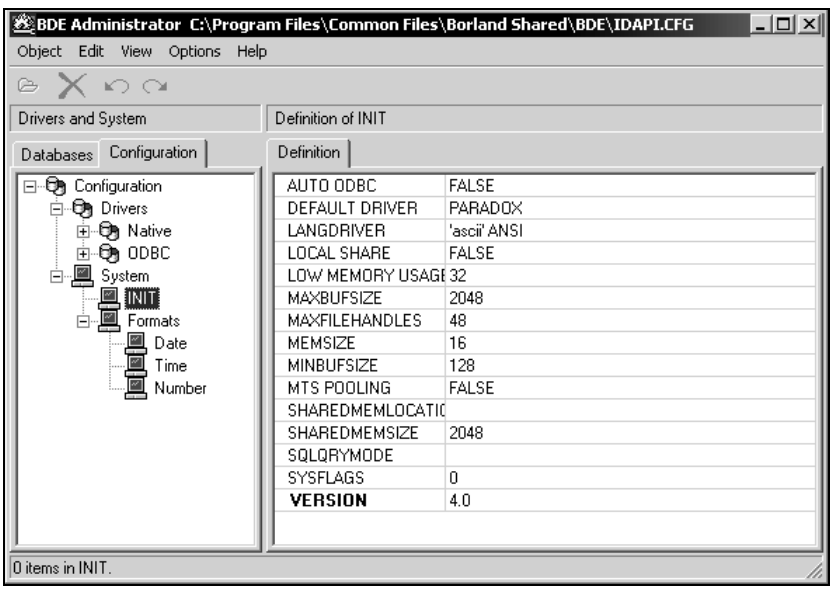


Рис. 16.3. Окно утилиты BDE Administrator с открытой страницей **Configuration**

Страница **Configuration** (рис. 16.3) используется для настройки параметров драйверов BDE, предназначенных для обеспечения доступа к локальным СУБД и серверам БД. Также здесь определяется системная конфигурация

BDE, которая включает параметры числовых форматов, форматов даты и времени. Вся информация на этой странице также структурирована в виде иерархического дерева.

При выборе в левой части панели утилиты какого-либо узла, в правой части на странице **Definition** отображается вся необходимая информация для этого объекта.

Сохранение изменений осуществляется при помощи команд меню **Object**, всплывающего меню или при перемещении на другой псевдоним.

Для создания нового псевдонима требуется выбрать команду **New** из меню **Object** или из всплывающего меню узла **Databases** на одноименной странице. Затем в появившемся простом диалоге задается необходимый драйвер.

Отметим, что один из четырех стандартных локальных драйверов устанавливается на странице **Configuration** в качестве предопределенного, поэтому в списке он доступен под названием STANDARD, а остальные не видны вообще. Из драйверов SQL Links доступны те, которые были установлены при инсталляции Delphi или позже.

Кроме того, в списке можно выбрать один из драйверов ODBC, установка которых осуществляется стандартными системными средствами на Панели управления Windows.

После выбора драйвера в дереве псевдонимов БД появляется новый узел, для драйвера которого требуется установить необходимые параметры *(см. ниже)*.

Для четырех локальных драйверов список параметров в правой части панели утилиты на странице **Definition** ограничивается параметрами стандартного драйвера (STANDARD), подробная настройка для каждого драйвера осуществляется на странице **Configuration**.

Назначение параметров локальных драйверов BDE (Paradox, dBASE, FoxPro, ASCII) представлено в табл. 16.2.

Таблица 16.2. *Параметры драйверов BDE для локальных баз данных*

Параметр	Назначение
STANDARD	
DEFAULT DRIVER	Задает тип конкретного локального драйвера (Paradox, dBASE, FoxPro, ASCII)
ENABLE BCD	Определяет способ представления вещественных чисел. При значении `True` такие числа преобразуются в формат BCD (Binary Coded Decimals — десятичные с двоичным кодированием). Точность составляет 20 знаков после запятой
PATH	Указывает путь к файлам базы данных

Таблица 16.2 *(продолжение)*

Параметр	Назначение
PARADOX	
NET DIR	Указывает путь к файлу обеспечения сетевого доступа к БД PDOXUSRS.NET. Драйвер приложения, которое работает с БД локально, должен указывать на этот файл, расположенный на том же компьютере. Драйвер приложения, обращающегося к БД по сети, должен указывать на подключенный сетевой диск с этим файлом
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для Paradox имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки (используйте драйвер Paradox Cyrr 866)
BLOCK SIZE	Задает размер блоков дискового пространства для хранения записей, кратно 1024
FILL FACTOR	Определяет процент заполнения блока дискового пространства при хранении индексов, по умолчанию 95%
LEVEL	Задает формат временной таблицы в памяти: • 3 — совместим с Paradox 3.5 и ниже; • 4 — Paradox 4.0; • 5 — Paradox 5.0; • 7 — Paradox для WIN32
STRICTINTEGRTY	Определяет возможность использования приложениями на базе Paradox 4.0 более поздних таблиц со ссылочной целостностью. При значении `True` использование разрешается, но возникает риск нарушения целостности данных
DBASE	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для dBASE имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки (используйте драйвер dBASE RUS cp866)
LEVEL	Задает формат таблиц. Значение соответствует номеру версии СУБД
MDX BLOCK SIZE	Размер блоков для файлов с расширением mdx, кратно 512
MEMO FILE BLOCK SIZE	Размер блоков для файлов с данными типа Memo (расширение dbt), кратно 512

Таблица 16.2 *(окончание)*

Параметр	Назначение
FOXPRO	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для FoxPro имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки
LEVEL	Имеет значение 25

Примечание

Драйвер текстовых файлов ASCIIDRV имеет параметры стандартного драйвера.

Назначение параметров драйверов SQL Links для серверов SQL представлено в табл. 16.3. Сначала приведены параметры, которые встречаются в двух и более драйверах, затем уникальные для каждого драйвера параметры. Драйверы для серверов InterBase и Sybase не представлены, т. к. содержат только общие для двух серверов параметры.

Таблица 16.3. *Параметры драйверов BDE для серверов SQL*

Параметр	Назначение
Общие параметры (встречаются как минимум у двух драйверов)	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Только для чтения
DLL	Название библиотеки динамического связывания SQL Links для 16-разрядного драйвера. Только для чтения
DLL32	Название библиотеки динамического связывания SQL Links для 32-разрядного драйвера. Только для чтения
DRIVER FLAGS	Используется только при необходимости применения старых версий драйвера, где не поддерживается уровень изоляции транзакций Read Committed. Для этого необходимо установить значение 512
TRACE MODE	Содержит битовую маску, которая определяет тип выдаваемой отладочной информации
BATCH COUNT	Задает число записей, модифицируемых в одном пакете при фиксации транзакций
BLOB SIZE	Размер кэша для данных типа BLOB. Диапазон от 32K до 1000K

Таблица 16.3 *(продолжение)*

Параметр	Назначение
BLOBS TO CACHE	Задает число кэшируемых записей с данными BLOB. Диапазон от 64 до 65 536
ENABLE BCD	Определяет способ представления вещественных чисел. При значении `True` такие числа преобразуются в формат BCD (Binary Coded Decimals — десятичные с двоичным кодированием), который позволяет округлять погрешности высших разрядов дробной части числа. Изменение параметра для псевдонима работает, только если параметр драйвера на странице **Configuration** не пустой
ENABLE SCHEMA CACHE	Определяет режим кэширования структуры данных. При значении `True` структура таблиц БД кэшируется локально в каталоге, задаваемом параметром SCHEMA CACHE DIR. Рекомендуется использовать только для баз данных с постоянной структурой
LANGDRIVER	Определяет драйвер языковой поддержки
MAX ROWS	Ограничивает максимальное число записей, которое может быть передано клиенту в ответ на запрос. Значение по умолчанию –1 (ограничений нет)
OPEN MODE	Режим работы с записями БД: • READ/WRITE — полный доступ; • READ ONLY — только чтение
SCHEMA CACHE DIR	Каталог для локального кэширования структуры данных (см. параметр ENABLE SCHEMA CACHE)
SCHEMA CACHE SIZE	Задает число таблиц, структура данных которых может кэшироваться
SCHEMA CACHE TIME	Задает время хранения кэшируемой структуры данных: • –1 — время не ограничено; • 0 — данные не кэшируются; • 1 — 2147483647 — секунды
SERVER NAME	Указывает путь к таблицам БД (это может быть локальный маршрут или маршрут с указанием удаленного сервера БД)
SQLPASSTHRU MODE	Задает способ использования соединения с сервером прямыми запросами SQL и запросами, управляемыми пользователем. SHARED AUTOCOMMIT — соединение используется совместно и прямые запросы фиксируются автоматически.

Таблица 16.3 *(продолжение)*

Параметр	Назначение
(прод.)	SHARED NO AUTOCOMMIT — соединение используется совместно и прямые запросы фиксируются сервером самостоятельно. NOT SHARED — совместное использование запрещено
SQLQRYMODE	Задает режим управления запросами. NULL — сначала запрос передается серверу, если тот не может обработать его, запрос выполняется локально. SERVER — запрос передается серверу. LOCAL — запрос выполняется локально
VENDOR INIT	Название файла динамической библиотеки поставщика
CONNECT TIMEOUT	Определяет временной интервал, после которого клиент попытается восстановить прерванную связь с сервером
TIMEOUT	Задает время ожидания ответа сервера на запрос
BLOB EDIT LOGGING	Управляет механизмом сохранения всех изменений для полей типа BLOB. При значении `True` изменения сохраняются
DATABASE NAME	Имя базы данных
MAX QUERY TIME	Задает максимальное время ожидания ответа на запрос
USER NAME	Имя пользователя, которое используется сервером при подключении
Microsoft SQL Server (MSSQL)	
MAX DBPROCESSES	Максимальное число процессов, одновременно работающих в данном соединении
APPLICATION NAME	Имя приложения, помогающее серверу идентифицировать процессы
DATE MODE	Определяет формат даты: • 0 — МДГ; • 1 — ДМГ; • 2 — ГМД
HOST NAME	Содержит имя рабочей станции. Помогает серверу при идентификации процессов
NATIONAL LANG NAME	Задает национальный язык, который используется для вывода текста в сообщениях об ошибках
TDS PACKET SIZE	Определяет размер пакетов потоков данных

Таблица 16.3 *(окончание)*

Параметр	Назначение
Oracle (ORACLE)	
NET PROTOCOL	Устанавливает сетевой протокол передачи данных
Informix (INFORMIX)	
DATE SEPARATOR	Задает разделитель для формата даты
Microsoft Access (MSACCESS)	
SYSTEM DATABASE	Путь к системной базе данных с информацией о правах доступа. При изменении параметра драйвер необходимо перезагрузить
DB2 (DB2)	
DB2 DSN	Задает имя соединения с БД. Это название псевдонима клиента DB2, который создается на сервере
DRIVER	Имя драйвера DB2
ROWSET SIZE	Определяет число записей, передаваемых одновременно
Драйверы ODBC	
ODBC DRIVER	Имя драйвера ODBC
ODBC DSN	Имя набора данных ODBC

После настройки параметров драйвера и сохранения текущей конфигурации новый псевдоним становится доступен для любого приложения, использующего BDE.

Страница **Configuration**, помимо настройки установленных в BDE драйверов, позволяет редактировать параметры, используемые BDE при инициализации приложения. Эти параметры доступны при выборе узлов **System**, а затем **INIT** иерархического дерева. Назначение параметров представлено в табл. 16.4.

Таблица 16.4. *Параметры инициализации приложения*

Параметр	Назначение
AUTO ODBC	В значении `True` при каждой инициализации в BDE автоматически импортируются все установленные в системе драйверы ODBC
DATA REPOSITORY	Имя текущего словаря данных
DEFAULT DRIVER	Локальный драйвер, используемый по умолчанию в драйвере STANDARD

Таблица 16.4 *(окончание)*

Параметр	Назначение
LANGDRIVER	Драйвер языковой поддержки. При использовании стандартных локальных драйверов это значение перекрывают те, которые определены непосредственно в конфигурациях драйверов Paradox, dBASE, FoxPro, ASCII
LOCAL SHARE	Устанавливает режим совместного использования файлов приложениями, работающих через BDE и другие программы. В значении `True` совместное использование разрешено
LOW MEMORY USAGE LIMIT	Максимальный объем памяти (в Кбайтах), который BDE пытается использовать в первом Мбайте оперативной памяти
MAXBUFSIZE	Максимальный размер кэша данных. Он должен быть не меньше значения параметра MINBUFSIZE и кратен 128
MAXFILEHANDLES	Максимальное число используемых файлов
MEMSIZE	Максимальный объем используемой BDE памяти в Мбайтах
MINBUFSIZE	Минимальный размер кэша данных. Он должен быть не больше значения параметра MAXBUFSIZE и кратен 128
MTS POOLING	Управляет режимом объединения ресурсов MTS (Microsoft Transaction Server). Обеспечивает лучшую производительность
SHAREDMEMLOCATION	Содержит адрес памяти, который пытаются использовать Менеджер памяти и Менеджер буфера. При возникновении конфликтов адрес необходимо поменять вручную. Задается только второе слово адреса
SHAREDMEMSIZE	Максимальный объем памяти, используемый Менеджером памяти и Менеджером буфера
SQLQRYMODE	См. табл. 16.2
SYSFLAGS	Не используется
VERSION	Номер внутренней версии BDE. Только для чтения

Также на странице **Configuration** устанавливаются параметры форматов даты, времени и чисел. Доступ к параметрам осуществляется через узлы **System** и **Format**.

Интерфейс прикладного программирования BDE

Как уже говорилось выше, любое приложение Delphi, работающее с базами данных и написанное с использованием стандартных компонентов доступа к данным, обращается к данным и получает результат при помощи BDE. При этом механизм доступа к данным использует вызовы функций из API BDE.

Достаточно сложно представить себе такую ситуацию, когда возникает необходимость создания приложения, использующего только функции BDE, без применения компонентов доступа к данным VCL. А вот отдельные функции вполне могут понадобиться в любой программе. Поэтому рассмотрим процесс работы приложения, использующего вызовы BDE, т. к. это дает хорошую возможность понять механизм доступа к данным, который реализован в Delphi.

Итак, для создания приложения на основе вызовов функций BDE необходимо выполнить следующие операции:

1. Инициализация BDE (функция `DbiInit`).
2. Открытие объекта базы данных (функция `DbiOpenDatabase`).
3. Определение рабочего каталога (функция `DbiSetDirectory`), если на предыдущем этапе не задается псевдоним БД.
4. Определение временного каталога (функция `DbiSetPrivateDir`).
5. Открытие набора данных и создание курсора (функции `DbiOpenTable`, `DbiQExec` и пр.; дескриптор курсора `hDBICur`).
6. Заполнение структуры `CURProps`, содержащей данные о курсоре и наборе данных (функция `DbiGetCursorProps`).
7. Выделение памяти для буфера записи.
8. Навигация по набору данных (функции `DbiSetToBegin`, `DbiSetToEnd`, `DbiSetToCursor` и пр.)
9. Чтение необходимой записи (функции `DbiGetRelativeRecord`, `DbiGetNextRecord`, `DbiGetRecord`, `DbiGetPriorRecord` и др.).
10. Чтение или обновление необходимого поля (функции `DbiGetField`, `DbiPutField`).
11. Освобождение всех ресурсов (освобождение буфера записи, закрытие курсора, таблицы, BDE).

При использовании в программе функций из API BDE необходимо включить в секцию `uses` модуль BDE.

В прикладном программировании задачи, которые требовали бы выполнения всех описанных выше операций, практически не встречаются. Между

тем, включение в программный код отдельных функций API BDE оправдано. В качестве примера рассмотрим приложение, которое позволяет очистить таблицу базы данных (рис. 16.4).

Дело в том, что при удалении записи из таблицы локальной СУБД (например Paradox) размер файла таблицы остается прежним, даже если удалить все записи. То есть на самом деле запись не удаляется, а только становится недоступной. При интенсивном использовании базы данных файлы таблиц могут занимать значительные объемы дискового пространства при довольно умеренном числе записей.

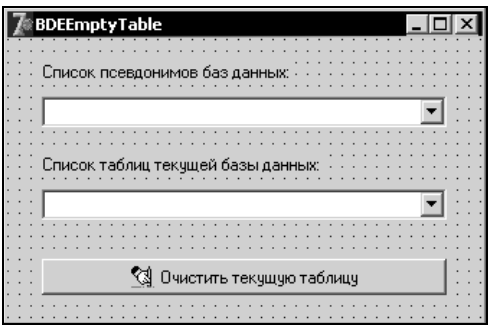


Рис. 16.4. Главная форма проекта BDEEmptyTable

Полная очистка таблиц базы данных осуществляется функцией `DbiEmptyTable` из API BDE. Именно она используется в демонстрационном приложении для радикального уменьшения размера таблиц.

Примечание

Функция `DbiEmptyTable` используется в методе `EmptyTable` компонентов доступа к данным *(см. гл. 17)*.

В листинге 16.1 приведен исходный код этого приложения. Помимо указанной функции, в нем используются функции создания списка параметров доступных баз данных и таблиц текущей базы данных.

Листинг 16.1. Модуль главной формы приложения BDEEmptyTable

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, BDE, ExtCtrls, DBCtrls, Grids, DBGrids, Db, DBTables, Buttons;
```

```
type
  TMainForm = class(TForm)
    AliasesList: TComboBox;
    TablesList: TComboBox;
    EmptyBtn: TBitBtn;
    Label1: TLabel;
    Label2: TLabel;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormShow(Sender: TObject);
    procedure AliasesListChange(Sender: TObject);
    procedure EmptyBtnClick(Sender: TObject);
  private
    hDB: hDBIDB;
    hCursor: hDBICur;
    DBDesc: DBDesc;
    TblDesc : TBLBaseDesc;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormShow(Sender: TObject);
var Rslt: DBIResult;
begin
 AliasesList.Items.Clear;
 TablesList.Items.Clear;
 hDB := Nil;
 try
  DbiInit(Nil);   // Инициализация BDE
  DbiOpenDatabaseList(hCursor);
  repeat
   Rslt:= DbiGetNextRecord(hCursor, dbiNOLOCK, @DBDesc, nil);
   if (Rslt <> DBIERR_EOF) then
AliasesList.Items.Add(StrPas(DBDesc.szName));
  until (rslt <> DBIERR_NONE);
  DbiCloseCursor(hCursor);
 except
  on E:EDBEngineError do ShowMessage('Ошибка инициализации BDE');
 end;
end;
```

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 try
 finally
  if hDB <> Nil then DbiCloseDatabase(hDB);// Закрытие базы данных
  DbiExit;                                 // Закрытие сеанса работы с BDE
 end
end;

procedure TMainForm.AliasesListChange(Sender: TObject);
begin
 try
  if hDB <> Nil
  then DbiCloseDatabase(hDB);// Закрытие базы данных
  DbiOpenDatabase            // Открытие базы данных
  (
   PChar(AliasesList.Text),  // Псевдоним базы данных
   Nil,                      // Тип базы данных
   dbiReadWrite,             // Режим редактирования данных
   dbiOpenShared,            // Режим разделения данных
   Nil,                      // Пароль
   0,                        // Число дополнительных параметров
   Nil,                      // Перечень полей для доп. параметров
   Nil,                      // Список доп. параметров
   hDB                       // Дескриптор базы данных
);
  DbiSetPrivateDir('c:\temp');// Определение временного каталога
  DbiOpenTableList(hDb, False, False, '*.DB', hCursor);
  TablesList.Items.Clear;
  TablesList.Clear;
  while DbiGetNextRecord(hCursor, dbiNOLOCK, @TblDesc, nil) = dbiErr_None
   do TablesList.Items.Add(TblDesc.szName);
  DbiCloseCursor(hCursor);
 except
  on E:EDBEngineError do ShowMessage('Ошибка открытия базы данных');
 end;
end;

procedure TMainForm.EmptyBtnClick(Sender: TObject);
begin
 try
 DbiEmptyTable(hDB, Nil, PChar(TablesList.Text), '');
 except
```

```
  on E:EDBEngineError do ShowMessage('Неверно задана таблица');
 end;
end;

end.
```

При открытии главной формы (метод-обработчик `FormShow`) функция `DbiInit` осуществляет инициализацию BDE. Затем функция `DbiOpenDatabaseList` создает в памяти временную таблицу, в которую записываются характеристики каждой зарегистрированной базы данных. Для этого применяется структура `DBDesc`. Курсор `hCursor` обеспечивает доступ к записям о базах данных.

После этого функция `DbiGetNextRecord` позволяет осуществить последовательное считывание имен псевдонимов баз данных (для этого в параметре передается указатель на структуру `DBDesc`) и их запись в список компонента `AliasesList` типа `TComboBox`.

При выборе из этого списка конкретного псевдонима работает метод-обработчик `AliasesListChange`. В нем открывается соответствующая база данных (функция `DbiOpenDatabase`), доступ к которой в дальнейшем осуществляется через дескриптор `hDB`.

Функция `DbiOpenTableList` создает временную таблицу в памяти, в которую помещаются данные о таблицах выбранной базы данных в соответствии с форматом структуры `TBLBaseDesc`. Функция `DbiGetNextRecord` позволяет передать эту информацию в список компонента `TablesList` типа `TComboBox`.

При щелчке на кнопке `EmptyBtn` в методе-обработчике `EmptyBtnClick` работает функция `DbiEmptyTable`, которая очищает выбранную ранее в компоненте `TablesList` таблицу.

Теперь рассмотрим пример простейшего приложения, которое может отображать два поля из таблицы COUNTRY.DB в демонстрационной базе данных DBDEMOS. Эта база данных поставляется в комплекте Delphi. В примере использованы только функции API BDE.

Проект называется DirectBDE и имеет только одну форму, в которой отображаются сведения из таблицы COUNTRY.DB о государствах и их столицах (рис. 16.5). Кнопки в нижней части формы позволяют перемещаться по набору данных.

Листинг 16.2. Модуль главной формы приложения DirectBDE

```
unit Unit1;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, BDE, ExtCtrls;

type
  TMainForm = class(TForm)
    PriorBtn: TBitBtn;
    Panel2: TPanel;
    NextBtn: TBitBtn;
    Label3: TLabel;
    CountryEdit: TEdit;
    Label1: TLabel;
    CapitalEdit: TEdit;
    procedure PriorBtnClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure NextBtnClick(Sender: TObject);
  private
    hDB: hDBIDB;
    hCur: hDBICur;
    CursProps: CurProps;
    RecBuf: pByte;
    FValue: array[0..255] of Char;
    IsEmpty: Bool;
    procedure OnBDEError;
  public
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.OnBDEError;
var ErrInfo: dbiErrInfo; // Структура, содержащая информацию об ошибках
    AStr: String;
begin
 DbiGetErrorInfo(True, ErrInfo);// Функция возвращает информацию об ошибке
 case ErrInfo.iError of
   9733: AStr := 'Для создания записи недостаточно параметров';
  10024: AStr := 'Ошибка доступа к данным';
  10245: AStr := 'База данных занята другим пользователем';
  10038: AStr := 'Значение поля задано неверно';
```

```
  11871: AStr := 'Несоответствие типов';
  11959: AStr := 'В выражении отсутствует оператор GROUP BY';
 else
  AStr := 'Ошибочная операция с данными';
 end;
 ShowMessage(AStr);
end;

procedure TMainForm.FormShow(Sender: TObject);
begin
 hDB := Nil;
 hCur := Nil;
 DbiInit(Nil);          // Инициализация системы BDE
 DbiOpenDatabase        // Открытие базы данных
 (
  'DBDEMOS',            // Псевдоним базы данных
  Nil,                  // Тип базы данных
  dbiReadWrite,         // Режим редактирования данных
  dbiOpenShared,        // Режим разделения данных
  Nil,                  // Пароль
  0,                    // Число дополнительных параметров
  Nil,                  // Перечень полей для доп. параметров
  Nil,                  // Список доп. параметров
  hDB                   // Дескриптор базы данных
);
 DbiSetPrivateDir('c:\temp'); // Определение временного каталога
 DbiOpenTable           // Открытие таблицы
 (
  hDB,                  // Дескриптор базы данных
  PChar('COUNTRY'),     // Название таблицы
  PChar(szParadox),     // Тип таблицы (только для локальных БД)
  Nil,                  // Название индекса (необязательный)
  Nil,                  // IndexTagName — только для dBASE
  0,                    // 0 — использовать первичный индекс
  dbiReadWrite,         // Режим редактирования данных
  dbiOpenShared,        // Режим разделения данных
  xltField,             // Режим трансляции данных
  False,                // Признак одностороннего перемещения курсора
  Nil,                  // Дополнительные параметры
  hCur                  // Дескриптор курсора таблицы
);
 DbiGetCursorProps      // Определение параметров курсора
 (
  hCur,                 // Дескриптор курсора таблицы
```

```
  CursProps             // Структура параметров курсора
);
 GetMem                 // Выделение памяти под буфер записи
(
 RecBuf,
 CursProps.iRecbufSize*SizeOf(Byte)
);
 DbiSetToBegin(hCur);  // Установка курсора в начало набора данных
 DbiGetNextRecord      // Перемещение на первую запись
 (
  hCur,                 // Дескриптор курсора таблицы
  dbiNoLock,            // Режим ограничения доступа
  RecBuf,               // Буфер записи
  Nil                   // Параметры записи
);
 DbiGetField            // Получение значения поля
 (
  hCur,                 // Дескриптор курсора таблицы
  1,                    // Номер поля в структуре таблицы
  RecBuf,               // Буфер записи
  @FValue,              // Переменная, в которую передается значение
  IsEmpty               // Признак пустой ячейки
);
 MainForm.CountryEdit.Text := FValue;
 DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
 MainForm.CapitalEdit.Text := FValue;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 try
 finally
  FreeMem(RecBuf);         // Освобождение памяти буфера записи
  DbiCloseCursor(hCur);    // Закрытие курсора
  DbiCloseDatabase(hDB);   // Закрытие базы данных
  DbiExit;                 // Закрытие сеанса работы с BDE
 end
end;

procedure TMainForm.PriorBtnClick(Sender: TObject);
begin
 try
  if DbiGetPriorRecord(hCur, dbiNoLock, RecBuf, Nil) = DBIERR_BOF
  then PriorBtn.Enabled := False
```

```
  else
  begin
   if Not NextBtn.Enabled then NextBtn.Enabled := True;
   DbiGetField(hCur, 1, RecBuf, @FValue, IsEmpty);
   MainForm.CountryEdit.Text := FValue;
   DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
   MainForm.CapitalEdit.Text := FValue;
  end;
 except
  OnBDEError;
 end;
end;

procedure TMainForm.NextBtnClick(Sender: TObject);
begin
 try
  if DbiGetNextRecord(hCur, dbiNoLock, RecBuf, Nil)=DBIERR_EOF
  then NextBtn.Enabled := False
  else
  begin
   if Not PriorBtn.Enabled then PriorBtn.Enabled := True;
   DbiGetField(hCur, 1, RecBuf, @FValue, IsEmpty);
   MainForm.CountryEdit.Text := FValue;
   DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
   MainForm.CapitalEdit.Text := FValue;
  end;
 except
  OnBDEError;
 end;
end;

end.
```

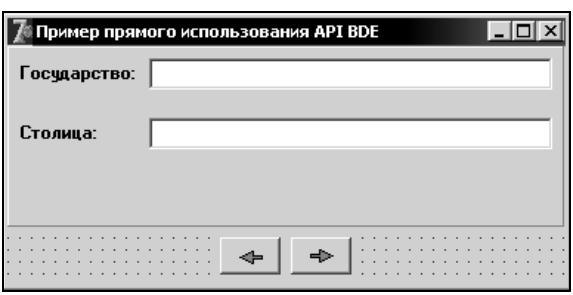


Рис. 16.5. Главная форма проекта DirectBDE

При показе главной формы приложения в процедуре `FormShow` проводится инициализация BDE, открытие базы данных и таблицы. При этом создаются дескрипторы базы данных `hDB` и курсора таблицы `hCur`, которые играют в дальнейшей работе приложения важную роль. Если при создании базы данных не указывать псевдоним БД, то обязательно нужно определить рабочий каталог базы данных с помощью функции `DbiSetDirectory`.

После этого отводится память под буфер записи, в который будут передаваться значения полей текущей строки таблицы. Размер буфера определяется при помощи структуры `CURProps`.

Затем курсор устанавливается на начало набора данных и на первую запись и осуществляется чтение значений двух полей таблицы.

Навигация по набору данных реализована в методах-обработчиках на нажатие кнопок формы. Их действие аналогично за исключением направления перемещения. При щелчке на кнопке выполняется переход на следующую или предыдущую запись, данные из новой записи помещаются в буфер записи `RecBuf`. Оттуда при помощи функции `DbiGetField` осуществляется чтение значений полей. При достижении начала или конца набора данных кнопка деактивируется.

При закрытии формы проводятся операции по освобождению памяти буфера записи, закрытию базы данных и BDE.

Соединение с источником данных

Все обращения из приложения к таблицам одной базы данных осуществляются через одно соединение, на которое замыкаются все компоненты доступа к данным, имеющие соответствующие значения свойства `DatabaseName` *(см. ниже)*.

Все управление одиночным соединением с какой-либо базой данных в BDE осуществляется компонентом `TDatabase` (табл. 16.5). В процессе работы компонент активно использует параметры псевдонимов и драйверов BDE.

Таблица 16.5. *Свойства и методы компонента* `TDatabase`

Объявление	Тип	Описание
Свойства		
`property AliasName: string;`	Pb	Задает имя псевдонима BDE используемой базы данных
`property Connected: Boolean;`	Pb	Управляет включением соединения с базой данных
`property DatabaseName: string;`	Pb	Определяет имя базы данных

Таблица 16.5 *(продолжение)*

Объявление	Тип	Описание
`property DataSetCount: Integer;`	Ro	Возвращает число открытых наборов данных, работающих через данное соединение
`property DataSets[Index: Integer]: TDBDataSet;`	Ro	Индексированный список всех объектов открытых наборов данных этого соединения
`property Directory: string;`	Pu	Определяет текущий каталог для баз данных Paradox и dBASE
`property DriverName: string;`	Pb	Содержит имя драйвера базы данных
`property Exclusive: Boolean;`	Pb	При значении `True` другие приложения не могут работать с базой данных одновременно с этим компонентом
`type HDBIDB: Longint;` `property Handle: HDBIDB;`	Pu	Дескриптор BDE. Используется для прямых вызовов функций API BDE
`property HandleShared: Boolean;`	Pu	При значении `True` дескриптор BDE компонента доступен в компоненте `TSession`
`property InTransaction: Boolean`	Ro	Показывает состояние транзакции. При значении `True` транзакция выполняется
`property IsSQLBased: Boolean;`	Ro	При значении `True` соединение работает через драйвер SQL Links
`property KeepConnection: Boolean;`	Pb	При значении `True` соединение продолжает оставаться активным после закрытия всех наборов данных. При значении `False` после закрытия последнего набора данных соединение закрывается
`type TLocale: Pointer;` `property Locale: TLocale;`	Ro	Указывает на языковый драйвер BDE, используемый при работе с базой данных
`property LoginPrompt: Boolean;`	Pb	Управляет отображением стандартного диалога регистрации пользователя при подключении к серверу
`property Params: TStrings;`	Pb	Содержит список значений параметров псевдонима BDE, которые пользователь задает перед подключением к серверу

Таблица 16.5 *(продолжение)*

Объявление	Тип	Описание
`property Session: TSession`	Ro	Указывает на компонент `TSession`, который управляет работой данного компонента
`property SessionAlias: Boolean;`	Ro	При значении `True` при подключении к БД используется псевдоним сессии
`property SessionName: string;`	Pb	Содержит имя сеанса, который управляет работой компонента
`property ReadOnly: Boolean;`	Pb	Управляет режимом доступа к данным "только для чтения"
`property Temporary: Boolean;`	Pu	Значение `True` говорит о том, что экземпляр компонента создан во время выполнения

`type`
`TTraceFlag = (tfQPrepare, tfQExecute, tfError, tfStmt, tfConnect, tfTransact, tfBlob, tfMisc, tfVendor, tfDataIn, tfDataOut);`
`TTraceFlags = set of TTraceFlag;`
`property TraceFlags: TTraceFlags;`
	Pu	Определяет перечень операций, выполнение которых отображается в утилите SQL Monitor при выполнении приложения

`type TTransIsolation = (tiDirtyRead, tiReadCommitted, tiRepeatableRead);` `property TransIsolation: TTransIsolation;`	Pb	Определяет уровень изоляции транзакций: • `tiDirtyRead` — незавершенное чтение; • `tiReadCommitted` — завершенное чтение; • `tiRepeatableRead` — повторяемое чтение
Методы		
`procedure ApplyUpdates(const DataSets: array of TDBDataSet);`	Pu	Фиксирует все изменения в наборах данных, работающих через это соединение, в базе данных
`procedure Close;`	Pu	Закрывает все открытые наборы данных и соединение
`procedure CloseDatasets;`	Pu	Закрывает все открытые наборы данных, работающие через это соединение

Таблица 16.5 *(окончание)*

Объявление	Тип	Описание
procedure Commit;	Pu	Завершает выполнение текущей транзакции и фиксирует все изменения в базе данных
function Execute(const SQL: string; Params: TParams = nil; Cache: Boolean = False; Cursor: phDBICur = nil): Integer;	Pu	Выполняет запрос SQL без использования компонента TQuery. Текст запроса содержится в параметре SQL. Параметры запроса определяются параметром Params. Режим кэширования изменений включается параметром Cache. Параметр Cursor может использоваться при работе с функциями BDE, использующими курсор набора данных *(см. гл. 14)*
procedure FlushSchemaCache(const TableName: string);	Pu	Изменяет представление о структуре таблиц БД, загруженной в память
procedure Open;	Pu	Открывает соединение
procedure Rollback;	Pu	Отменяет все операции текущей транзакции и завершает ее
procedure StartTransaction;	Pu	Начинает выполнение транзакции
procedure ValidateName(const Name: string);	Pu	Вызывает исключительную ситуацию, если база данных Name уже открыта в текущей сессии
Методы-обработчики событий		
type TLoginEvent = procedure(Database: TDatabase; LoginParams: TStrings) of object;	Pb	Вызывается при регистрации пользователя на сервере
property OnLogin: TLoginEvent;		
property AfterConnect: TNotifyEvent;	Pb	Вызывается после подключения
property AfterDisconnect: TNotifyEvent;	Pb	Вызывается после отключения
property BeforeConnect: TNotifyEvent;	Pb	Вызывается перед подключением
property AfterDisconnect: TNotifyEvent;	Pb	Вызывается перед отключением

Обычно компонент TDatabase размещается в модуле данных приложения.

Для определения базы данных (сервера), с которой приложение устанавливает соединение при помощи компонента TDatabase, чаще используется свойство AliasName. Свойства DatabaseName и DriverName предоставляют альтернативный способ создания соединения.

Если соединение задано свойством AliasName, то свойство DatabaseName можно использовать для создания временного псевдонима, который будет доступен только для компонентов доступа к данным внутри приложения. При щелчке на кнопке списка доступных псевдонимов свойства DatabaseName в Инспекторе объектов для любого компонента доступа к данным в списке будет доступен и временный псевдоним компонента TDatabase.

Например, при переключении приложения на другую базу данных можно изменить только значение псевдонима в компоненте TDatabase. Если все компоненты наборов данных подключены к временному псевдониму компонента TDatabase, то они автоматически переключатся на новую БД.

Дополнительные возможности управления наборами данных при переключении соединения предоставляют свойства Connected и KeepConnection. Они позволяют одновременно с соединением закрыть все активные наборы данных.

Если наборы данных приложения подключены к базе данных через компонент TDatabase, то перед их открытием необходимо установить соединение с БД. Соединение с БД устанавливается при помощи метода Open. Если попытаться активизировать набор данных без этого метода, то соединение будет установлено автоматически.

Аналогичная картина возникает при закрытии наборов данных и отключении от БД. Дополнительное средство управления в этом случае предоставляет свойство KeepConnection. Если оно равно значению True, то при закрытии последнего открытого набора данных соединение остается открытым. В противном случае соединение автоматически закрывается.

Это позволяет управлять соединением в различных исходных ситуациях. При большой загруженности сервера бывает необходимо прерывать соединение каждый раз. Если требуется разгрузить сетевой график, то соединение лучше оставлять включенным.

При подключении к базе данных довольно часто требуется задать значения для параметров драйвера BDE. Для этого используется свойство Params, представляющее собой обычный список. В нем необходимо задавать названия изменяемых параметров и их новые значения:

```
USERNAME=SYSDBA
PASSWORD=masterkey
```

Значения параметров можно задавать как статически, так и динамически во время выполнения.

Компонент TDatabase может облегчить подключение к базам данных с регистрацией пользователей. При регистрации на сервере достаточно задать имя пользователя, пароль в свойстве Params *(см. выше)* и установить для свойства LoginPrompt значение False. Эта комбинация работает как во время выполнения, так и во время разработки.

Примечание

Для организации доступа к защищенным паролем таблицам Paradox используется метод AddPassword компонента TSession *(см. выше)*.

Дополнительные возможности обработки регистрации пользователя дает единственный метод-обработчик OnLogin, программный код которого выполняется вместо появления стандартного диалога ввода имени и пароля. Это позволяет разработчику создавать собственные сценарии регистрации пользователей.

Для обеспечения доступа к функциям API BDE используется свойство Handle (BDE играет важную роль при создании соединения).

Управление выполнением транзакций осуществляется при помощи методов StartTransaction, Commit и RollBack.

Компоненты доступа к данным

Компоненты доступа к данным, используемые при разработке приложений BDE, располагаются на странице **BDE** Палитры компонентов. Их общими предками являются классы TBDEDataSet и TDBDataSet (см. рис. 12.1). Они обеспечивают работоспособность основных компонентов доступа к данным BDE — TTable, TQuery, TStoredProc.

Класс *TBDEDataSet*

Этот класс является потомком класса TDataSet, его значение трудно переоценить: именно TBDEDataSet обеспечивает работоспособность важнейших механизмов набора данных за счет обращения к функциям BDE (табл. 16.6). Например, класс TBDEDataSet перекрывает абстрактные методы своего предка TDataSet, отвечающие за такие важнейшие операции, как чтение данных и сохранение изменений в базе данных, навигация по записям набора данных, фильтрация.

Напомним, что все эти механизмы не созданы "с нуля", а только дополнены обращениями к функциям BDE в необходимых местах методов, изначально описанных в классе TDataSet. Например, для обеспечения фильтрации записей набора данных к классу добавлено новое свойство:

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

Оно определяет дополнительные параметры отбора записей по фильтру (чувствительность к регистру символов и отбор по текстовому шаблону).

Дополнительно к существующим добавлен механизм кэширования изменений. Теперь все вносимые пользователем изменения могут накапливаться в специальном буфере, а их передачей в базу данных можно управлять.

Примечание

Эта возможность очень полезна при создании клиентских приложений в архитектуре клиент/сервер и играет ключевую роль при обеспечении возможности редактирования наборов данных сложных запросов SQL.

Дополнительно к методам работы с полями класса `TDataSet` добавлены функции использования полей в формате BLOB.

Для обеспечения использования функций API BDE на программном уровне добавлено свойство, содержащее дескриптор курсора, соответствующего текущей записи набора данных:

```
type HDBICur: Longint;
property Handle: HDBICur;
```

Также класс обеспечивает возможность программного управления вторичными индексами набора данных в зависимости от типа таблицы базы данных.

Таблица 16.6. *Свойства и методы класса* `TBDEDataSet`

Объявление	Тип	Описание
Свойства		
`property BlockReadSize: Integer;`	Pu	Определяет размер буфера при блочном чтении данных. Такой режим используется для быстрого перемещения по большим массивам данных. Если значение свойства больше нуля, навигация по набору данных осуществляется без изменения состояния компонентов отображения данных и вызова методов-обработчиков событий
`property CacheBlobs: Boolean;`	Pu	Разрешает использование буфера памяти для данных типа BLOB

Таблица 16.6 *(продолжение)*

Объявление	Тип	Описание
`property CachedUpdates: Boolean;`	Pb	Включает или отключает режим кэширования изменений в наборе данных. Используется в клиентских приложениях архитектуры клиент/сервер
`property CanModify: Boolean;`	Pu, Ro	Если набор данных позволяет делать изменения, свойство возвращает `True`, иначе — `False`
`property ExpIndex: Boolean;`	Pu, Ro	Показывает, используются ли в наборе данных индексы dBASE
`property Filter: string;`	Pb, Ro	Содержит выражение для фильтра набора данных
`property Filtered: Boolean;`	Pb, Ro	Управляет включением фильтра набора данных
`TFilterOption = (foCaseInsensitive, foNoPartialCompare);` `property FilterOptions: TFilterOptions;`	Pb	Определяет параметры фильтра: • `foCaseInsensitive` — строковые значения фильтруются без учета регистра; • `foNoPartialCompare` — при фильтрации символ "*" рассматривается как обычный символ, иначе он означает, что на этом месте может находиться произвольное подмножество любых символов
`type HDBICur: Longint;` `property Handle: HDBICur;`	Pu, Ro	Указатель на курсор BDE, связанный с текущей записью набора данных
`property KeySize: Word;`	Pu, Ro	Содержит размер ключа для текущего индекса набора данных
`type TLocale: Pointer;` `property Locale: TLocale;`	Pu, Ro	Указатель на языковый драйвер BDE
`property RecNo: Longint;`	Pu	Номер текущей записи набора данных
`property RecordCount: Longint;`	Ro	Содержит число записей в наборе данных
`property RecordSize: Word;`	Ro	Содержит размер одной записи набора данных
`property UpdateObject: TDataSetUpdateObject;`	Pu	Экземпляр объекта `TUpdateObject`, используемого при кэшировании изменений

Таблица 16.6 *(продолжение)*

Объявление	Тип	Описание
`type TUpdateRecordTypes = set of (rtModified, rtInserted, rtDeleted, rtUnmodified);` `property UpdateRecordTypes: TUpdateRecordTypes;`	Pu	Определяет видимость записей в режиме кэширования изменений в зависимости от их состояния: • `rtModified` — доступны измененные записи; • `rtinserted` — доступны добавленные записи; • `rtDeleted` — доступны удаленные записи; • `rtUnmodified` — доступны немодифицированные записи
`property UpdatesPending: Boolean;`	Ro	Значение `True` говорит о том, что буфер изменений при кэшировании содержит не сохраненные на сервере изменения
Методы		
`procedure ApplyUpdates;`	Pu	Записывает изменения из буфера в базу данных в режиме кэширования
`function BookmarkValid(Bookmark: TBookmark): Boolean; override;`	Pu	Проверяет существование экземпляра закладки, передаваемого в параметре `Bookmark`
`procedure Cancel;`	Pu	Отменяет все изменения, сделанные в текущей записи с момента последнего сохранения
`procedure CancelUpdates;`	Pu	Отменяет все изменения, сделанные с момента последней записи в базу данных и очищает буфер в режиме кэширования
`procedure CommitUpdates;`	Pu	Очищает буфер изменений в режиме кэширования
`function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;`	Pu	Проверяет идентичность закладок, указанных в параметрах `Bookmark1` и `Bookmark2`. При значении 1 сигнализирует о наличии отличий в двух закладках
`function ConstraintCallBack(Req: DsInfoReq; var ADataSources: DataSources): DBIResult; stdcall;`	Pu	Обеспечивает доступ к функциям ограничения данных API BDE

Таблица 16.6 *(продолжение)*

Объявление	Тип	Описание
`function ConstraintsDisabled: Boolean;`	Pu	Показывает, включены или отключены ограничения данных
`function CreateBlobStream(Field: TField; Mode: TblobStreamMode): TStream; override;`	Pu	Создает поток для чтения/записи данных типа BLOB
`procedure DisableConstraints;`	Pu	Отключает ограничения данных
`procedure EnableConstraints;`	Pu	Включает ограничения данных
`procedure FetchAll;`	Pu	Переносит все изменения из буфера и восстанавливает все записи от текущей позиции до конца набора данных
`procedure FlushBuffers;`	Pu	Передает в базу данных все изменения из буфера записи
`function GetBlobFieldData(FieldNo: Integer; var Buffer: TblobByteData): Integer; override;`	Pu	Читает все данные BLOB из поля `FieldNo` в буфер `Buffer`
`function GetCurrentRecord(Buffer: PChar): Boolean;`	Pu	Помещает текущую строку в буфер `Buffer`
`procedure GetIndexInfo;`	Pu	Обновляет информацию о текущем индексе набора данных
`function IsSequenced: Boolean; override;`	Pu	Определяет, поддерживает ли таблица БД нумерацию последовательности записей. В классе `TDataSet` всегда возвращает `True`, т. к. абстрактный набор данных свободен от конкретной реализации БД и всегда нумерует записи
`function Locate(const KeyFields: string; const KeyValues: Variant; Options: TlocateOptions): Boolean;`	Pu	Осуществляет поиск в наборе данных. Параметр `KeyFields` содержит список полей, по которым ведется поиск. Параметр `KeyValues` содержит значения полей для поиска. Параметр `Options` определяет условия поиска. Если запись найдена, курсор набора данных устанавливается на эту запись и возвращается `True` *(см. гл. 14)*

Таблица 16.6 *(окончание)*

Объявление	Тип	Описание
`function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant;`	Pu	Осуществляет поиск в наборе данных. Возвращает массив значений требуемых полей найденной записи. Параметры аналогичны методу `Locate` *(см. гл. 14)*
`procedure Post; override;`	Pu	Пересылает сделанные в текущей записи изменения в базу данных
`procedure RevertRecord;`	Pu	Отменяет все изменения в текущей строке при работающем буфере изменений
`procedure Translate(Src, Dest: PChar; ToOem: Boolean); override;`	Pu	Форматирует текст. Если параметр `ToOem = True`, текст `Src` в формате ANSI переводится в текст `Dest` в формате OEM и наоборот
`function UpdateStatus: TUpdateStatus;`		Возвращает тип сохраняемых в буфере изменений данных (см. табл. 16.1)
Методы-обработчики событий		

`TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);`
`TUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;`
`property OnUpdateError: TUpdateErrorEvent;`
	Pu	Вызывается при возникновении ошибки переноса кэшированных в буфере изменений в таблицу базы данных

`type TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);`
`TUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;`
`property OnUpdateRecord: TUpdateRecordEvent;`
	Pu	Вызывается при сохранении кэшированных в буфере изменений для отдельной записи.
		Применяется для организации дополнительного управления этим процессом, например для контроля какого-либо конкретного значения

Класс *TDBDataSet*

Класс `TDBDataSet` является непосредственным предком основных компонентов доступа к данным `TTable`, `TQuery` и `TStoredProc`. Новые свойства и методы класса обеспечивают соединение набора данных с базой данных и используют функции BDE (табл. 16.7).

В процессе соединения важнейшую роль играет свойство `DatabaseName`, которое должно содержать псевдоним или полный путь к файлам БД. Для управления отдельным соединением с базой данных можно применять специальный компонент `TDatabase`. Указатель на экземпляр такого компонента содержится в свойстве `Database`.

Многие функции API BDE используют в своей работе дескриптор специальной структуры, описывающей подключенную базу данных. Доступ к этому дескриптору можно получить через свойство `DBHandle`.

Приложение баз данных одновременно может использовать несколько наборов данных, каждый из которых подключен к собственной базе данных. Совокупность соединений управляется в рамках сеанса работы, который инкапсулируется компонентом `TSession`. Указатель на экземпляр такого компонента можно использовать в наборе данных при помощи свойства `DBSession`.

Для работы с удаленными серверами в класс введено свойство `Provider`, обеспечивающее доступ к интерфейсу `IProvider`.

Таблица 16.7. *Свойства и методы класса* `TDBDataSet`

Объявление	Тип	Описание
Свойства		
`property AutoRefresh: Boolean;`	`Pb`	При значении `True` все автоматически создаваемые значения полей (автоинкрементные, значения по умолчанию) обновляются автоматически
`property Database: TDatabase;`	`Pu, Ro`	Указатель связанного с набором данных компонента `TDatabase`
`property DatabaseName: string;`	`Pu, Pb`	Псевдоним базы данных
`type HDBISES: Longint;` `property DBHandle: HDBISES;`	`Pu, Ro`	Дескриптор базы данных. Используется при работе с API BDE
`type TLocale: Pointer;` `property DBLocale: TLocale;`	`Pu, Ro`	Идентифицирует языковый драйвер API BDE
`property DBSession: TSession`	`Pu, Ro`	Указатель для компонента `TSession`, с которым работает набор данных

Таблица 16.7 *(окончание)*

Объявление	Тип	Описание
`property Provider: IProvider;`	Pu, Ro	Идентифицирует интерфейс `IProvider`
`property SessionName: string;`	Pu, Ro	Содержит имя компонента сеанса, в котором работает набор данных
Методы		
`function CheckOpen(Status: DBIResult): Boolean;`	Pu	Возвращает результат вызова BDE. Используется для тестирования соединения
`procedure CloseDatabase(Database: TDatabase);`	Pu	Закрывает связь с базой данных, определяемой параметром `Database`
`procedure GetProviderAttributes(List: TList); override;`	Pu	Возвращает в списке `List` параметры языкового драйвера
`function OpenDatabase: TDatabase;`	Pu	Открывает связь с базой данных, определяемой свойством `DatabaseName`

Компонент *TTable*

Компонент `TTable` инкапсулирует таблицу реляционной базы данных, причем независимо от типа базы данных. Для доступа к данным компонент использует функции BDE *(см. выше)*.

Необходимая для работы база данных задается свойством `DatabaseName`, в котором можно указать зарегистрированный в BDE псевдоним БД или полный путь к файлам БД.

Таблица БД, на основе которой создается набор данных, определяется свойством `TableName`. При необходимости тип таблицы задается свойством `TableType`, хотя обычно это свойство имеет значение `ttDefault` (см. табл. 16.4), которое включает автоматическое определение типа таблицы по расширению файла.

Примечание

Свойство `TableType` работает только в локальных БД. Обратите внимание, что возможные значения свойства соответствуют основным типам локальных драйверов BDE.

При помощи методов `Open` и `Close` набор данных открывается и закрывается. О его состоянии можно судить по значению свойства `Active`. Более подробно о состоянии набора данных расскажет свойство `State` *(см. ниже)*.

Записи в набор данных можно отбирать при помощи свойств `Filter`, `Filtered`, `FilterOptions`, создающих фильтр, ограничивающий набор данных по значениям данных в одном или нескольких полях.

Методы `SetRangeStart`, `SetRangeEnd`, `SetRange`, `ApplyRange`, `EditRangeStart`, `EditRangeEnd` создают специальный диапазон включаемых в набор данных записей, отбор в диапазон проводится по задаваемым граничным значениям любых полей набора данных.

Поиск нужной записи можно осуществлять методами `Lookup` или `Locate` (достаточно просто, но не очень быстро) или, используя существующие в таблице базы данных индексы, методом `FindKey` (сложнее, но очень быстро).

От предков компонент унаследовал инструменты для работы с закладками. Это свойство `Bookmark` и методы `GetBookmark`, `FreeBookmark`, `GotoBookmark`.

Работа с полями осуществляется целой группой свойств и методов, среди которых особое место занимает свойство `Fields`, представляющее собой индексированный список всех полей набора данных. Это свойство удобно использовать в процессе разработки для организации доступа к полям.

Использование индексов обеспечено свойствами `IndexName`, `IndexFields`, `IndexFieldNames`, `IndexFiles`.

Свойства `MasterSource`, `MasterField`, `IndexName` дают возможность установить отношение типа главный/подчиненный с другой таблицей.

Очень полезны в практическом использовании методы и свойства для работы с буфером изменений (свойства `CachedUpdates`, `PendingUpdates`, `UpdateRecordTypes`, методы `ApplyUpdates`, `CancelUpdates`, `CommitUpdate`, `RevertRecord`). Буфер применяется в клиентских приложениях многоуровневых систем доступа к данным.

От классов `TDataSet` и `TBDEDataSet` унаследован обширный набор методов-обработчиков событий, позволяющий решать любые задачи по управлению набором данных.

В табл. 16.8 приведена справочная информация о свойствах и методах компонента `TTable`. После этого рассматриваются подробности применения основных механизмов набора данных.

Таблица 16.8. *Свойства и методы класса* `TTable`

Объявление	Тип	Описание
Свойства		
`property DataSource: TDataSource;`	`Pu, Ro`	Ссылается на компонент `TDataSource` главного набора данных в отношении главный/подчиненный

Таблица 16.8 *(продолжение)*

Объявление	Тип	Описание
`property DefaultIndex: Boolean;`	`Pb`	Управляет сортировкой данных. При значении `True` записи упорядочиваются по первичному ключу. При значении `False` упорядочивание не производится
`property Exclusive: Boolean;`	`Pb`	Ограничивает доступ к таблице. При значении `True` с таблицей может работать только одно приложение. Это свойство важно при одновременной работе нескольких приложений с данными в локальной сети
`property Exists: Boolean;`	`Pu, Ro`	Значение `True` говорит о том, что связанная с компонентом таблица базы данных существует
`property IndexDefs: TindexDefs;`	`Pb`	Содержит информацию об индексах таблицы
`property IndexFieldCount: Integer;`	`Pu, Ro`	Возвращает число полей в текущем индексе таблицы
`property IndexFieldNames: string;`	`Pb`	Разделенный запятыми список названий полей, составляющих текущий индекс. Используется для таблиц серверов SQL
`property IndexFields: [Index: Integer]: TField;`	`Pu`	Индексированный список полей текущего индекса
`property IndexFiles: TStrings;`	`Pb`	Список индексных файлов для таблиц dBASE
`property IndexName: string;`	`Pb`	Определяет вторичный индекс для таблицы. Используется для таблиц локальных СУБД
`property KeyExclusive: Boolean;`	`Pu`	Управляет границами диапазона, задаваемого методом `SetRange`. При значении `True` крайние записи в диапазон не включаются
`property KeyFieldCount: Integer;`	`Pu`	Содержит число полей ключа, используемых при поиске. При значении 0 применяется только первое поле, при значении 1 — два первых поля и т. д. По умолчанию устанавливается полное число полей ключа
`property MasterFields: string;`	`Pb`	Список имен полей главной таблицы, разделенных запятой, используемых при создании отношения главный/подчиненный

Таблица 16.8 *(продолжение)*

Объявление	Тип	Описание
`property MasterSource: TDataSource;`	Pb	Содержит имя компонента `TDataSource`, связанного с набором данных, который является главным в отношении главный/подчиненный
`property ReadOnly: Boolean;`	Pb	Включает и отключает режим "только для чтения". В некоторых случаях набор данных можно открыть только в этом режиме
`property StoreDefs: Boolean;`	Pb	При значении `True` все сведения об индексах и структуре таблицы хранятся вместе с формой или модулем данных. В этом случае при создании набора данных одновременно создаются поля, индексы, ограничения
`property TableLevel: Integer;`	Pu	Содержит значение уровня таблицы, используемого в драйвере BDE
`property TableName: TFileName;`	Pb	Определяет имя таблицы
`type TTableType = (ttDefault, ttParadox, ttDBase, ttASCII, ttFoxPro);`	Pb	Определяет тип таблицы для стандартного драйвера BDE. Значение `ttDefault` означает, что тип таблицы определяется по расширению файла
`property TableType: TTableType;`		
Методы		
`procedure AddIndex(const Name, Fields: string; Options: TIndexOptions);`	Pu	Создает новый индекс. Параметр `Name` определяет имя нового индекса, параметр `Fields` — список полей индекса через запятую, параметр `Options` задает тип индекса
`procedure ApplyRange;`	Pu	Включает в работу границы диапазона, заданные методами `SetRangeStart`, `SetRangeEnd` или `EditRangeStart`, `EditRangeEnd`
`type TBatchMode = (batAppend, batUpdate, batAppendUpdate, batDelete, batCopy);` `function BatchMove(ASource: TBDEDataSet; AMode: TBatchMode): Longint;`	Pu	Переносит записи из таблицы `ASource` в набор данных. Тип операции задается параметром `AMode`. Возвращает число обработанных записей

Таблица 16.8 *(продолжение)*

Объявление	Тип	Описание
`procedure CancelRange;`	Pu	Удаляет текущий диапазон
`procedure CloseIndexFile(const IndexFileName: string);`	Pu	Закрывает индексный файл для таблиц dBASE
`procedure CreateTable;`	Pu	Создает новую таблицу, основываясь на данных о структуре таблицы, содержащихся в свойствах `FieldDefs` и `IndexDefs`. Если свойство `FieldDefs` пустое, используется свойство `Fields`. Структура и данные существующей таблицы перезаписываются
`procedure DeleteIndex(const Name: string);`	Pu	Удаляет вторичный индекс
`procedure DeleteTable;`	Pu	Уничтожает таблицу базы данных. Набор данных при этом должен быть закрыт
`procedure EditKey;`	Pu	Переводит набор данных в режим редактирования буфера поиска. После использования этого метода можно изменять значения полей, которые применяются для поиска записей
`procedure EditRangeEnd;`	Pu	Разрешает редактирование нижней границы диапазона
`procedure EditRangeStart;`	Pu	Разрешает редактирование верхней границы диапазона
`procedure EmptyTable;`	Pu	Удаляет все записи из набора данных
`function FindKey(const KeyValues: array of const): Boolean;`	Pu	Проводит поиск записи, значения полей которой удовлетворяют условиям, заданным параметром `KeyValues`. Значения разделяются запятыми. Для поиска можно использовать только поля, входящие в текущий индекс. Для локальных стандартных таблиц BDE это поля, определяемые свойством `IndexName`. Для таблиц серверов SQL индекс можно задать свойством `IndexFieldNames`. При успешном поиске функция возвращает значение `True`
`procedure FindNearest(const KeyValues: array of const);`	Pu	Проводит поиск записи, значения полей которой, заданные параметром `KeyValues`, в минимальной степени отличаются от требуемых в большую сторону. Значения для поиска разделяются запятыми.

Таблица 16.8 *(продолжение)*

Объявление	Тип	Описание
(прод.)		Для поиска можно использовать только поля, входящие в текущий индекс. Для локальных стандартных таблиц BDE это поля, определяемые свойством `IndexName`. Для таблиц серверов SQL индекс можно задать свойством `IndexFieldNames`. При успешном поиске функция возвращает `True`
`procedure GetIndexNames(List: TStrings);`	Pu	Возвращает список индексов таблицы
`procedure GotoCurrent(Table: TTable);`	Pu	Синхронизирует курсор набора данных с курсором таблицы, заданной параметром `Table`
`function GotoKey: Boolean;`	Pu	Устанавливает курсор на запись, соответствующую значениям полей, заданным при последнем применении метода `SetKey` или `EditKey`
`procedure GotoNearest;`	Pu	Устанавливает курсор на запись, точно соответствующую значениям полей, заданным при последнем применении метода `SetKey` или `EditKey`, или следующую ближайшую к ним по значениям
`type TLockType = (ltReadLock, ltWriteLock);` `procedure LockTable(LockType: TLockType);`	Pu	Закрывает доступ к таблице Paradox или dBASE из других приложений
`procedure OpenIndexFile(const IndexFileName: string);`	Pu	Открывает индексный файл таблицы dBASE
`procedure RenameTable(const NewTableName: string);`	Pu	Переименовывает таблицу Paradox или dBASE
`procedure SetKey;`	Pu	Очищает буфер поиска. После использования этого метода можно изменять значения полей, используемые для поиска записей
`procedure SetRange(const StartValues, EndValues: array of const);`	Pu	Задает диапазон отбора записей. Параметр `StartValues` определяет значения полей для верхней границы диапазона.

Таблица 16.8 *(окончание)*

Объявление	Тип	Описание
(прод.)		Параметр `EndValues` определяет значения полей для нижней границы диапазона. Значения диапазона задаются для полей текущего индекса
`procedure SetRangeEnd;`	`Pu`	Задает нижнюю границу диапазона. После этого метода необходимо задать значения для полей текущего индекса, которые и будут нижней границей
`procedure SetRangeStart;`	`Pu`	Задает верхнюю границу диапазона. После этого метода необходимо задать значения для полей текущего индекса, которые и будут верхней границей
`type TLockType = (ltReadLock, ltWriteLock);`	`Pu`	Разблокирует таблицу Paradox или dBASE для доступа из других приложений
`procedure UnlockTable(LockType: TLockType);`		

Компонент *TQuery*

Компонент `TQuery` реализует все основные функции стандартного компонента запроса, описанные в *гл. 12*. Прямым предком компонента является класс `TDBDataSet`.

Для подключения к базе данных используется свойство `DatabaseName`, в котором задается псевдоним BDE или путь к базе данных.

Текст запроса определяется свойством `SQL`, для задания которого применяется простой редактор, открывающийся при щелчке на кнопке свойства в Инспекторе объектов (рис. 16.6).

Для управления текстом запроса во время выполнения приложения можно использовать возможности класса `TStrings`.

Основные свойства и методы компонента `TQuery` представлены в табл. 16.9.

Таблица 16.9. *Свойства и методы компонента* `TQuery`

Объявление	Тип	Описание
Свойства		
`property Constrained: Boolean;`	`Pb`	При значении `True` запрещает внесение в набор данных таких значений, которые не соответствуют условиям отбора запроса. Применимо для локальных БД

Таблица 16.9 *(продолжение)*

Объявление	Тип	Описание
`property DataSource: TDataSource;`	Pb	Ссылается на компонент `TDataSource`, из набора данных которого задаются значения параметров
`property Local: Boolean;`	Ro	Значение `True` означает, что запрос обращается к локальной таблице
`property ParamCheck: Boolean;`	Pb	При значении `True` параметры запроса обновляются при изменении свойства `SQL` во время выполнения
`property ParamCount: Word;`	Ro	Возвращает число параметров в запросе
`property Params[Index: Word]TParams;`	Pb	Индексированный список объектов `TParams`, каждый из которых соответствует одному параметру запроса
`property Prepared: Boolean;`	Pu	Возвращает результат выполнения операции подготовки запроса к выполнению
`property RequestLive: Boolean;`	Pb	При значении `False` результат запроса нельзя редактировать, независимо от того, редактируемый результат или нет. При значении `True` результат запроса можно редактировать, но только если он "живой"
`property RowsAffected: Integer;`	Ro	Возвращает число модифицированных записей набора данных с момента последнего выполнения запроса
`property SQL: TStrings;`	Pb	Содержит текст запроса
`property SQLBinary: PChar;`	Pu	Внутреннее свойство для обеспечения работы с BDE
`property StmtHandle: HDBIStmt;`	Ro	Возвращает экземпляр объекта, соответствующего запросу в BDE. Используется при прямом вызове функций BDE
`property Text: PChar;`	Ro	Указатель на символьный массив, содержащий передаваемый в BDE текст запроса
`property UniDirectional: Boolean;`	Pb	Определяет тип используемого курсора данных
Методы		
`procedure ExecSQL;`	Pu	Выполняет запрос без открытия набора данных

Таблица 16.9 *(окончание)*

Объявление	Тип	Описание
`procedure GetDetailLinkFields (MasterFields, DetailFields: TList); override;`	Pu	Заполняет списки параметров метода экземплярами объектов полей двух таблиц запроса, находящихся в отношении "один-ко-многим"
`function ParamByName(const Value: string): TParam;`	Pu	Возвращает ссылку на экземпляр объекта параметра с именем, переданным в параметре `Value`
`procedure Prepare;`	Pu	Готовит запрос к выполнению
`procedure UnPrepare;`	Pu	Освобождает ресурсы, занятые при подготовке запроса к выполнению

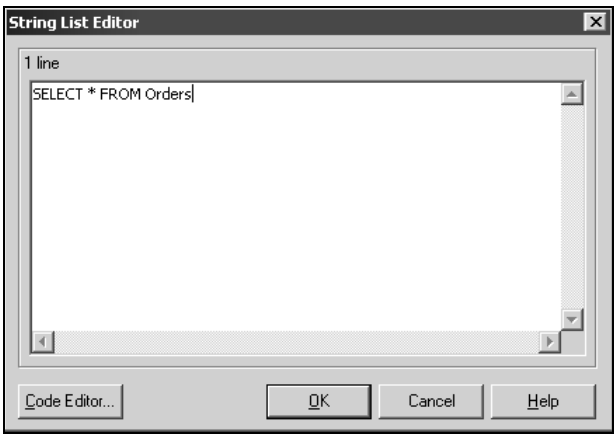


Рис 16.6. Редактор свойства `SQL` компонента `TQuery`

Компонент *TStoredProc*

Компонент `TStoredProc` обеспечивает использование в приложениях BDE хранимых процедур. Прямым предком компонента является класс `TDBDataSet`. Поэтому результатом выполнения хранимой процедуры может быть не только одиночный результат, но и полноценный набор данных.

Основные функции компонента `TStoredProc` соответствуют возможностям стандартного компонента хранимой процедуры, описанного в *гл. 12*. Свойства и методы компонента `TStoredProc` представлены в табл. 16.10.

Средствами классов-предков выполняется и подключение компонента к базе данных. Свойство `DatabaseName` определяет базу данных.

Свойство `StoredProcName` задает имя хранимой процедуры.

Перед выполнением хранимую процедуру необходимо подготовить. В частности, на этом этапе осуществляется передача параметров и выделение ресурсов. Эта операция выполняется автоматически при использовании методов `ExecProc` и `Open`, или задается явно методом `Prepare`.

Явная подготовка процедуры полезна при неоднократном вызове хранимой процедуры. Если перед первым вызовом процедуры выполнить метод `Prepare`, то все последующие вызовы будут осуществляться без подготовки, которая уже была сделана. В противном случае подготовка будет производиться автоматически перед каждым выполнением хранимой процедуры.

Таблица 16.10. *Свойства и методы компонента* `TStoredProc`

Объявление	Тип	Описание
Свойства		
`property Overload: Word;`	Pb	Идентификатор процедуры. Используется только для сервера Oracle
`type TParamBindMode = (pbByName, pbByNumber);` `property ParamBindMode: TParamBindMode;`	Pb	Определяет порядок присваивания значений параметров: • `pbByName` — по именам параметров; • `pbByNumber` — по номерам параметров в списке свойства `Params`
`property ParamCount: Word;`	Ro	Возвращает общее число параметров
`property Params: TParams;`	Pb	Индексированный список параметров
`property Prepared: Boolean;`	Pu	Возвращает значение `True`, если подготовка процедуры уже проводилась
`property StmtHandle: HDBIStmt;`	Ro	Дескриптор выражения BDE. Используется при прямом вызове функций BDE
`property StoredProcName: string;`	Pb	Содержит имя хранимой процедуры
Методы		
`procedure CopyParams(Value: TParams);`	Pu	Копирует параметры из списка `Value`
`function DescriptionsAvailable: Boolean;`	Pu	При значении `True` параметры хранимой процедуры доступны из приложения
`procedure ExecProc;`	Pu	Передает на сервер сигнал для запуска хранимой процедуры

Таблица 16.10 *(окончание)*

Объявление	Тип	Описание
`procedure GetResults;`	Pu	Возвращает выходные параметры в приложение (используется только для сервера Sybase)
`function ParamByName(const Value: string): TParam;`	Pu	Возвращает параметр с именем `Value`
`procedure Prepare;`	Pu	Готовит процедуру к выполнению
`procedure UnPrepare;`	Pu	Освобождает ресурсы, использованные во время подготовки процедуры

Резюме

Процессор баз данных Borland Database Engine 5 реализует стандартные функции доступа к данным и является ПО промежуточного слоя между приложением и базой данных. При помощи системы драйверов и псевдонимов BDE обеспечивает универсальный способ доступа к данным.

Специальный набор компонентов доступа к данным Delphi использует BDE. Но при необходимости разработчик может применять функции API BDE напрямую.

ГЛАВА 17

Технология dbExpress

Одной из проблем различных технологий доступа к данным, используемым в приложениях Delphi, является трудность распространения готовых приложений. Для BDE требуется отдельная установка, которая занимает порядка 15 Мбайт дискового пространства, а также специальная настройка псевдонимов *(см. гл. 16)*. ADO предустановлена в операционной системе, но нуждается в настраиваемых провайдерах данных *(см. гл. 19)*. При необходимости обновить версию ADO, дистрибутив вашего приложения "потяжелеет" более чем на 2 Мбайт.

Новая технология доступа к данным dbExpress обеспечивает взаимодействие приложения с серверами баз данных. Драйверы dbExpress используют для получения данных исключительно запросы SQL. При этом на клиентской стороне отсутствует кэширование данных, вследствие этого здесь применяются исключительно однонаправленные курсоры и отсутствует возможность прямого редактирования наборов данных.

Примечание

Проблема редактирования данных в dbExpress может быть решена несколькими путями *(см. ниже)*. Однако любые предлагаемые способы повышают затраты на программирование и снижают эффективность полученного кода.

Взамен этих (весьма существенных для построения полноценных приложений) неудобств разработчики получили легкий и быстрый механизм доступа к данным.

Для функционирования компонентов dbExpress необходим только один драйвер, который взаимодействует напрямую с клиентским программным обеспечением для выбранного сервера БД. В поставку входят драйверы для первых четырех из списка серверов баз данных:

- DB2;
- InterBase;

- MySQL;
- Oracle;
- Microsoft SQL Server 2000.

Драйверы реализованы в виде динамических библиотек, а при необходимости могут быть прикомпилированы непосредственно к исполняемому файлу приложения. Поэтому проблема распространения совместно с приложением средств доступа к данным в случае с dbExpress снимается полностью. Естественно, на компьютере должно быть установлено клиентское ПО соответствующего SQL сервера.

Кроме того, технология dbExpress обеспечивает доступ к данным в кросс-платформенных приложениях для Windows и Linux, т. к. применяется и в Delphi и Kylix *(см. гл. 4)*, а способы ее применения идентичны.

Таким образом, технология dbExpress является наилучшим решением для приложений, в которых необходим быстрый и необременительный просмотр данных серверов SQL. И вряд ли он подойдет для сложных клиент-серверных или многоуровневых приложений, обеспечивающих серьезную работу с данными.

Технология dbExpress представляет собой совокупность драйверов, компонентов, инкапсулирующих соединения, транзакции, запросы и наборы данных, а также интерфейсов, обеспечивающих универсальный доступ к функциям dbExpress.

Компоненты dbExpress располагаются в Палитре компонентов на одноименной странице.

В этой главе рассматриваются следующие вопросы:

- настройка соединений с различными серверами баз данных, подключение драйверов и установка их параметров;
- способы использования компонентов dbExpress для просмотра данных и создание пользовательского интерфейса приложений;
- программная реализация редактирования данных;
- работа с данными в режиме кэширования изменений и использование компонента `TSimpleDataSet`;
- использование интерфейсов;
- распространение приложений с интегрированной технологией dbExpress.

Драйверы доступа к данным

Технология dbExpress обеспечивает доступ к серверу баз данных при помощи драйвера, реализованного как динамическая библиотека. Для каждого сервера имеется своя динамическая библиотека.

Таблица 17.1. *Драйверы dbExpress*

Сервер БД	Драйвер	Клиентское ПО
DB2	Dbexpdb2.dll.	Db2cli.dll
InterBase	Dbexpint.dll	GDS32.DLL
Informix	Dbexpinf.dll	Isqlb09a.dll
Microsoft SQL Server 2000	Dbexpmss.dll	OLE DB
MySQL	Dbexpmys.dll	LIBMYSQL.DLL
Oracle	Dbexpora.dll	OCI.DLL

Перечисленные в табл. 17.1 файлы находятся в папке \Delphi7\Bin.

Для доступа к данным сервера драйвер должен быть установлен на компьютере клиента. Для доступа к данным драйвер взаимодействует с клиентским ПО сервера, которое также должно быть инсталлировано на клиентской стороне.

Стандартные настройки для каждого драйвера хранятся в файле \Borland Shared\DBExpress\dbxdrivers.ini.

Соединение с сервером баз данных

Для создания соединения с сервером в рамках технологии dbExpress приложение должно использовать компонент `TSQLConnection`. Это обязательный компонент, все остальные компоненты связаны с ним и используют его для получения данных.

После переноса этого компонента в модуль данных или на форму необходимо выбрать тип сервера и настроить параметры соединения.

Свойство

```
property ConnectionName: string;
```

позволяет выбрать из выпадающего списка конкретное настроенное соединение. По умолчанию разработчику доступно по одному настроенному соединению для каждого сервера БД. После выбора соединения автоматически устанавливаются значения свойств:

- `property DriverName: string;`

  определяет используемый драйвер;

- `property LibraryName: string;`

  задает динамическую библиотеку драйвера dbExpress;

- `property VendorLib: string;`

  определяет динамическую библиотеку клиентского ПО сервера (табл. 17.1);

- `property Params: TStrings;`

  список этого свойства содержит настройки для выбранного соединения.

При необходимости все перечисленные свойства можно установить дополнительно.

Разработчик может дополнять и изменять список настроенных соединений. Для этого используется специализированный Редактор соединений **dbExpress Connections** (рис. 17.1). Он открывается после двойного щелчка на компоненте `TSQLConnection` или выбора команды **Edit Connection Properties** из всплывающего меню компонента.

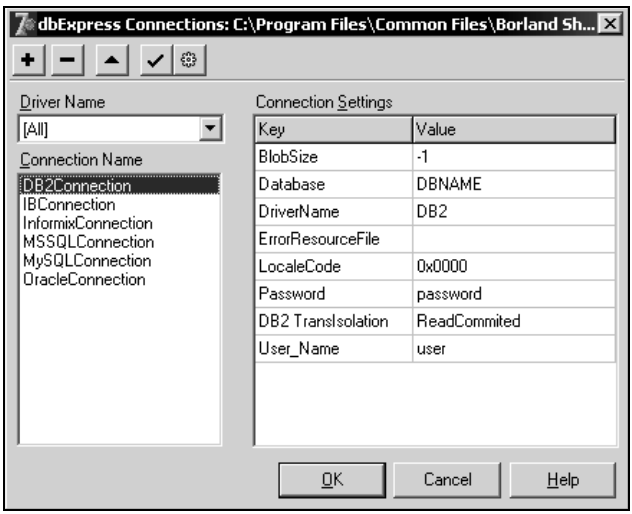


Рис. 17.1. Окно редактора настроенных соединений компонента `TSQLConnection`

В списке слева располагаются существующие соединения. В правой части для выбранного соединения отображаются текущие настройки. При помощи кнопок на Панели инструментов можно создавать, переименовывать и удалять соединения. Настройки также можно редактировать.

Список соединений в левой части соответствует списку выбора свойства `ConnectionName` в Инспекторе объектов. Настройки соединения из правой части отображаются в свойстве `Params`.

Настроенные соединения и их параметры сохраняются в файле \Borland Shared\DBExpress\dbxconnections.ini.

Примечание

При установке Delphi 7 поверх Delphi 6 сведения о соединениях из старого файла dbxconnections.ini добавляются в новый.

Конкретные значения настроек соединений зависят от используемого сервера БД и требований приложения (табл. 17.2).

Таблица 17.2. *Настройка соединения dbExpress*

Параметр	Значение
Общие настройки	
`BlobSize`	Задает ограничение на объем пакета данных для данных BLOB
`DriverName`	Имя драйвера
`ErrorResourceFile`	Файл сообщений об ошибках
`LocaleCode`	Код локализации, определяющий влияние национальных символов на сортировку данных
`User_Name`	Имя пользователя
`Password`	Пароль
DB2	
`Database`	Имя клиентского ядра
`DB2 TransIsolation`	Уровень изоляции транзакций
Informix	
`HostName`	Имя компьютера, на котором работает сервер Informix
`Informix TransIsolation`	Уровень изоляции транзакции
`Trim Char`	Определяет, нужно ли удалять из строковых значений полей пробелы, дополняющие значение до полной строки, заданной размером поля данных
Interbase	
`CommitRetain`	Задает поведение курсора по завершении транзакции. При значении `True` курсор обновляется, иначе — удаляется
`Database`	Имя файла базы данных (файл GDB)
`InterBase TransIsolation`	Уровень изоляции транзакции
`RoleName`	Роль пользователя
`SQLDialect`	Используемый диалект SQL. Для IntreBase 5 возможно только одно значение –1

Таблица 17.2 *(окончание)*

Параметр	Значение
`Trim Char`	Определяет, нужно ли удалять из строковых значений полей пробелы, дополняющие значение до полной строки, заданной размером поля данных
`WaitOnLocks`	Разрешение на ожидание занятых ресурсов
Microsoft SQL Server 2000	
`Database`	Имя базы данных
`HostName`	Имя компьютера, на котором работает сервер MS SQL Server 2000
`MSSQL TransIsolation`	Уровень изоляции транзакции
`OS Autenification`	Использование учетной записи текущего пользователя операционной системы (домена или Active Directory) при доступе к ресурсам сервера
MySQL	
`Database`	Имя базы данных
`HostName`	Имя компьютера, на котором работает сервер MySQL
Oracle	
`AutoCommit`	Флаг завершения транзакции. Устанавливается только сервером
`BlockingMode`	Задает режим завершения запроса. При значении `True` соединение дожидается окончания запроса (синхронный режим), иначе — начинает выполнение следующего (асинхронный режим)
`Database`	Запись базы данных в файле TNSNames.ora
`Multiple Transaction`	Поддержка управлением несколькими транзакциями в одной сессии
`Oracle TransIsolation`	Уровень изоляции транзакций
`OS Autenification`	Использование учетной записи текущего пользователя операционной системы (домена или Active Directory) при доступе к ресурсам сервера
`Trim Char`	Определяет, нужно ли удалять из строковых значений полей пробелы, дополняющие значение до полной строки, заданной размером поля данных

После выбора настроенного соединения или выбора типа сервера и настройки параметров соединения компонент `TSQLConnection` готов к работе.

Свойство

```
property Connected: Boolean;
```

открывает соединение с сервером при значении `True`. Аналогичную операцию выполняет метод

```
procedure Open;
```

После открытия соединения все компоненты dbExpress, инкапсулирующие наборы данных и связанные с открытым компонентом `TSQLConnection`, получают доступ к базе данных.

Соединение закрывается тем же свойством `Connected` или методом

```
procedure Close;
```

При открытии и закрытии соединения разработчик может использовать обработчики событий

```
property BeforeConnect: TNotifyEvent;
property AfterConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
```

Например, на стороне клиента можно организовать проверку пользователя приложения:

```
procedure TForm1.MyConnectionBeforeConnect(Sender: TObject);
begin
 if MyConnection.Params.Values['User_Name']) <> DefaultUser then
 begin
  MessageDlg('Wrong user name', mtError, [mbOK], 0);
  Abort;
 end;
end;
```

Свойство

```
property LoginPrompt: Boolean;
```

определяет, нужно ли отображать диалог авторизации пользователя перед открытием соединения.

О текущем состоянии соединения можно судить по значению свойства

```
TConnectionState = (csStateClosed, csStateOpen, csStateConnecting,
csStateExecuting, csStateFetching, csStateDisconnecting);
property ConnectionState: TConnectionState;
```

Параметры соединения можно настраивать на этапе разработки в Инспекторе объектов или Редакторе соединений (см. рис. 17.1). Также это можно

сделать и непосредственно перед открытием соединения, используя свойство `Params` или метод

```
procedure LoadParamsFromIniFile(AFileName : String = '');
```

который загружает заранее подготовленные параметры из INI-файла.

Проверить успешность этой операции можно при помощи свойства

```
property ParamsLoaded: Boolean;
```

значение `True` которого сигнализирует об успехе загрузки.

```
procedure TForm1.StartBtnClick(Sender: TObject);
begin
 if MyConnection.Params.Values['DriverName'] = '' then
 MyConnection.LoadParamsFromIniFile('c:\Temp\dbxalarmconnections.ini');
 if MyConnection.ParamsLoaded then
 try
  MyConnection.Open;
 except
  MessageDlg('Database connection error', mtError, [mbOK], 0);
 end;
end;
```

Управление наборами данных

Компонент `TSQLConnection` позволяет выполнять некоторые операции с подключенными наборами данных и следить за их состоянием.

Свойство

```
property DataSetCount: Integer;
```

возвращает число подключенных через данное соединение наборов данных.

Но это только активные наборы данных, переданные в связанные компоненты. Общее число выполняющихся в настоящий момент запросов возвращает свойство

```
property ActiveStatements: LongWord;
```

Если сервер БД установил для данного соединения максимальное число одновременно выполняющихся запросов, то оно доступно в свойстве

```
property MaxStmtsPerConn: LongWord;
```

Поэтому перед открытием набора данных можно выполнять следующий код, который повысит надежность приложения:

```
if MyQuery.SQLConnection.ActiveStatements <=
   MyQuery.SQLConnection.MaxStmtsPerConn
```

```
    then MyQuery.Open
    else MessageDlg('Database connection is busy', mtWarning, [mbOK], 0);
```

В случае возникновения непредвиденной ситуации все открытые через данное соединение наборы данных можно быстро закрыть методом

```
procedure CloseDataSets;
```

без разрыва соединения.

При необходимости компонент `TSQLConnection` может самостоятельно выполнять запросы SQL, не прибегая к помощи компонента `TSQLQuery` или `TSQLDataSet`. Для этого предназначена функция

```
function Execute(const SQL: string; Params: TParams;
ResultSet:Pointer=nil): Integer;
```

Если запрос должен содержать параметры, то необходимо сначала создать объект — список параметров `TParams` и заполнить его. При этом, т. к. объект `TParams` еще не связан с конкретным запросом, важен порядок следования параметров, который должен совпадать в списке `TParams` и в тексте SQL.

Если запрос возвращает результат, метод автоматически создает объект типа `TCustomSQLDataSet` и возвращает указатель на него в параметр `ResultSet`. Функция возвращает число обработанных запросом записей. Следующий фрагмент кода иллюстрирует применение функции `Execute`.

```
procedure TForm1.SendBtnClick(Sender: TObject);
var FParams: TParams;
    FDataSet: TSQLDataSet;
begin
 FParams := TParams.Create;
 try
  FParams.Items[0].AsInteger := 1234;
  FParams.Items[1].AsInteger := 6751;
  MyConnection.Execute('SELECT * FROM Orders WHERE OrderNo >= :Ord AND
  EmpNo = :Emp', FParams, FDataSet);
  if Assigned(FDataSet) then
  with FDataSet do
  begin
   Open;
   while Not EOF do
   begin
    {...}
    Next;
   end;
   Close;
  end;
```

```
 finally
  FParams.Free;
 end;
end;
```

Если запрос не имеет настраиваемых параметров и не возвращает набор данных, можно использовать функцию

```
function ExecuteDirect(const SQL: string): LongWord;
```

которая возвращает 0 в случае успешного выполнения запроса или код ошибки.

Метод

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
```

возвращает список таблиц базы данных. Параметр `SystemTables` позволяет включать в формируемый список `List` системные таблицы.

Метод `GetTableNames` дополнительно управляется свойством

```
TTableScope = (tsSynonym, tsSysTable, tsTable, tsView);
TTableScopes = set of TTableScope;
property TableScope: TTableScopes;
```

которое позволяет задать тип таблиц, имена которых попадают в список.

Для каждой таблицы можно получить список полей, использовав метод

```
procedure GetFieldNames(const TableName: String; List: TStrings);
```

и список индексов при помощи метода

```
procedure GetIndexNames(const TableName: string; List: TStrings);
```

В обоих методах список возвращаемых значений содержится в параметре `List`.

Аналогичным образом метод

```
procedure GetProcedureNames(List: TStrings);
```

возвращает список доступных хранимых процедур, а метод

```
procedure GetProcedureParams(ProcedureName: String; List: TList);
```

определяет параметры отдельной процедуры.

Транзакции

Подобно своим аналогам в BDE и ADO компонент `TSQLConnection` поддерживает механизм транзакций и делает это сходным образом.

Начало, фиксацию и откат транзакции выполняют методы

```
procedure StartTransaction(TransDesc: TTransactionDesc);
procedure Commit(TransDesc: TTransactionDesc);
procedure Rollback(TransDesc: TTransactionDesc);
```

При этом запись `TTransactionDesc` возвращает параметры транзакции:

```
TTransIsolationLevel = (xilDIRTYREAD, xilREADCOMMITTED,
xilREPEATABLEREAD, xilCUSTOM);
  TTransactionDesc = packed record
     TransactionID    : LongWord;
     GlobalID         : LongWord;
     IsolationLevel   : TTransIsolationLevel;
     CustomIsolation  : LongWord;
     end;
```

Запись содержит уникальный в рамках соединения идентификатор транзакции `TransactionID` и уровень изоляции транзакции `IsolationLevel`. При уровне изоляции `xilCUSTOM` определяется параметр `CustomIsolation`. Идентификатор `GlobalID` используется при работе с сервером Oracle.

Некоторые серверы БД не поддерживают транзакции, и для определения этого факта используется свойство

```
property TransactionsSupported: LongBool;
```

Если соединение уже находится в транзакции, свойству

```
property InTransaction: Boolean;
```

присваивается значение `True`. Поэтому, если сервер не поддерживает множественные транзакции, всегда полезно убедиться, что соединение не обслуживает начатую транзакцию:

```
var TransInfo: TTransactionDesc;
{...}
if Not MyConnection.InTransaction then
 try
MyConnection.StartTransaction(TransInfo);
{...}
MyConnection.Commit(TransInfo);
except
MyConnection.Rollback(TransInfo);
end;
```

Использование компонентов наборов данных

Набор компонентов dbExpress, инкапсулирующих набор данных, вполне обычен и сравним с аналогичными компонентами BDE, ADO, InterBase Express. Это компоненты `TSQLDataSet`, `TSQLTable`, `TSQLQuery`, `TSQLStoredProc`.

Примечание

Компонент `TSimpleDataSet` также относится к рассматриваемой группе, но т. к. он обладает рядом специфических возможностей, его описание вынесено в отдельный пункт.

Однако необходимость создания легкой технологии доступа к данным, какой является dbExpress, наложила на эти компоненты ряд ограничений.

Хотя общим предком всех рассматриваемых здесь компонентов является класс `TDataSet` (см. рис. 12.1), который обладает полным инструментарием для работы с набором данных, компоненты dbExpress используют только однонаправленные курсоры и не позволяют редактировать данные. Однонаправленные курсоры ограничивают навигацию по набору данных и обеспечивают перемещение только на следующую запись и возврат на первую. Также здесь недоступны любые операции, требующие буферизации данных — это поиск, фильтрация, синхронный просмотр.

Таким образом, для выключения ряда механизмов класса `TDataSet` понадобился еще один промежуточный класс `TCustomSQLDataSet`.

Отображение данных при помощи компонентов со страницы **Data Controls** также ограничено. Нельзя использовать компоненты `TDBGrid` и `TDBCtrlGrid`, а в компоненте `TDBNavigator` не забудьте отключить кнопки возврата на одну позицию назад и перехода на последнюю запись. Также ничего хорошего не получится из попытки применить компоненты синхронного просмотра. Остальные компоненты можно использовать обычным способом *(см. гл. 15)*.

Основные способы применения компонентов dbExpress остаются стандартными и подробно описаны в *части III*.

Доступ к данным во всех рассматриваемых компонентах осуществляется одинаково — через соединение, инкапсулированное компонентом `TSQLConnection`. Привязывание к соединению выполняет свойство

```
property SQLConnection: TSQLConnection;
```

Рассмотрим теперь компоненты dbExpress подробнее.

Класс *TCustomSQLDataSet*

Так как общим предком компонентов dbExpress объявлен класс `TDataSet`, то задачей класса `TCustomSQLDataSet` является не столько внесение новой

функциональности, сколько корректное ограничение возможностей, заложенных в `TDataSet`. Непосредственно в приложениях этот класс не используется, но информация о нем полезна для понимания других компонентов dbExpress и для создания собственных компонентов на его основе.

Класс `TCustomSQLDataSet` является общим предком для компонентов, инкапсулирующих запросы, таблицы и хранимые процедуры. Для их поддержки используются свойства:

- `TSQLCommandType = (ctQuery, ctTable, ctStoredProc);`
  `property CommandType: TSQLCommandType;`

  определяющее тип команды, направляемой серверу;

- `property CommandText: string;`

  содержащее текст команды.

Если серверу передается запрос SQL (`CommandType = ctQuery`), свойство `CommandText` содержит текст запроса. Если это команда на получение таблицы, свойство `CommandText` содержит имя таблицы, а далее с использованием имени таблицы создается запрос SQL на получение всех полей этой таблицы. Если необходимо выполнить процедуру, свойство `CommandText` содержит имя этой процедуры.

Текст команды, которая реально передается на сервер для выполнения, содержится в защищенном свойстве

```
property NativeCommand: string;
```

Для использования в табличном представлении существует свойство

```
property SortFieldNames: string;
```

определяющее порядок сортировки записей табличного набора данных. Свойство должно содержать список полей, разделенных точкой с запятой. Это свойство используется для создания выражения `ORDER BY` для генерируемой команды.

Для обработки исключительных ситуаций в классах — потомках может быть использовано защищенное свойство

```
property LastError: string;
```

которое возвращает текст последней ошибки dbExpress.

Для ускорения работы набора данных можно отключить получение от сервера метаданных об объекте запроса (таблицы, процедуры, полей, индексов), которые обычно направляются клиенту вместе с результатом запроса. Для этого свойству

```
property NoMetadata: Boolean;
```

присваивается значение `True`.

Однако пользоваться им нужно осторожно, т. к. для некоторых видов команд метаданные необходимы (это операции с использованием индексов).

Разработчик может управлять процессом получения метаданных. Для этого необходимо заполнить структуру

```
TSchemaType = (stNoSchema, stTables, stSysTables, stProcedures,
stColumns, stProcedureParams, stIndexes);
  TSchemaInfo = record
    FType      : TSchemaType;
    ObjectName : String;
    Pattern    : String;
  end;
```

которая доступна через защищенное свойство

```
property SchemaInfo: TSQLSchemaInfo;
```

а значит, может использоваться только при создании новых компонентов на основе `TCustomSQLDataSet`.

Параметр `FType` определяет тип требуемой информации. Параметр `ObjectName` — имя таблицы или хранимой процедуры, если в параметре `FType` указаны поля, индексы или параметры процедур.

Внимание!

Если компонент должен получать результирующий набор данных, параметр `FType` должен обязательно иметь значение `stNoSchema`. При изменении значения свойства `CommandText` это условие выполняется автоматически.

Параметр `Pattern` определяет, какие ограничения накладываются на метаданные. Он содержит символьную маску, подобную свойству `Mask` многих визуальных компонентов. Последовательность символов маски обозначается символом `%`, единичный символ определяется символом `_`.

При необходимости использовать управляющие символы в качестве маскирующих, применяются двойные символы `%%` и `__`.

Подобно свойству `Tag` класса `TComponent`, класс `TCustomSQLDataSet` имеет строковое свойство

```
property DesignerData: string
```

в котором разработчик может хранить любую служебную информацию. По существу, это просто лишняя строковая переменная, которую нет необходимости объявлять.

Компонент *TSQLDataSet*

Компонент `TSQLDataSet` является универсальным и позволяет выполнять запросы SQL (подобно `TSQLQuery`), просматривать таблицы целиком (подобно `TSQLTable`) или выполнять хранимые процедуры (подобно `TSQLStoredProc`).

Для определения режима работы компонента используется свойство `CommandType` *(см. выше)*.

Если ему присвоить значение `ctTable`, в списке свойства `CommandText` можно выбрать имя таблицы, если конечно компонент подключен к соединению. При выборе значения `ctQuery` в свойстве `CommandText` необходимо определить текст запроса SQL. Для работы в режиме хранимой процедуры для свойства `CommandType` используется значение `ctStoredProc`, а в списке свойства `CommandText` можно выбрать нужную процедуру.

Для открытия набора данных используются традиционные способы: свойство `Active` или метод `Open`. Если же запрос SQL или хранимая процедура не возвращают набор данных, для их выполнения используется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect` определяет, необходимо ли произвести подготовку параметров перед выполнением команды. Если параметры запроса или процедуры существуют, параметр `ExecDirect` должен иметь значение `False`.

Дополнительно для табличного режима можно использовать свойство `SortFieldNames` *(см. выше)*, определяющее порядок сортировки записей таблицы.

В режиме запросов и хранимых процедур для задания параметров используются свойства `Params` и `ParamCheck` *(см. часть III)*.

Информация об используемых в результирующем наборе данных индексах сохраняется в свойстве

```
property IndexDefs: TIndexDefs;
```

Компонент *TSQLTable*

Компонент `TSQLTable` предназначен для просмотра таблиц целиком и по основным функциям подобен своим аналогам `TTable`, `TADOTable`, `TIBTable` *(подробнее о функциях компонентов таблиц см. часть III)*.

Для получения табличного набора данных компонент `TSQLTable` самостоятельно формирует запрос на сервер, используя для этого возможности, унаследованные от предка `TCustomSQLDataSet`.

Метод

```
procedure PrepareStatement; override;
```

генерирует для выбранной таблицы текст запроса, который формируется компонентом для передачи на сервер.

Для определения имени таблицы используется свойство `TableName`, и, если компонент подключен к соединению, имя таблицы можно выбрать из списка.

Для подключения простых или составных индексов используются свойства `IndexFieldNames`, `IndexFields`, `IndexName`. А метод

```
procedure GetIndexNames(List: TStrings);
```

возвращает в параметр `List` список используемых индексов.

Связь между двумя наборами данных главный/подчиненный организуется свойствами `MasterFields`, `MasterSource`.

Компонент `TSQLTable` предоставляет разработчику некоторое подобие функций редактирования. Для удаления всех записей из связанной с компонентом таблицы на сервере используется метод

```
procedure DeleteRecords;
```

Компонент *TSQLQuery*

Компонент `TSQLQuery` повторяет функциональность своих аналогов в BDE, ADO, InterBase Express и позволяет выполнять на сервере запросы SQL клиента. Подробнее о функциях компонентов запросов SQL *см. часть III*.

Текст запроса содержится в свойстве

```
property SQL: TStrings;
```

а его простое строковое представление в свойстве

```
property Text: string;
```

Если запрос возвращает набор данных, его выполнение осуществляется свойством `Active` или методом `Open`. В противном случае используется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect = False` означает, что запрос не имеет настраиваемых параметров.

Компонент *TSQLStoredProc*

Компонент `TSQLStoredProc` инкапсулирует функциональность хранимых процедур для их выполнения в рамках технологии dbExpress. Он подобен другим своим аналогам. Подробнее о функциях компонентов хранимых процедур *см. часть III*.

Имя хранимой процедуры определяется свойством

```
property StoredProcName: string;
```

Для работы с входными и выходными параметрами предназначено свойство

```
property Params: TParams;
```

Внимание!

При работе с параметрами желательно использовать обращение к конкретному параметру по имени при помощи метода `ParamByName`. При работе с некоторыми серверами порядок следования параметров до выполнения процедуры и после может изменяться.

Процедура выполняется методом

```
function ExecProc: Integer; virtual;
```

если она не возвращает набор данных. Иначе используются свойство `Active` или метод `Open`.

Если хранимая процедура возвращает несколько связанных наборов данных (подобно иерархическим запросам ADO), доступ к следующему набору данных осуществляет метод

```
function NextRecordSet: TCustomSQLDataSet;
```

автоматически создавая объект типа `TCustomSQLDataSet` для инкапсуляции новых данных. Возврат к предыдущему набору данных возможен, если вы определили объектные переменные для каждого набора данных:

```
var SecondSet: TCustomSQLDataSet;
...
 MyProc.Open;
 while Not MyProc.Eof do
 begin
  {...}
  Next;
 end;
 SecondSet := MyProc.NextRecordSet;
 SecondSet.Open;
 {...}
 SecondSet.Close;
 MyProc.Close;
```

Компонент *TSimpleDataSet*

Компонент `TSimpleDataSet` обеспечивает кэширование полученных данных и сделанных изменений на стороне клиента и последующую передачу их на сервер для фиксации. В отличие от компонента `TClientDataSet`, основным назначением которого является обслуживание набора данных, полученного

от удаленного сервера при помощи серверных компонентов `DataSnap`, компонент `TSimpleDataSet` призван быть лишь средством редактирования набора данных в технологии dbExpress.

Компонент использует двунаправленный курсор и позволяет редактировать данные, правда только в режиме кэширования *(см. гл. 22)*.

Таким образом, компонент `TSimpleDataSet` позволяет исправить основные недостатки технологии dbExpress.

Для подключения к источнику данных компонент использует свойство

```
property DBConnection: TSQLConnection;
```

которое позволяет связать его с соединением `TSQLConnection` *(см. выше)*. Или свойство

```
property ConnectionName: string;
```

которое позволяет выбрать тип соединения dbExpress напрямую.

При этом у компонента отсутствует механизм создания удаленного доступа к данным, представленный у компонента `TClientDataSet` свойствами `RemoteServer` и `ProviderName`.

После создания соединения с сервером БД можно определить тип используемой команды, подобно компоненту `TSQLDataSet`.

Тип команды определяется свойством

```
TSQLCommandType = (ctQuery, ctTable, ctStoredProc);
property CommandType: TSQLCommandType;
```

А содержание команды задает свойство

```
property CommandText: string;
```

После этого компонент можно связывать с компонентами отображения данных, просматривать и редактировать данные.

Для передачи на сервер сделанных и сохраненных в локальном кэше изменений используется метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

где параметр `MaxErrors` определяет максимально возможное число ошибок при сохранении. Обычно этому параметру присваивается −1, что снимает ограничение на число ошибок.

Метод

```
function Reconcile(const Results: OleVariant): Boolean;
```

очищает локальный кэш компонента от записей, которые успешно сохранены на сервере.

Отменить локальные изменения можно методом

```
procedure CancelUpdates;
```

Обратите внимание, что в компоненте действуют традиционные методы набора данных `Edit`, `Post`, `Cancel`, `Apply`, `Insert`, `Delete`. Но они оказывают влияние только на записи, кэшированные локально. Вы можете сколько угодно редактировать набор данных при помощи перечисленных методов, но они будут изменять только содержимое кэша. Настоящее сохранение на сервере осуществляется методом `ApplyUpdates`.

Данные между сервером и компонентом пересылаются пакетами.

Доступ к текущему пакету возможен при помощи свойства

```
property Data: OleVariant;
```

Сделанные изменения содержатся в свойстве

```
property Delta: OleVariant;
```

При этом разработчик может регулировать размер пакетов. Например, при ухудшении соединения можно уменьшить размер пакетов. Размер пакета определяется свойством

```
property PacketRecords: Integer;
```

которое задает число записей в пакете. Автоматическое назначение пакетов включается

```
PacketRecords := -1
```

Если значение `PacketRecords` равно 0, между клиентом и сервером пересылаются только метаданные.

Если свойство `PacketRecords` больше нуля, то необходимо вручную организовывать подкачку данных с сервера. Для этого используется метод

```
function GetNextPacket: Integer;
```

Для организации такой подкачки вполне подойдут методы-обработчики событий

```
property BeforeGetRecords: TRemoteEvent;
property AfterGetRecords: TRemoteEvent;
```

В компоненте `TSimpleDataSet` развиты средства работы с одиночными записями. Можно просмотреть общее число записей

```
property RecordCount: Integer;
```

и номер текущей записи

```
property RecNo: Integer;
```

Размер одной записи сохраняется в свойстве

```
property RecordSize: Word;
```

Все изменения, сделанные в текущей записи, отменяются методом

```
procedure RevertRecord;
```

Обновить значение полей для текущей записи с сервера можно методом

```
procedure RefreshRecord;
```

Обработка исключительных ситуаций для компонента `TSimpleDataSet` состоит из двух этапов.

Во-первых, необходимо отслеживать ошибки на стороне клиента — это могут быть некорректный ввод данных, ошибки кэширования и т. д. В этом случае подходят все стандартные способы, применяемые для наборов данных.

Во-вторых, ошибки могут возникнуть при сохранении изменений на сервере. И поскольку само событие, приведшее к исключительной ситуации, возникает на другом компьютере или в другом процессе, для отслеживания таких ошибок используется специальный метод-обработчик

```
TReconcileErrorEvent = procedure(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction)
of object;
property OnReconcileError: TReconcileErrorEvent;
```

который срабатывает, если с сервера пересылается сообщение об ошибке. Информация об ошибке находится в параметре `E: EReconcileError`.

Более детальная информация о клиентских наборах данных содержится в *гл. 22*.

Способы редактирования данных

Несмотря на декларированные недостатки технологии dbExpress — однонаправленные курсоры и невозможность редактирования — существуют программные способы уменьшить масштаб проблемы или даже решить ее.

Во-первых, в нашем распоряжении имеется компонент `TSimpleDataSet`, который реализует двунаправленный курсор и обеспечивает редактирование данных путем их кэширования на клиентской стороне.

Во-вторых, редактирование можно обеспечить настройкой и выполнением запросов SQL `INSERT`, `UPDATE` и `DELETE`.

У каждого способа есть свои преимущества и недостатки.

Компонент `TSimpleDataSet` безусловно хорош. Он технологичен, относительно прост в использовании и, главное, прячет всю функциональность за

несколькими свойствами и методами. Но локальное кэширование изменений подходит далеко не для всех приложений.

Например, при многопользовательском интенсивном доступе к данным с их редактированием при локальном кэшировании могут возникнуть проблемы с целостностью и адекватностью данных. Самый распространенный пример: продавец в строительном супермаркете, обслуживая покупателя в пик летних ремонтов, резервирует несколько наименований ходовых товаров. Но покупатель замешкался, выбирая обои и плитку. А за это время другой продавец уже продал другому покупателю (заказ первого покупателя еще находится в локальном кэше!) часть его товаров.

Конечно, фиксацию изменений на сервере можно выполнять после локального сохранения каждой записи, но это приведет к загрузке соединения и снижению эффективности системы.

Использование модифицирующих запросов, с одной стороны, позволяет оперативно вносить изменения в данные на сервере, а с другой — требует больших затрат на программирование и отладку. Сложность кода в этом случае существенно выше.

Рассмотрим небольшой пример реализации обоих способов. Приложение Demo DBX использует соединение с сервером InterBase. Подключена тестовая база данных \Borland Shared\Data\MastSQL.gdb.

Листинг 17.1. Пример приложения dbExpress с редактируемыми наборами данных

```
implementation

{$R *.dfm}

procedure TfmDemoDBX.FormCreate(Sender: TObject);
begin
 tblVens.Open;
 cdsCusts.Open;
end;

procedure TfmDemoDBX.FormDestroy(Sender: TObject);
begin
 tblVens.Close;
 cdsCusts.Close;
end;

{Editing feature with updating query}

procedure TfmDemoDBX.tblVensAfterScroll(DataSet: TDataSet);
begin
```

```
 edVenNo.Text := tblVens.FieldByName('VENDORNO').AsString;
 edVenName.Text := tblVens.FieldByName('VENDORNAME').AsString;
 edVenAdr.Text := tblVens.FieldByName('ADDRESS1').AsString;
 edVenCity.Text := tblVens.FieldByName('CITY').AsString;
 edVenPhone.Text := tblVens.FieldByName('PHONE').AsString;
end;

procedure TfmDemoDBX.sbCancelClick(Sender: TObject);
begin
 tblVens.First;
end;

procedure TfmDemoDBX.sbNextClick(Sender: TObject);
begin
 tblVens.Next;
end;

procedure TfmDemoDBX.sbPostClick(Sender: TObject);
begin
 with quUpdate do
 try
  ParamByName('Idx').AsInteger :=
tblVens.FieldByName('VENDORNO').AsInteger;
  ParamByName('No').AsString := edVenNo.Text;
  ParamByName('Name').AsString := edVenName.Text;
  ParamByName('Adr').AsString :=  edVenAdr.Text;
  ParamByName('City').AsString :=  edVenCity.Text;
  ParamByName('Phone').AsString :=  edVenPhone.Text;
  ExecSQL;
 except
  MessageDlg('Vendor''s info post error', mtError, [mbOK], 0);
  tblVens.First;
 end;
end;

{Editing feature with cached updates}

procedure TfmDemoDBX.cdsCustsAfterPost(DataSet: TDataSet);
begin
 cdsCusts.ApplyUpdates(-1);
end;

procedure TfmDemoDBX.cdsCustsReconcileError(DataSet:
TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
```

```
begin
 MessageDlg('Customer''s info post error', mtError, [mbOK], 0);
 cdsCusts.CancelUpdates;
end;

end.
```

Для просмотра и редактирования выбраны таблицы Vendors и Customers. Первая таблица подключена через настроенное соединение (компонент `cnMast`) к компоненту `tblVens` типа `TSQLTable`. Значение пяти полей отображается в обычных компонентах `TEdit`, т. к. компоненты отображения данных, связанные с компонентом dbExpress через компонент `TDataSource`, работают только в режиме просмотра, не позволяя редактировать данные (рис. 17.2).

Использование метода-обработчика `AfterScroll` позволило легко решить проблему заполнения компонентов `TEdit` при навигации по набору данных. Для сохранения сделанных изменений (нажатие на кнопку `sbPost`) используется компонент `quUpdate` типа `TSQLQuery`. В параметрах запроса передаются текущие значения полей из компонентов `TEdit`. Так как в этом случае работает однонаправленный курсор, проблема обновления набора данных после выполнения модифицирующего запроса не возникает и набор данных обновляется только при вызове метода `First` компонента `tblVens`.

Вторая таблица подключена через тот же компонент `cnMast` к компоненту `cdsCusts` типа `TSimpleDataSet`. Он работает в табличном режиме. Данные отображаются в обычном компоненте `TDBGrid`.

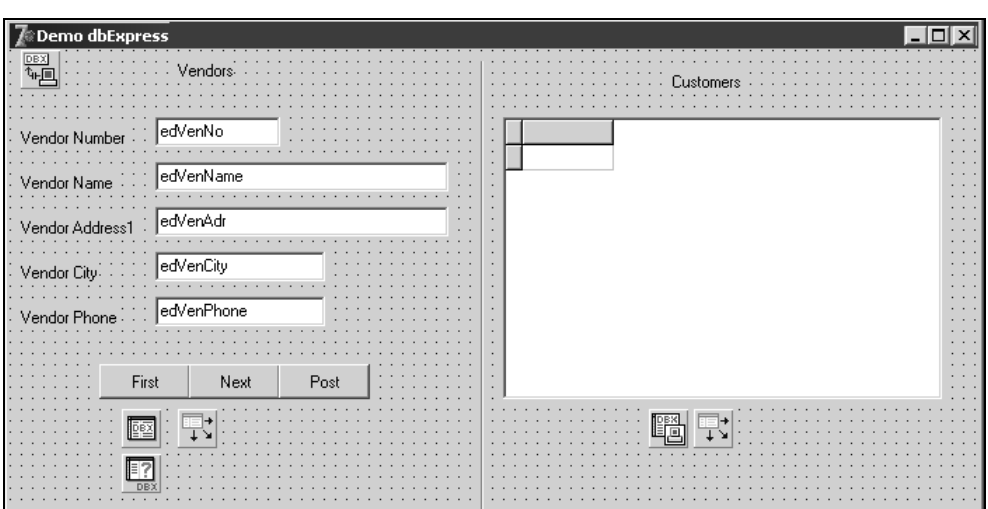


Рис. 17.2. Окно приложения Demo dbExpress

Для сохранения сделанных изменений здесь использован метод `ApplyUpdates`, размещенный в методе-обработчике `AfterPost`, когда изменения уже попали в локальный кэш. Метод-обработчик вызывается каждый раз при переходе в компонент `TDBGrid` на новую строку.

Для компонента `cdsCusts` также предусмотрена простейшая обработка исключительных ситуаций, возникающих на сервере.

Обратите также внимание на настройку компонента `cnMast` типа `TSQLConnection`. Свойства `KeepConnection` и `LoginPrompt` со значениями `False` обеспечивают открытие наборов данных при создании формы и автоматическое закрытие соединения при закрытии приложения с минимальным исходным кодом.

Интерфейсы dbExpress

Технология dbExpress основана на использовании четырех базовых интерфейсов, методы которых применяются во всех компонентах dbExpress. При серьезной работе с технологией или при проектировании собственных компонентов информация об этих интерфейсах будет полезна.

Интерфейс *ISQLDriver*

Интерфейс `ISQLDriver` инкапсулирует всего три метода для обслуживания драйвера dbExpress. Экземпляр интерфейса создается для соединения и обеспечивает его связь с драйвером.

Методы

```
function SetOption(eDOption: TSQLDriverOption; PropValue: LongInt):
SQLResult; stdcall;
function GetOption(eDOption: TSQLDriverOption; PropValue: Pointer;
MaxLength: SmallInt; out Length: SmallInt): SQLResult; stdcall;
```

позволяют работать с параметрами драйвера. А метод

```
function getSQLConnection(out pConn: ISQLConnection): SQLResult; stdcall;
```

возвращает указатель на интерфейс связанного с драйвером соединения `ISQLConnection`.

Получить доступ к интерфейсу `ISQLDriver` разработчик может, использовав защищенное свойство

```
property Driver: ISQLDriver read FSQLDriver;
```

компонента `TSQLConnection`.

Интерфейс *ISQLConnection*

Интерфейс `ISQLConnection` обеспечивает работу соединения. Он передает запросы серверу и возвращает результаты, создавая экземпляры интерфейса `ISQLCommand`; управляет транзакциями; поддерживает передачу метаданных при помощи интерфейса `ISQLMetaData`.

Для открытия соединения используется метод

```
function connect(ServerName: PChar; UserName: PChar; Password: PChar):
SQLResult; stdcall;
```

где `ServerName` — имя базы данных, `UserName` и `Password` — имя и пароль пользователя.

Закрывает соединение метод

```
function disconnect: SQLResult; stdcall;
```

Параметры соединения управляются методами

```
function SetOption(eConnectOption: TSQLConnectionOption; lValue:
LongInt): SQLResult; stdcall;
function GetOption(eDOption: TSQLConnectionOption; PropValue: Pointer;
MaxLength: SmallInt; out Length: SmallInt): SQLResult; stdcall;
```

Для обработки запроса, проходящего через соединение, создается интерфейс `ISQLCommand`

```
function getSQLCommand(out pComm: ISQLCommand): SQLResult; stdcall;
```

Обработка транзакций осуществляется тремя методами:

```
function beginTransaction(TranID: LongWord): SQLResult; stdcall;
function commit(TranID: LongWord): SQLResult; stdcall;
function rollback(TranID: LongWord): SQLResult; stdcall;
```

При помощи метода

```
function getErrorMessage(Error: PChar): SQLResult; overload; stdcall;
```

организована обработка исключительных ситуаций в компоненте `TSQLConnection`. В нем реализована защищенная процедура `SQLError`, которую можно использовать в собственных компонентах и при необходимости дорабатывать.

Например, можно написать собственную процедуру контроля ошибок примерно по такому образцу:

```
procedure CheckError(IConn: ISQLConnection);
var FStatus: SQLResult;
    FSize:SmallInt;
    FMessage: pChar;
```

```
begin
 FStatus := IConn.getErrorMessageLen(FSize);
 if (FStatus = SQL_SUCCESS)and(FSize > 0) then
 begin
  FMessage := AllocMem(FSize + 1);
  FStatus := IConn.getErrorMessage(FMessage);
  if FStatus = SQL_SUCCESS
   then MessageDlg(FMessage, mtError, [mbOK], 0)
   else MessageDlg('Checking error', mtWarning, [mbOK], 0);
  if Assigned(FMessage)
   then FreeMem(FMessage);
 end;
end;
```

Доступ к интерфейсу `ISQLConnection` можно получить через свойство

```
property SQLConnection: ISQLConnection;
```

компонента `TSQLConnection`.

Интерфейс *ISQLCommand*

Интерфейс `ISQLCommand` обеспечивает функционирование запроса dbExpress. Компоненты dbExpress, работающие с наборами данных, используют его для реализации своих методов.

Параметры запроса устанавливаются методом

```
function setParameter(ulParameter: Word; ulChildPos: Word; eParamType:
TSTMTParamType; uLogType: Word; uSubType: Word; iPrecision: Integer;
iScale: Integer; Length: LongWord; pBuffer: Pointer; lInd: Integer):
SQLResult; stdcall;
```

где `ulParameter` — порядковый номер параметра; если параметр является дочерним для сложных типов данных, `ulChildPos` задает его порядковый номер; `eParamType` задает тип параметра (входной, выходной, смешанный); `uLogType` — тип данных параметра; `uSubType` — вспомогательный параметр типа данных; `iScale` — максимальный размер значения в байтах; `iPrecision` — максимальная точность типа данных; `Length` — размер буфера; `pBuffer` — буфер, содержащий значение параметра; `lInd` — флаг, определяющий, может ли параметр иметь нулевое значение.

Для каждого параметра метод вызывается снова.

Информацию о параметре можно получить, используя метод

```
function getParameter(ParameterNumber: Word; ulChildPos: Word; Value:
Pointer; Length: Integer; var IsBlank: Integer): SQLResult; stdcall;
```

где `ParameterNumber` — порядковый номер параметра; если параметр является дочерним для сложных типов данных, `ulChildPos` задает его порядковый номер; `Value` — указатель на буфер значения параметра; `Length` — размер буфера; `IsBlank` — признак незаполненного параметра.

Метод

```
function prepare(SQL: PChar; ParamCount: Word): SQLResult; stdcall;
```

готовит запрос к выполнению с учетом значений параметров.

Выполнение запроса осуществляется методом

```
function execute(var Cursor: ISQLCursor): SQLResult; stdcall;
```

который возвращает в параметре интерфейс курсора, если запрос выполнен.

Или метод

```
function executeImmediate(SQL: PChar; var Cursor: ISQLCursor): SQLResult;
stdcall;
```

который выполняет запрос, не требующий подготовки (не имеющий параметров). Он также возвращает в параметре `Cursor` готовый интерфейс курсора, если запрос выполнен успешно. Текст запроса определяется параметром `SQL`.

И метод

```
function getNextCursor(var Cursor: ISQLCursor): SQLResult; stdcall;
```

определяет в параметре `Cursor` курсор следующего набора данных, если выполнялась хранимая процедура, которая возвращает несколько наборов данных.

Интерфейс `ISQLCommand` используется компонентом `TCustomSQLDataSet` и недоступен потомкам.

Интерфейс *ISQLCursor*

Интерфейс `ISQLCursor` обладает совокупностью методов, которые помогут получить информацию о полях курсора, а также значения этих полей. Все эти методы имеют одинаковое представление. Для получения нужной информации необходимо задать порядковый номер поля в структуре курсора.

Метод

```
function next: SQLResult; stdcall;
```

обновляет курсор, занося в него информацию из следующей строки набора данных.

Интерфейс `ISQLCursor` используется компонентом `TCustomSQLDataSet` и недоступен потомкам.

Отладка приложений с технологией dbExpress

Наряду с обычными методами отладки исходного кода, в dbExpress существует возможность контроля запросов, проходящих на сервер через соединение. Для этого используется компонент `TSQLMonitor`.

Через свойство

```
property SQLConnection: TSQLConnection;
```

компонент связывается с отлаживаемым соединением.

Затем компонент включается установкой `Active = True`.

Теперь во время выполнения приложения сразу после открытия соединения свойство

```
property TraceList: TStrings;
```

будет заполняться информацией обо всех проходящих командах.

Содержимое этого списка можно сохранить в файле при помощи метода

```
procedure SaveToFile(AFileName: string);
```

Эту же информацию можно автоматически добавлять в текстовый файл, определяемый свойством

```
property FileName: string;
```

но только тогда, когда свойство

```
property AutoSave: Boolean;
```

будет иметь значение `True`.

Свойство

```
property MaxTraceCount: Integer;
```

определяет максимальное число контролируемых команд, а также управляет процессом контроля. При значении −1 ограничения снимаются, а при значении 0 контроль останавливается.

Текущее число проверенных команд содержится в свойстве

```
property TraceCount: Integer;
```

Перед записью команды в список вызывается метод-обработчик

```
TTraceEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc;
var LogTrace: Boolean) of object;
property OnTrace: TTraceEvent;
```

а сразу после записи в список вызывается

```
TTraceLogEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc)
of object;
property OnLogTrace: TTraceLogEvent;
```

Таким образом, разработчик получает компактный и симпатичный компонент, позволяющий без усилий получать информацию о прохождении команд в соединении.

Если же компонент `TSQLMonitor` не подходит, можно воспользоваться методом

```
procedure SetTraceCallbackEvent(Event: TSQLCallbackEvent; IClientInfo:
Integer);
```

компонента `TSQLConnection`. Параметр процедурного типа `Event` определяет функцию, которая будет вызываться при выполнении каждой команды. Параметр `IClientInfo` должен содержать любое число.

Он позволяет разработчику самостоятельно определить функцию типа `TSQLCallbackEvent`:

```
TRACECat = TypedEnum;
TSQLCallbackEvent = function(CallType: TRACECat; CBInfo: Pointer):
CBRType; stdcall;
```

Эта функция будет вызываться каждый раз при прохождении команды. Текст команды будет передаваться в буфер `CBInfo`. Разработчику необходимо лишь выполнить запланированные действия с буфером внутри функции.

Рассмотрим в качестве примера следующий исходный код.

```
function GetTraceInfo(CallType: TRACECat; CBInfo: Pointer): CBRType;
stdcall;
begin
 if Assigned(Form1.TraceList) then Form1.TraceList.Add(pChar(CBinfo));
end;

procedure TForm1.MyConnectionBeforeConnect(Sender: TObject);
begin
 TraceList := TStringList.Create;
end;

procedure TForm1.MyConnectionAfterDisconnect(Sender: TObject);
begin
 if Assigned(TraceList) then
 begin
  TraceList.SaveToFile('c:\Temp\TraceInfo.txt');
```

```
  TraceList.Free;
 end;
end;

procedure TForm1.StartBtnClick(Sender: TObject);
begin
  MyConnection.SetTraceCallbackEvent(GetTraceInfo, 8);
  MyConnection.Open;
  {...}
  MyConnection.Close;
end;
```

Перед открытием соединения в методе-обработчике `BeforeConnection` создается объект типа `TStringList`. После закрытия соединения этот объект сохраняется в файле и уничтожается.

Перед открытием соединения (метод-обработчик нажатия кнопки Start) при помощи метода `SetTraceCallbackEvent` с соединением связывается функция `GetTraceInfo`.

Таким образом, по мере прохождения команд информация о них будет накапливаться в списке. После закрытия соединения список сохраняется в текстовом файле.

Примечание

В своей работе компонент `TSQLMonitor` также использует вызовы метода `SetTraceCallbackEvent`. Поэтому одновременно применять компонент и собственные функции нельзя.

Распространение приложений с технологией dbExpress

Готовое приложение, использующее технологию dbExpress, можно поставлять заказчикам двумя способами.

Вместе с приложением поставляется динамическая библиотека для выбранного сервера (см. колонку "Драйвер" табл. 17.1). Она находится в папке \Delphi7\Bin.

Дополнительно, если в приложении используется компонент `TSimpleDataSet`, необходимо включить в поставку динамическую библиотеку Midas.dll.

Приложение компилируется вместе со следующими DCU-файлами: dbExpInt.dcu, dbExpOra.dcu, dbExpDb2.dcu, dbExpMy.dcu (в зависимости от выбранного сервера). Если в приложении используется компонент `TSimpleDataSet`, следует добавить файлы Crtl.dcu и MidasLib.dcu. В результате необходимо поставлять только исполняемый файл приложения.

Если дополнительная настройка соединений не требуется, файл dbxconnections.ini не нужен.

Резюме

Технология dbExpress предназначена для создания приложений, требующих быстрого доступа к базам данных, хранящимся на серверах SQL. Доступ осуществляется при помощи небольших драйверов, реализованных в виде динамических библиотек. В настоящее время созданы драйверы для четырех серверов баз данных. Это:

- DB2;
- InterBase;
- MySQL;
- Oracle.

Технология dbExpress реализована на основе использования стандартных типов компонентов доступа к данным, проста при распространении (исполняемый файл приложения или одна-две динамические библиотеки). Поддерживает кроссплатформенную разработку для Linux и легко интегрируется в приложения CLX.

К недостаткам технологии нужно отнести использование однонаправленных курсоров и ограниченные возможности по редактированию (редактирование возможно только при кэшировании изменений на клиенте или выполнением специальных модифицирующих запросов).

ГЛАВА 18

Сервер баз данных InterBase и компоненты InterBase Express

На странице **InterBase** Палитры компонентов содержатся компоненты доступа к данным, адаптированные для работы с сервером InterBase и объединенные названием *InterBase Express*. Компоненты из набора InterBase Express предназначены для работы с сервером InterBase версии не ниже 5.5.

Их преимущество заключается в реализации всех функций за счет прямого обращения к API сервера InterBase. Благодаря этому существенно повысилась скорость работы компонентов.

Новые компоненты предоставляют разработчику новые возможности. Среди них:

- улучшенное управление транзакциями (для этого теперь предназначен отдельный компонент `TIBTransaction`);
- новые компоненты доступа к данным, позволяющие лучше решать распространенные задачи программирования (компоненты `TIBDataSet`, `TIBSQL`);
- возможность получения сведений о состоянии базы данных без прямого обращения к ее системным таблицам (компонент `TIBDatabaseInfo`);
- отслеживание состояния процессов выполнения запросов (компонент `TIBSQLMonitor`).

С точки зрения разработчика, за исключением нескольких новых свойств, методика использования этих компонентов в приложениях БД не отличается от стандартной методики *(см. часть III)*. Любой новый компонент, инкапсулирующий набор данных, совершенно обычным образом через компонент `TDataSource` можно подключить к любому стандартному компоненту отображения данных.

В этой главе рассматриваются следующие вопросы:

- соединение с сервером InterBase и полноценное управление транзакциями из клиентского приложения;

- что изменилось в стандартных компонентах доступа к данным;
- назначение и возможности новых компонентов доступа к данным;
- отслеживание процессов выполнения запросов на сервере из клиентского приложения;
- оценивание состояния базы данных;
- особенности переноса клиентских приложений, работающих с сервером InterBase, на новую компонентную базу.

Механизм доступа к данным InterBase Express

Для компонентов InterBase Express соединение с сервером БД осуществляет компонент `TIBDatabase`.

Для создания приложения клиент/сервер необходимо не только иметь работающий сервер, но и инсталлировать на клиентских рабочих местах специальное программное обеспечение, выполняющее соединение клиентского приложения с сервером.

Механизм доступа к данным InterBase Express использует для обращений к серверу возможности клиентского ПО InterBase, которое должно быть инсталлировано на компьютере. Если с данного компьютера доступны базы данных какого-либо сервера на платформе InterBase, то рассматриваемые здесь компоненты могут обращаться к этому серверу. При этом не требуется использовать BDE или любой другой механизм доступа к данным.

Но в результате все компоненты InterBase Express, инкапсулирующие набор данных, должны обращаться к базе данных только через компонент соединения `TIBDatabase`. На самом деле эта особенность не является недостатком в клиентских приложениях, т. к. организация соединения через один специализированный компонент всячески приветствуется и является хорошим тоном в программировании.

Компонент *TIBDatabase*

Так как для доступа к базе данных компонентам InterBase Express не требуется BDE, то для создания соединения используется всего одно свойство `DatabaseName`. В нем необходимо указать полный путь (включая имя сервера) к выбранному файлу БД с расширением gdb. Для этого можно воспользоваться стандартным диалогом выбора файла при щелчке на кнопке свойства в Инспекторе объектов.

Компонент имеет собственный редактор, который позволяет задать значения основных свойств, обеспечивающих соединение с базой данных (рис. 18.1).

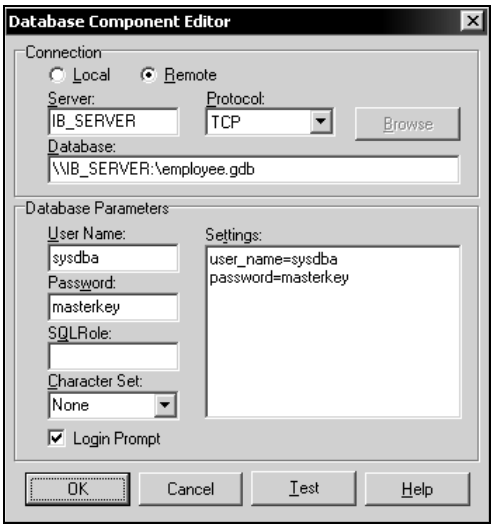


Рис. 18.1. Редактор компонента `TIBDatabase`

Настройка соединения проводится следующим образом.

На панели **Connection** выбирается требуемый сервер InterBase (локальный или доступный удаленно), затем в списке **Protocol** определяется используемый сетевой протокол и при помощи кнопки **Browse** выбирается файл базы данных.

На панели **Database Parameters** задаются имя пользователя, его пароль и роль. Также можно выбрать и набор шрифтов для языковой адаптации приложения (список **Character Set**).

Для задания вводимых при подключении параметров (имя пользователя, пароль, схема, роль и т. д.) также можно использовать свойства `Params` и `LoginPrompt`.

Путь к файлу базы данных задается свойством

```
property DatabaseName: String;
```

Соединение включается и отключается свойством

```
property Connected : Boolean;
```

При этом свойство

```
property AllowStreamedConnected : Boolean;
```

управляет включением соединения при запуске приложения и служит дополнительным предохранителем. При значении `False` свойство запрещает открытие соединения при запуске приложения, даже если свойство `Connected` имело значение `True`. Так как часто приложение отлаживается на тестовой базе данных, а используется на реальной, то неверный путь

в свойстве `DatabaseName` и не отключенное на этапе разработки свойство `Connected` приведет к возникновению ошибки открытия соединения при запуске приложения на другом компьютере.

Параметры соединения, которые нельзя задать свойствами, устанавливаются свойством

```
property Params: TStrings;
```

в котором в каждой строке задается имя параметра и затем через знак равенства — его значение. Наиболее распространенный пример использования свойства `Params` — задание имени пользователя и его пароля:

```
user_name=sysdba
password=masterkey
```

Свойство

```
property DBParamByDPB: [const Idx: Integer]: String;
```

позволяет получить доступ к отдельным параметрам соединения, не обращаясь к свойству `Params`.

Примечание

Полный список индексов всех возможных параметров соединения Interbase можно найти в файле \Delphi7\Source\Vcl\IBHeader.pas.

Если соединение настроено правильно, метод

```
procedure TestConnected: Boolean;
```

возвращает значение `True`, иначе — `False`.

Свойство

```
property IdleTimer: Integer;
```

задает временной интервал до отключения неиспользуемого соединения.

В компоненте `TIBDatabase` отсутствуют средства управления транзакциями, которые вынесены в отдельный компонент `TIBTransaction` *(см. ниже)*.

Свойство

```
property DefaultTransaction: TIBTransaction;
```

позволяет задать транзакцию по умолчанию. При этом все компоненты с наборами данных, использующие данное соединение автоматически, начинают применять этот компонент транзакции. Изменяя значение этого свойства, можно в одном соединении работать с несколькими транзакциями.

Общее число связанных с данным соединением транзакций возвращает свойство

```
property TransactionCount: Integer;
```

а их полный перечень содержится в индексированном списке свойства

```
property Transactions [Index: Integer]: TIBTransaction;
```

Добавить к списку используемых новую транзакцию можно при помощи метода

```
function AddTransaction(TR: TIBTransaction): Integer;
```

Отменить связь между соединением и компонентом транзакции позволяет метод

```
procedure RemoveTransaction(Idx: Integer);
```

Но можно поступить и более радикально. Метод

```
procedure RemoveTransactions;
```

отменяет связи со всеми транзакциями.

Используемый в методе `RemoveTransaction` индекс транзакции может быть найден методом

```
function FindTransaction (TR: TIBTransaction): Integer;
```

а метод

```
function FindDefaultTransaction: TIBTransaction;
```

возвращает транзакцию по умолчанию.

С компонентом соединения можно связать произвольное число объектов, отслеживающих возникновение событий в базе данных InterBase *(см. ниже)*. Для этого используется метод

```
procedure AddEventNotifier(Notifier: IIBEventNotifier);
```

который связывает с соединением либо интерфейс `IIBEventNotifier`, либо объект `TIBEvents`.

Парный ему метод

```
procedure RemoveEventNotifier(Notifier: IIBEventNotifier);
```

разрывает связь соединения с объектом-обработчиком событий.

Свойство

```
type
TTraceFlag = (tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
tfConnect, tfTransact, tfBlob, tfService, tfMisc);
TTraceFlags = set of TTraceFlag;
property TraceFlags: TTraceFlags;
```

позволяет управлять сведениями о выполнении запросов, возвращаемыми компонентом `TSQLMonitor` *(см. ниже описание этого компонента)*.

Группа методов позволяет судить о реальном состоянии соединения во время выполнения. Все они в случае неудачи проверки генерируют исключение `EIBClientError`.

Методы

```
procedure CheckActive;
```

и

```
procedure CheckInactive;
```

проверяют, функционирует или нет соединение.

Метод

```
procedure CheckDatabaseName;
```

проверяет, заполнено ли свойство `DatabaseName`.

Компонент `TIBDatabase` позволяет выполнять некоторые операции с метаданными базы данных.

При помощи метода

```
procedure CreateDatabase;
```

можно создавать новые базы данных, включая создание файла базы данных. Все параметры новой базы данных, которые разработчик посчитает нужным указать явно, должны быть включены в список свойства `Params` *(см. выше)*.

Имя файла новой базы данных должно быть указано в свойстве `DatabaseName`.

Метод

```
procedure DropDatabase;
```

удаляет существующую базу данных, путь к которой указан свойством `DatabaseName`.

Список `List` имен таблиц, имеющихся в базе данных, возвращает метод

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
```

При этом параметр `SystemTables` управляет включением в список имен системных таблиц.

Метод

```
procedure GetFieldNames(const TableName: string; List: TStrings);
```

аналогичным образом возвращает список полей для таблицы, заданной параметром `TableName`.

Методы-обработчики событий компонента `TIBDatabase` представлены в табл. 18.1.

Таблица 18.1. *Методы-обработчики событий компонента* `TIBDatabase`

Объявление	Тип	Описание
`property AfterConnect: TNotifyEvent;`	Pb	Выполняется после открытия соединения
`property AfterDisconnect: TNotifyEvent;`	Pb	Выполняется после закрытия соединения
`property BeforeConnect: TNotifyEvent;`	Pb	Выполняется перед открытием соединения
`property BeforeDisconnect: TNotifyEvent;`	Pb	Выполняется перед закрытием соединения
`property OnDialectDowngradeWarning: TNotifyEvent;`	Pb	Выполняется в случае изменения диалекта SQL при открытии соединения
`property OnIdleTimer: TNotifyEvent;`	Pb	Вызывается по истечении времени, заданного свойством `IdleTimer`
`TDatabaseLoginEvent = procedure(Database: TIBDatabase; LoginParams: TStrings) of object;` `property OnLogin: TDatabaseLoginEvent;`	Pb	Вызывается для регистрации пользователя при открытии соединения

Компонент *TIBTransaction*

Компонент `TIBTransaction` инкапсулирует средства управления транзакцией при работе с сервером InterBase. Для этого он должен быть связан с компонентом `TIBDatabase` при помощи своего свойства

```
property DefaultDatabase: TIBDatabase;
```

Один компонент транзакции может быть связан с несколькими компонентами `TIBDatabase`. Для этого необходимо задать один компонент транзакции в свойствах `DefaultTransaction` всех необходимых компонентов соединений *(см. выше)*.

Список всех связанных компонентов соединений содержится в свойстве

```
property Databases[Index: Integer]: TIBDatabase;
```

а их общее число возвращает свойство

```
property DatabaseCount: Integer;
```

Во время выполнения новое соединение может быть связано с транзакцией методом

```
function AddDatabase(db: TIBDatabase): Integer;
```

Или же, связь может быть отменена:

```
procedure RemoveDatabase(Idx: Integer);
```

А метод

```
procedure RemoveDatabases;
```

разрывает все установленные связи с компонентом `TIBDatabase`.

Индекс связанного соединения в списке **Databases** транзакции можно получить при помощи метода

```
function FindDatabase (db: TIBDatabase): Integer;
```

Например, если вам не известно ничего, кроме имени компонента, можно поступить так:

```
var i, FIndex: Integer;
...
for i := 0 to Form1.ComponentCount – 1 do
  if Form1.Components[i].Name = 'IBDatabase1'
   then FIndex :=
IBTransaction1.FindDatabase(TIBDatabase(Form1.Components[i]));
...
```

Соединение, заданное по умолчанию свойством `DefaultDatabase`, возвращает метод

```
function FindDefaultDatabase: TIBDatabase;
```

Транзакция может иметь набор параметров, задать которые можно при помощи свойства

```
property Params: TStrings;
```

аналогично компоненту `TIBDatabase`. Прямой доступ для чтения к буферу параметров транзакции Transaction Parameters Buffer (TPB) типа `PChar` обеспечивает свойство

```
property TPB: PChar;
```

Длина буфера содержится в свойстве

```
property TPBLength: Short;
```

Дескриптор транзакции представлен свойством

```
property Handle: TISC_TR_HANDLE;
```

После того как транзакция настроена, ее можно начать, сохранить или отменить.

Транзакция стартует при помощи метода

```
procedure StartTransaction;
```

При необходимости сохранить все сделанные в рамках текущей транзакции изменения используется метод

```
procedure Commit;
```

Если выполненные действия нужно отменить, применяется метод

```
procedure Rollback;
```

Для открытия и сохранения транзакции можно использовать традиционное свойство

```
property Active: Boolean;
```

После начала новой транзакции свойство

```
property InTransaction: Boolean;
```

принимает значение `True`, а после фиксации или отката — значение `False`.

При работе с сервером InterBase 6.0 можно использовать методы `CommitRetaining` и `RollbackRetaining`. В отличие от стандартных операций фиксации и отката транзакций, эти методы после передачи или отмены изменений оставляют текущую транзакцию открытой.

Если сервер перегружен и не откликается на транзакцию, то по истечении времени, заданного свойством

```
property IdleTimer: Integer;
```

выполняется действие, заданное свойством

```
type TTransactionAction = (taRollback, taCommit, taRollbackRetaining,
taCommitRetaining);
property DefaultAction: TTransactionAction;
```

`taRollback` — откат транзакции;

`taCommit` — фиксация транзакции;

`taRollbackRetaining` — отмена изменений без завершения транзакции (для сервера InterBase 6.0);

`taCommitRetaining` — фиксация изменений без завершения транзакции (для сервера InterBase 6.0).

Для компонента транзакции можно настроить ее автоматическое завершение при закрытии последнего открытого компонента, инкапсулирующего набор данных, связанного с тем же соединением, что и транзакция.

Для этого свойство

```
type TAutoStopAction = (saNone, saRollback, saCommit,
saRollbackRetaining, saCommitRetaining);
property AutoStopAction : TAutoStopAction;
```

не должно иметь значение `saNone`.

Остальные значения свойства выполняют следующие действия:

- `saRollback` — откат транзакции;
- `saCommit` — фиксация транзакции;
- `saRollbackRetaining` — отмена изменений без завершения транзакции (для сервера InterBase 6.0);
- `saCommitRetaining` — фиксация изменений без завершения транзакции (для сервера InterBase 6.0).

Метод

```
procedure CheckAutoStop;
```

выполняет действие, предусмотренное текущим значением свойства `AutoStopAction`.

Диагностика состояния транзакции во время выполнения осуществляется группой специальных методов. В случае отрицательного результата все они генерируют исключение `EIBClientError`.

Метод

```
procedure CheckDatabasesInList;
```

проверяет, имеются ли в списке **Databases** связанные соединения.

Метод

```
procedure CheckInTransaction;
```

проверяет, открыта ли в данный момент транзакция.

Метод

```
procedure CheckNotInTransaction;
```

проверяет, закрыта ли в данный момент транзакция.

Единственный метод-обработчик транзакции

```
property OnIdleTimer: TNotifyEvent;
```

вызывается по истечении срока ожидания выполнения транзакции, заданного свойством `IdleTimer`.

Компоненты доступа к данным

Так как компоненты InterBase Express используют для получения набора данных собственный механизм, то иерархия классов-предков включает только обязательный для всех наборов данных `TDataSet` класс `TIBCustomDataSet`, который, собственно, и инкапсулирует механизм доступа InterBase Express (см. рис. 12.1).

Для связи с базой данных компоненты InterBase Express применяют компоненты соединения `TIBDatabase` *(см. выше)*. Для этого они используют свойство

```
property Database: TIBDatabase;
```

Доступ к связанной транзакции осуществляется через свойство

```
property Transaction: TIBTransaction;
```

Дополнительно к стандартным свойствам и методам, описываемым в *гл. 12*, класс `TIBCustomDataSet` имеет свойство

```
type TIBUpdateRecordTypes = set of (cusModified, cusInserted, cusDeleted,
cusUnmodified, cusUninserted);
property UpdateRecordTypes: TIBUpdateRecordTypes;
```

`cusModified` — модифицированные записи;

`cusInserted` — добавленные записи;

`cusDeleted` — удаленные записи;

`cusUnmodified` — немодифицированные записи;

`cusUninserted` — недобавленные записи.

Данное свойство определяет записи набора данных, на которые распространяются операции кэширования.

Свойство

```
property BufferChunks: Integer;
```

определяет число записей, которые компонент загружает в собственный локальный буфер для ускорения выполнения стандартных операций.

При использовании компонентов в приложениях необходимо учитывать некоторые особенности.

Обновление набора данных выполняется не при каждом сохранении изменений. Такое поведение компонента определяется свойством

```
property ForcedRefresh: Boolean;
```

которое по умолчанию имеет значение `False`.

Это ускоряет работу компонента. При необходимости выполнять обновление данных с максимальной частотой свойству `ForcedRefresh` нужно присвоить значение `True`.

В зависимости от настроек компонента, с ним можно выполнять различные виды операций редактирования, перечень которых содержится в свойстве "только для чтения":

```
type
  TLiveMode = (lmInsert, lmModify, lmDelete, lmRefresh);
  TLiveModes = set of TLiveMode;
property LiveMode: TLiveModes;
```

Так как все эти компоненты предназначены для работы с сервером, то изначально все они поддерживают режим кэширования изменений и имеют соответственные свойства, методы и методы-обработчики событий (табл. 18.2).

Таблица 18.2. *Методы-обработчики событий класса* `TIBCustomDataSet`

Объявление	Описание
`property AfterDatabaseDisconnect: TNotifyEvent;`	Выполняется после закрытия соединения с базой данных
`property AfterTransactionEnd: TNotifyEvent;`	Выполняется по окончании транзакции, с которой связан данный набор данных
`property BeforeDatabaseDisconnect: TNotifyEvent;`	Выполняется перед закрытием соединения с базой данных
`property BeforeTransactionEnd: TNotifyEvent;`	Выполняется перед окончанием транзакции, с которой связан данный набор данных
`property DatabaseFree: TNotifyEvent;`	Выполняется при обнулении свойства `Database` компонента набора данных

`type`
`TIBUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied, uaApply);`
`TIBUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction)of object;`
`property OnUpdateError: TIBUpdateErrorEvent;`
	Вызывается при возникновении ошибки сохранения изменений в режиме кэширования

Таблица 18.2 *(окончание)*

Объявление	Описание

`type`
`TIBUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApply, uaApplied);`
`TIBUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction) of object;`
`property OnUpdateRecord: TIBUpdateRecordEvent;`
	Вызывается при сохранении изменений в режиме кэширования

`property TransactionFree: TNotifyEvent;`	Выполняется при обнулении свойства `Transaction` компонента набора данных

Возможности компонентов `TIBTable`, `TIBQuery`, `TIBStoredProc`, `TIBUpdateSQL` мало чем отличаются от стандартных, описанных в *гл. 12*.

Для взаимодействия с сервером компоненты InterBase Express используют два класса, которые инкапсулируют важные структуры API InterBase. Эти структуры обеспечивают передачу серверу параметров запроса и возвращение результата выполнения запроса. Поэтому сначала рассмотрим классы `TIBXSQLDA` и `TIBXSQLVAR`, а затем перейдем к компонентам.

Область дескрипторов *XSQLDA*

Запрос может иметь собственные параметры, которые должны содержаться в свойстве `Params`. Однако, в отличие от обычного компонента запроса, в InterBase Express это свойство представляет собой экземпляр класса `TIBXSQLDA` (табл. 18.3). Этот класс инкапсулирует одноименную структуру API InterBase — `XSQLDA`, обеспечивающую передачу параметров запросу и возврат результатов. Такая структура имеется у каждого запроса, который выполняется сервером InterBase и называется *областью дескрипторов запроса* (descriptors area).

Таблица 18.3. *Свойства и методы класса* `TIBXSQLDA`

Объявление	Тип	Описание
Свойства		
`property AsXSQLDA: PXSQLDA;`	Pu	Ссылка на структуру `XSQLDA`
`property Count: Integer;`	Pu	Возвращает число полей в структуре

Таблица 18.3 *(окончание)*

Объявление	Тип	Описание
`property Modified: Boolean;`	Pu	Позволяет определить возможность редактирования полей структуры
`property Names: String;`	Pu	Возвращает имена полей в структуре
`property RecordSize: Integer;`	Pu	Возвращает размер записи структуры
`property Vars: [Idx: Integer]: TIBXSQLVAR;`	Pu	Индексированный список структур `XSQLVAR` *(см. ниже)*
Методы		
`procedure AddName(FieldName: String; Idx: Integer);`	Pu	Добавляет к структуре новое поле
`function ByName: [Idx: String]: TIBXSQLVAR;`	Pu	Возвращает структуру `XSQLVAR`, инкапсулирующую отдельное поле результата запроса *(см. ниже)*

Структура *XSQLVAR*

Рассмотренная выше область дескрипторов содержит возвращаемый результат запроса. Массив значений каждого возвращаемого поля сохраняется в отдельной структуре `XSQLVAR`. Индексированный список таких структур в области дескрипторов представлен свойством

```
property Vars: [Idx: Integer]: TIBXSQLVAR
```

В целом, рассматриваемая структура соответствует объекту поля Delphi *(см. гл. 13)*, о чем свидетельствует набор основных свойств и методов класса структуры, представленный в табл. 18.4.

Помимо представленных в таблице свойств, класс `TIBXSQLVAR` имеет ряд свойств, возвращающих значение в определенном формате: `AsCurrency`, `AsDate`, `AsDateTime`, `AsDouble`, `AsFloat`, `AsInt64`, `AsInteger`, `AsLong`, `AsPointer`, `AsQuad`, `AsShort`, `AsString`, `AsTime`, `AsVariant`.

Таблица 18.4. *Свойства и методы класса* `TIBXSQLVAR`

Объявление	Тип	Описание
Свойства		
`property AsXSQLVAR: PXSQLVAR;`	Pu	Представляет значение поля как структуру `XSQLVAR`
`property Data: PXSQLVAR;`	Pu	Ссылка на структуру `XSQLVAR`

Таблица 18.4 *(окончание)*

Объявление	Тип	Описание
`property Index: Integer;`	Pu	Возвращает индекс структуры в области дескрипторов
`property IsNull: Boolean;`	Pu	Позволяет определить наличие данных в структуре
`property IsNullable: Boolean;`	Pu	Позволяет определить, может ли структура иметь значение
`property Modified: Boolean;`	Pu	Позволяет определить, изменялось ли значение в структуре
`property Size: Integer;`	Pu	Максимальный размер данных в байтах
`property SQLType: Integer;`	Pu	Возвращает индекс API параметра
`property Value: Variant;`	Pu	Содержит возвращаемое значение
Методы		
`procedure Assign(Source: TIBXSQLVAR);`	Pu	Присваивает объект, передаваемый в параметре, данному объекту
`procedure LoadFromFile(const FileName: String);`	Pu	Загружает из файла данные в поле BLOB
`procedure LoadFromStream(Stream: TStream);`	Pu	Загружает из потока данные в поле BLOB
`procedure SaveToFile(const FileName: String);`	Pu	Сохраняет в файле данные из поля BLOB
`procedure SaveToStream(Stream: TStream);`	Pu	Сохраняет в потоке данные из поля BLOB

Компонент *TIBTable*

Компонент `TIBTable` реализует все возможности стандартного компонента, инкапсулирующего таблицу *(см. гл. 12)*. Дополнительно к ним можно обратить внимание на несколько полезных свойств и методов.

При выборе таблицы (свойство `TableName`) свойство

```
type
  TIBTableType = (ttSystem, ttView);
  TIBTableTypes = set of TIBTableType;
property TableTypes: TIBTableTypes;
```

определяет, какие таблицы доступны для выбора:

- `ttSystem` — доступны системные таблицы и просмотры;
- `ttView` — доступны определенные пользователем просмотры.

При открытии набора данных упорядочивание записей осуществляется в соответствии со значением свойства

```
property DefaultIndex: Boolean;
```

При значении `True` записи располагаются в порядке, определяемом первичным индексом таблицы БД.

Во время выполнения свойство

```
property Exists: Boolean;
```

позволяет определить, существует ли в базе данных таблица, имя которой определено свойством `TableName`.

Метод

```
procedure GotoCurrent(Table: TIBTable);
```

синхронизирует курсоры текущего набора данных и набора данных компонента, заданного параметром `Table`.

Методы-обработчики событий полностью соответствуют классу `TIBCustomDataSet` (см. табл. 18.2).

Компонент *TIBQuery*

Компонент `TIBQuery` выполняет все стандартные функции компонента запроса и наследует возможности класса `TIBCustomDataSet`.

Как и у остальных компонентов запросов, свойство

```
property SQL: TStrings;
```

содержит текст запроса и позволяет редактировать его. С этим свойством связан специализированный редактор (рис. 18.2).

Для просмотра текста запроса можно использовать свойство

```
property Text: string;
```

Параметры запроса хранятся в стандартном свойстве

```
property Params: TParams;
```

Общее число параметров запроса возвращает свойство

```
property ParamCount: Word;
```

При создании новых записей в редактируемых наборах данных компонентов запросов возникает проблема присвоения значений полям первичных ин-

дексов. Очевидно, что при сохранении новой записи в базе данных поле первичного индекса будет инкрементировано средствами сервера InterBase (соответствующими генератором и триггером). Однако получить это значение в приложении можно только сохранив изменения и обновив набор данных, что зачастую требует больших затрат ресурсов.

Для решения этой проблемы в компоненте `TIBQuery` используется свойство

```
property GeneratorField: TIBGeneratorField;
```

Редактор свойства (рис. 18.2) позволяет связать генератор с инкрементируемым полем.

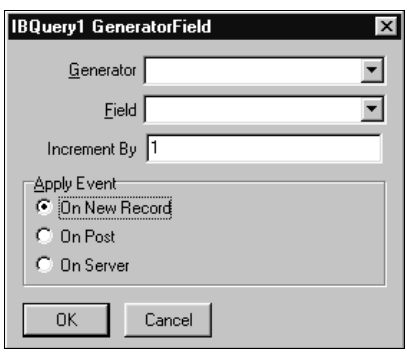


Рис. 18.2. Редактор свойства `GeneratorField` компонента `TIBQuery`

Список **Generator** позволяет выбрать один из доступных генераторов базы данных. Список **Field** задает инкрементируемое поле набора данных. В строке **Increment By** определяется шаг прибавляемого значения поля.

Группа радиокнопок **Apply Event** определяет событие, при котором срабатывает генератор:

- **On New Record** — при создании новой записи;
- **On Post** — при сохранении новой записи;
- **On Server** — генератор управляется сервером.

Редактор свойства `GeneratorField` попросту присваивает значения полям экземпляра класса `TIBGeneratorField`.

Методы-обработчики событий полностью соответствуют классу `TIBCustomDataSet` (см. табл. 18.2).

Компонент *TIBStoredProc*

Компонент `TIBStoredProc` полностью соответствует стандартному прототипу, описываемому в *гл. 12*.

Имя хранимой процедуры задается свойством

```
property StoredProcName: String;
```

Список всех доступных на этапе выполнения хранимых процедур возвращает свойство

```
property StoredProcedureNames: TStrings;
```

Параметры хранимой процедуры содержатся в стандартном свойстве

```
property Params: TParams;
```

Общее число параметров возвращает свойство

```
property ParamCount: Word;
```

Свойство

```
property Prepared: Boolean;
```

позволяет определить, подготовлена ли хранимая процедура к выполнению.

Методы-обработчики событий полностью соответствуют классу `TIBCustomDataSet` (см. табл. 18.2).

Компонент *TIBDataSet*

Компонент `TIBDataSet` предназначен для представления в приложениях наборов данных от сложных запросов (свойства и методы описаны в табл. 18.5). При этом набор данных остается редактируемым. Это достигается возможностью задать дополнительные запросы на удаление, изменение и добавление данных. Аналогичным образом работает стандартный компонент `TUpdateSQL` *(см. гл. 22)*. Однако в компоненте `TIBDataSet` интегрированы одновременно и сам основной запрос, и вспомогательные запросы.

Основной запрос содержится в свойстве

```
property SelectSQL: TStrings;
```

Создание запроса облегчает простой редактор, вызываемый при щелчке на кнопке в поле редактирования свойства в Инспекторе объектов (рис. 18.3).

Каждому запросу (основному и вспомогательным) соответствует собственный объект `TIBSQL`, который подробно рассматривается ниже.

Таблица 18.5. *Свойства и методы компонента* `TIBDataSet`

Объявление	Тип	Описание
Свойства		
`property BufferChunks: Integer;`	Pb	Определяет число записей в буфере набора данных

Таблица 18.5 *(продолжение)*

Объявление	Тип	Описание
`property DeleteSQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего удаление записей из набора данных
`property InsertSQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего добавление записей в набор данных
`property ModifySQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего изменение записей из набора данных
`property Params: TIBXSQLDA;`	Ro	Структура API, содержащая параметры запроса
`property Prepared: Boolean;`	Ro	Позволяет определить, подготовлен ли запрос к выполнению
`property QDelete: TIBSQL;`	Ro	Объект запроса на удаление
`property QInsert: TIBSQL;`	Ro	Объект запроса на добавление
`property QModify: TIBSQL;`	Ro	Объект запроса на изменение
`property QRefresh: TIBSQL;`	Ro	Объект запроса на обновление
`property QSelect: TIBSQL;`	Ro	Объект запроса на отбор данных
`property RefreshSQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего обновление записей набора данных
`property SelectSQL: TStrings;`	Pb	Содержит текст основного запроса набора данных
`type TIBSQLTypes = set of (SQLUnknown, SQLSelect, SQLInsert, SQLUpdate, SQLDelete, SQLDDL, SQLGetSegment, SQLPutSegment, SQLExecProcedure, SQLStartTransaction, SQLCommit, SQLRollback, SQLSelectForUpdate, SQLSetGenerator);`	Ro	Возвращает тип основного запроса набора данных: • `SQLUnknown` — неизвестный тип; • `SQLSelect`, `SQLInsert`, `SQLUpdate`, `SQLDelete` — стандартные типы; • `SQLDDL` — выражение DDL; • `SQLGetSegment`, `SQLPutSegment` — запросы с полями BLOB; • `SQLExecProcedure`, `SQLStartTransaction`, `SQLCommit`, `SQLRollback` — обработка транзакций; • `SQLSelectForUpdate` — хранимая процедура, возвращающая набор данных;

Таблица 18.5 *(окончание)*

Объявление	Тип	Описание
(прод.)		• `SQLSetGenerator` — выполнение генератора
Методы		
`procedure Prepare;`	Pu	Осуществляет подготовку всех запросов компонента к выполнению
`procedure UnPrepare;`	Pu	Возвращает все запросы набора данных к исходному состоянию
Методы-обработчики событий		
`property DatabaseDisconnected: TNotifyEvent;`	Pb	Вызывается после отключения базы данных
`property DatabaseDisconnecting: TNotifyEvent;`	Pb	Вызывается во время отключения базы данных
`property DatabaseFree: TNotifyEvent;`	Pb	Вызывается после того, как компонент соединения освобождает занимаемую память

Рис. 18.3. Редактор запроса компонента `TIBDataSet`

Компонент *TIBSQL*

Компонент `TIBSQL` предназначен для быстрого выполнения запросов SQL, поэтому не обеспечивает связи с компонентами представления данных (свойства и методы описаны в табл. 18.6).

Для обеспечения скорости выполнения запроса из компонента удалены все дополнительные механизмы, обслуживающие набор данных. Фактически компонент TIBSQL не имеет отношения к обычным компонентам доступа к данным, его непосредственным предком является класс TComponent, а не TDataSet. Поэтому он только передает через компонент соединения TIBDatabase запрос серверу и получает назад результат выполнения запроса.

Для повышения скорости компонент не обеспечивает полноценной навигации по набору данных. Перемещение по набору данных возможно только в прямом направлении (однонаправленный курсор).

Возвращаемые набором данных текущие значения полей содержатся не в привычном наборе объектов полей TField, а в объекте TIBXSQLDA *(см. выше)*. Так как структура XSQLDA создается сервером при выполнении запроса, существенно уменьшается время открытия набора данных компонента.

Таблица 18.6. *Свойства и методы компонента TIBSQL*

Объявление	Тип	Описание
Свойства		
property Bof: Boolean;	Pu	Значение True говорит о том, что курсор находится в начале набора данных
property Database: TIBDatabase;	Pb	Определяет компонент соединения с базой данных
property DBHandle: PISC_DB_HANDLE;	Pu	Указатель API на объект базы данных
property Eof: Boolean;	Pu	Значение True говорит о том, что курсор находится в конце набора данных
property FieldIndex: [FieldName: String]: Integer;	Pu	Список порядковых номеров полей по их именам
property Fields[const Idx: Integer]: TIBXSQLVAR;	Pu	Индексированный список структур XSQLVAR, хранящих значения полей набора данных
property GenerateParamNames: Boolean;	Pu	Установка свойства в значение True приводит к созданию списка имен параметров запроса в свойстве Params
property GoToFirstRecordOnExecute: Boolean;	Pb	Значение True обеспечивает установку курсора на первую запись набора данных при его открытии
property Handle: TISC_STMT_HANDLE;	Pu	Содержит указатель API на запрос

Таблица 18.6 *(продолжение)*

Объявление	Тип	Описание
`property Open: Boolean;`	Pu	Позволяет определить, открыт ли набор данных
`property ParamCheck: Boolean;`	Pb	Позволяет определить, был ли заново сгенерирован список параметров запроса при изменении его текста во время выполнения
`property Params: TIBXSQLDA;`	Pu	Область дескрипторов запроса *(см. выше)*
`property Plan: String;`	Pu	Содержит план запроса после его подготовки
`property Prepared: Boolean;`	Pu	Значение `True` сообщает о том, что запрос готов к выполнению
`property RecordCount: Integer;`	Pu	Возвращает число записей набора данных
`property RowsAffected: Integer;`	Pu	Возвращает число записей, обработанных запросом
`property SQL: TStrings;`	Pb	Содержит текст запроса
`property SQLType: TIBSQLTypes read FSQLType;`	Pu	Возвращает тип запроса (см. табл. 24.5)
`property Transaction: TIBTransaction;`	Pb	Указывает на компонент транзакции
`property TRHandle: PISC_TR_HANDLE;`	Pu	Содержит указатель API на транзакцию, в которой работает запрос
`property UniqueRelationName: String;`	Pu	Возвращает уникальное внутреннее имя запроса
Методы		
`procedure BatchInput(InputObject: TIBBatchInput);`	Pu	Выполняет запрос с параметрами для переноса в объект `InputObject`
`procedure BatchOutput(OutputObject: TIBBatchOutput);`	Pu	Выполняет запрос с параметрами для переноса в объект `OutputObject`
`function Call(ErrCode: ISC_STATUS; RaiseError: Boolean): ISC_STATUS;`	Pu	Возвращает текст сообщения об ошибке по ее коду `ErrCode`
`procedure CheckClosed;`	Pu	Вызывает исключение, если набор данных открыт

Таблица 18.6 *(окончание)*

Объявление	Тип	Описание
procedure CheckOpen;	Pu	Вызывает исключение, если набор данных закрыт
procedure CheckValidStatement;	Pu	Вызывает исключение, если запрос некорректен
procedure Close;	Pu	Закрывает набор данных
function Current: TIBXSQLDA;	Pu	Ссылка на область дескрипторов запроса
procedure ExecQuery;	Pu	Выполняет запрос
function FieldByName[FieldName: String]: TIBXSQLVAR;	Pu	Возвращает структуру XSQLVAR по имени поля
procedure FreeHandle;	Pu	Освобождает ресурсы, занятые запросом
function Next: TIBXSQLDA;	Pu	Возвращает область дескрипторов для следующей записи
procedure Prepare;	Pu	Готовит запрос к выполнению
Методы-обработчики событий		
property OnSQLChanging: TNotifyEvent;	Pb	Вызывается при изменении запроса

Текст запроса задается обычным для всех компонентов запросов свойством SQL. Для выполнения запроса используется также знакомое свойство ExecSQL. После этого можно обращаться к созданному компонентом набору данных.

Значения полей из текущей записи доступны через свойство Fields. Обратите внимание, что это не объекты типа TFields, а структуры XSQLVAR из области дескрипторов.

Будут ли переданы значения полей в компонент, зависит от значения свойства GoToFirstRecordOnExecute.

Доступ к области дескрипторов осуществляется через свойство Current.

Переход к следующей записи выполняется методом Next. При этом обновляется область дескрипторов запроса.

Обработка событий

Клиентское приложение Delphi, работающее с сервером InterBase, имеет возможность отслеживать события, происходящие в базе данных и вызываемые другими процессами или приложениями. Для этого используется

компонент TIBEvents. Он позволяет определить список необходимых событий и предоставляет разработчику простой механизм отслеживания возникающих на сервере событий. Свойства и методы компонента TIBEvents представлены в табл. 18.7.

Список событий задается свойством

```
property Events: TStrings;
```

в котором можно определить до 15 контролируемых событий.

Выбранные события необходимо зарегистрировать на сервере. Для этого применяется метод

```
procedure RegisterEvents;
```

Метод

```
procedure QueueEvents;
```

начинает процесс передачи сообщений от сервера.

При возникновении на сервере зарегистрированного события компонент вызывает метод-обработчик события

```
property OnEventAlert: TEventAlert;
TEventAlert = procedure( Sender: TObject; EventName: String; EventCount:
longint; var CancelAlerts: Boolean)
```

Параметр EventName содержит имя последнего произошедшего события.

Параметр EventCount содержит число заданных событий, произошедших с момента последнего вызова метода-обработчика.

Параметр CancelAlerts позволяет прервать процесс передачи сообщений приложению. Для этого необходимо присвоить параметру значение True.

Для возобновления работы компонента нужно снова использовать метод QueueEvents.

Таблица 18.7. *Свойства и методы компонента TIBEvents*

Объявление	Тип	Описание
Свойства		
property Database: TIBDatabase;	Pb	Задает базу данных
property Events: TStrings;	Pb	Список контролируемых событий
property Queued: Boolean;	Ro	Значение True говорит о том, что процесс передачи сообщений работает
property Registered: Boolean;	Pb	Определяет регистрацию сообщений на сервере

Таблица 18.7 *(окончание)*

Объявление	Тип	Описание
Методы		
`procedure CancelEvents;`	Pu	Останавливает процесс передачи сообщений
`procedure QueueEvents;`	Pu	Включает процесс передачи сообщений
`procedure RegisterEvents;`	Pu	Проводит регистрацию сообщений на сервере
`procedure UnRegisterEvents;`	Pu	Отменяет регистрацию сообщений на сервере
Методы-обработчики событий		
`property OnEventAlert: TEventAlert;` `TEventAlert = procedure(Sender: TObject; EventName: String; EventCount: longint; var CancelAlerts: Boolean)`	Pb	Вызывается при передаче сообщения от сервера компоненту

Информация о состоянии базы данных

В процессе отладки и выполнения клиентских приложений для сервера InterBase разработчик может получать подробную информацию об этих процессах.

Компонент `TIBDatabaseInfo` предоставляет информацию о текущем состоянии базы данных.

Компонент `TIBSQLMonitor` отслеживает выполнение запросов на сервере.

Компонент *TIBDatabaseInfo*

Компонент `TIBDatabaseInfo` обладает большим числом свойств и методов, содержащих разнообразные сведения о состоянии БД (табл. 18.8). Компонент очень прост в применении.

Для выбора базы данных (компонента `TIBDatabase`) используется стандартное свойство

```
property Database: TIBDatabase;
```

В процессе работы с базой данных свойствам компонента `TIBDatabaseInfo` передаются соответствующие значения. Разработчику необходимо лишь в нужных местах использовать значения требуемых свойств.

Таблица 18.8. *Свойства и методы компонента* `TIBDatabaseInfo`

Объявление	Тип	Описание
Свойства		
`property Allocation: Long;`	Ro	Число выделенных страниц БД
`property BackoutCount: TStringList;`	Ro	Число вариантов удаленных записей
`property BaseLevel: Long;`	Ro	Версия базы данных (содержится во втором байте)
`property CurrentMemory: Long;`	Ro	Объем памяти (в байтах), занятый сервером
`property Database: TIBDatabase;`	Pb	Ссылка на компонент соединения с БД
`property DBFileName: String;`	Ro	Имя файла БД
`property DBImplementationClass: Long;`	Ro	Номер класа описания
`property DBImplementationNo: Long;`	Ro	Номер описания
`property DBSiteName: String;`	Ro	Имя сайта БД
`property DBSQLDialect: Long;`	Ro	Номер диалекта SQL
`property DeleteCount: TStringList;`	Ro	Число удалений с момента последнего обновления БД
`property ExpungeCount: TStringList;`	Ro	Число удалений записей с момента последнего сохранения БД
`property Fetches: Long;`	Ro	Число чтений из кэша
`property ForcedWrites: Long;`	Ro	Режим чтения: 0 — асинхронное чтение; 1 — синхронное чтение.
`property InsertCount: TStringList;`	Ro	Число добавлений в БД с момента последнего сохранения
`property Marks: Long;`	Ro	Число выполненных записей в кэш
`property MaxMemory: Long;`	Ro	Максимальный размер памяти, занимаемый БД с момента последнего сохранения
`property NoReserve: Long;`	Ro	Резервирование страниц: 0 — резервирование есть; 1 — резервирования нет
`property NumBuffers: Long;`	Ro	Число выделенных буферов
`property ODSMajorVersion: Long;`	Ro	Верхнее значение ODS

Таблица 18.8 *(окончание)*

Объявление	Тип	Описание
`property ODSMinorVersion: Long;`	Ro	Нижнее значение ODS
`property PageSize: Long;`	Ro	Размер страницы БД
`property PurgeCount: TStringList;`	Ro	Общее число удаленных по любой причине записей
`property ReadIdxCount: TStringList;`	Ro	Число чтений через индексы с момента последнего сохранения
`property ReadOnly: Long;`	Ro	0 — БД только для чтения; 1 — перезаписываемая БД
`property Reads: Long;`	Ro	Число чтений из БД
`property ReadSeqCount: TStringList;`	Ro	Число чтений таблиц целиком с последнего сохранения
`property SweepInterval: Long;`	Ro	Число зафиксированных транзакций
`property UpdateCount: TStringList;`	Ro	Число обновлений БД с момента последнего сохранения
`property UserNames: TStringList;`	Ro	Список активных пользователей
`property Version: String;`	Ro	Версия БД
`property Writes: Long;`	Ro	Число постраничных записей
Методы		
`function Call(ErrCode: ISC_STATUS; RaiseError: Boolean): ISC_STATUS;`	Pu	Возвращает сообщение об ошибке по параметру `ErrCode`

Компонент *TIBSQLMonitor*

Компонент `TIBSQLMonitor` позволяет получать в клиентском приложении сообщения от сервера о выполняемых им операциях. Для этого используется метод-обработчик компонента

```
TSQLEvent = procedure(EventText: String) of object;
property OnSQL: TSQLEvent;
```

Параметр `EventText` содержит текст сообщения.

В компоненте соединения с БД можно установить перечень событий сервера, на которые будет реагировать компонент `TIBSQLMonitor`. Это делается

при помощи свойства `TraceFlags` *(см. выше)*. Вероятные значения множества означают контроль за следующими операциями:

- `tfQPrepare` — подготовка запроса к выполнению (вызов метода `Prepare`);
- `tfQExecute` — выполнение запроса (вызов метода `ExecSQL`);
- `tfQFetch` — вызов запроса (вызов методов `Open`, `Close`);
- `tfError` — возникновение ошибки;
- `tfStmt` — все операции с запросами;
- `tfConnect` — подключение и отключение БД;
- `tfTransact` — выполнение транзакций;
- `tfBlob` — операции с данными BLOB;
- `tfService` — вспомогательные операции;
- `tfMisc` — любые операции, не учтенные вышеперечисленными значениями.

Резюме

В этой главе рассмотрены возможности набора компонентов InterBase Express. Они обеспечивают быстрый и эффективный доступ к базам данных на серверах InterBase. Для доступа к данным этим компонентам не требуется BDE, они используют только возможности API InterBase.

Часть компонентов обеспечивает быстрый переход со стандартных компонентов, инкапсулирующих набор данных, и повторяет функциональность компонентов `TTable`, `TQuery`, `TStoredProc` и т. д.

Компоненты `TIBSQL` и `TIBDataSet` полностью основаны на механизмах API InterBase, работают еще эффективнее, но требуют нестандартных приемов работы.

ГЛАВА 19

Использование ADO средствами Delphi

Наряду с традиционными инструментами доступа к данным Borland Database Engine и ODBC в приложениях Delphi можно применять технологию Microsoft ActiveX Data Objects (ADO), которая основана на возможностях COM, а именно интерфейсов OLE DB.

Технология ADO завоевала популярность у разработчиков, благодаря универсальности — базовый набор интерфейсов OLE DB имеется в каждой современной операционной системе Microsoft. Поэтому для обеспечения доступа приложения к данным достаточно лишь правильно указать провайдер соединения ADO и затем переносить программу на любой компьютер, где имеется требуемая база данных и, конечно, установленная ADO.

В Палитре компонентов Delphi есть страница ADO, содержащая набор компонентов, позволяющих создавать полноценные приложения БД, обращающиеся к данным через ADO.

В этой главе рассматриваются следующие вопросы:

- краткий обзор технологии ADO, доступных провайдеров ADO, а также работающих в ней объектов и интерфейсов;
- как создать соединение с базой данных через ADO в приложении Delphi;
- применение объекта набора записей ADO в приложении;
- как использовать таблицы, запросы SQL и хранимые процедуры;
- что такое команды и объекты команды ADO.

Основы ADO

Технология Microsoft ActiveX Data Objects обеспечивает универсальный доступ к источникам данных из приложений БД. Такую возможность предоставляют функции набора интерфейсов, созданные на основе общей модели объектов COM и описанные в спецификации OLE DB.

Технология ADO и интерфейсы OLE DB обеспечивают для приложений единый способ доступа к источникам данных различных типов (рис. 19.1). Например, приложение, использующее ADO, может применять одинаково сложные операции и к данным, хранящимся на корпоративном сервере SQL, и к электронным таблицам, и локальным СУБД. Запрос SQL, направленный любому источнику данных через ADO, будет выполнен.

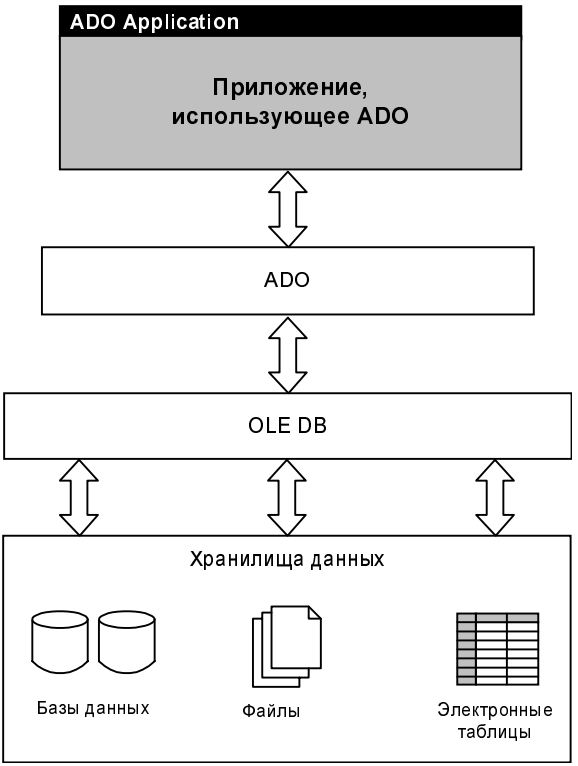


Рис. 19.1. Схема доступа к данным через ADO

Возникает вопрос: каким образом источники данных смогут выполнить этот запрос?

За серверы БД беспокоиться не стоит, обработка запросов SQL — это их основная обязанность. Но как быть с файловыми последовательностями, электронными таблицами, файлами электронной почты и т. д.? Здесь на помощь приходят механизмы ADO и интерфейсы OLE DB.

OLE DB представляет собой набор специализированных объектов COM, инкапсулирующих стандартные функции обработки данных, и специализированные функции конкретных источников данных и интерфейсов, обеспечивающих передачу данных между объектами.

Согласно терминологии ADO, любой источник данных (база данных, электронная таблица, файл) называется *хранилищем данных*, с которым при помощи провайдера данных взаимодействует приложение. Минимальный набор компонентов приложения может включать объект соединения, объект набора данных, объект процессора запросов.

Примечание

Объекты OLE DB создаются и функционируют так же, как и другие объекты COM. Каждому объекту соответствует идентификатор класса `CLSID`, хранящийся в системном реестре. Для создания объекта используется метод `CoCreateInstance` и соответствующая фабрика класса. Объекту соответствует набор интерфейсов, к методам которых можно обращаться после создания объекта.

В результате приложение обращается не прямо к источнику данных, а к объекту OLE DB, который "умеет" представить данные (например, из файла электронной почты) в виде таблицы БД или результата выполнения запроса SQL.

Технология ADO в целом включает в себя не только сами объекты OLE DB, но и механизмы, обеспечивающие взаимодействие объектов с данными и приложениями. На этом уровне важнейшую роль играют провайдеры ADO, координирующие работу приложений с хранилищами данных различных типов.

Такая архитектура позволяет сделать набор объектов и интерфейсов открытым и расширяемым. Набор объектов и соответствующий провайдер может быть создан для любого хранилища данных без внесения изменений в исходную структуру ADO. При этом существенно расширяется само понятие данных — ведь можно разработать набор объектов и интерфейсов и для нетрадиционных табличных данных. Например, это могут быть графические данные геоинформационных систем, древовидные структуры из системных реестров, данные CASE-инструментов и т. д.

Так как технология ADO основана на стандартных интерфейсах COM, которые являются системным механизмом Windows, это сокращает общий объем работающего программного кода и позволяет распространять приложения БД без вспомогательных программ и библиотек.

Примечание

Нижеследующее описание спецификации OLE DB представлено в соответствии с официальной терминологией Microsoft для данной предметной области.

Спецификация OLE DB различает следующие типы объектов, которые будут рассмотрены ниже.

- *Перечислитель* (Enumerator) выполняет поиск источников данных или других перечислителей. Используется для обеспечения функционирования провайдеров ADO.

- *Объект-источник данных* (Data Source Object) представляет хранилище данных.
- *Сессия* (Session) объединяет совокупность объектов, обращающихся к одному хранилищу данных.
- *Транзакция* (Trasaction) инкапсулирует механизм выполнения транзакции.
- *Команда* (Command) содержит текст команды и обеспечивает ее выполнение. Командой может быть запрос SQL, обращение к таблице БД и т. д.
- *Набор рядов* (Rowset) представляет собой совокупность строк данных, являющихся результатом выполнения команды ADO.
- *Объект-ошибка* (Error) содержит информацию об исключительной ситуации.

Рассмотрим функциональные возможности основных объектов и интерфейсов OLE DB.

Перечислители

Объекты-перечислители обеспечивают поиск любых объектов ADO, которые имеют доступ к источникам данных. При этом другие перечислители также видны в данном перечислителе.

Первичный поиск источников данных осуществляется в провайдере ADO. Перечислители могут отбирать только источники данных конкретных типов, поэтому провайдер обеспечивает доступ к конкретному типу хранилища данных.

В составе ADO имеется системный корневой перечислитель, который выполняет начальный поиск других перечислителей и источников данных. Его можно использовать, зная его идентификатор класса `CLSID_OLEDB_ENUMERATOR`.

Примечание

В Delphi GUID глобального перечислителя содержится в файле \Delphi7\Source\Vcl\OleDB.pas.

```
CLSID_OLEDB_ENUMERATOR: TGUID = '{C8B522D0-5CF3-11CE-ADE5-
00AA0044773D}';
```

Функции перечислителя содержатся в интерфейсе `ISourcesRowset`. Метод

```
function GetSourcesRowset(const punkOuter: IUnknown; const riid: TGUID;
cPropertySets: UINT; rgProperties: PDBPropSetArray; out ppSourcesRowset:
IUnknown): HResult; stdcall;
```

возвращает ссылку на объект набора рядов *(см. выше)*, содержащий сведения о найденных источниках данных или перечислителях.

Объекты соединения с источниками данных

Внутренний механизм ADO, обеспечивающий соединение с хранилищем данных, использует два типа объектов. Это объекты-источники данных и объекты-сессии.

Объект-источник данных обеспечивает представление информации о требуемом реальном источнике данных и подключение к нему.

Для ввода сведений о хранилище данных используется интерфейс `IDBProperties`. Для успешного подключения необходимо задать обязательные сведения. Вероятно, для любого хранилища данных будет актуальной информация об его имени, пользователе и пароле. Однако каждый тип хранилища имеет собственные уникальные настройки. Для получения списка всех обязательных параметров соединения с данным хранилищем можно воспользоваться методом

```
function GetPropertyInfo(cPropertyIDSets: UINT; rgPropertyIDSets:
PDBPropIDSetArray; var pcPropertyInfoSets: UINT; out prgPropertyInfoSets:
PDBPropInfoSet; ppDescBuffer: PPOleStr): HResult; stdcall;
```

который возвращает заполненную структуру `DBPROPINFO`.

```
PDBPropInfo = ^TDBPropInfo;
  DBPROPINFO = packed record
    pwszDescription: PWideChar;
    dwPropertyID: DBPROPID;
    dwFlags: DBPROPFLAGS;
    vtType: Word;
    vValues: OleVariant;
  end;
  TDBPropInfo = DBPROPINFO;
```

Для каждого обязательного параметра в элементе `dwFlags` устанавливается значение `DBPROPFLAGS_REQUIRED`.

Для инициализации соединения необходимо использовать метод

```
function Initialize: HResult; stdcall;
```

интерфейса `IDBInitialize` объекта-источника данных.

Сессия

Из объекта-источника данных можно создавать объекты-сессии. Для этого используется метод

```
function CreateSession(const punkOuter: IUnknown; const riid: TGUID;
out ppDBSession: IUnknown): HResult; stdcall;
```

интерфейса `IDBCreateSession`. Сессия предназначена для обеспечения работы транзакций и наборов рядов.

Транзакции

Управление транзакциями в OLE DB реализовано на двух уровнях.

Во-первых, всеми необходимыми методами обладает объект сессии. Он имеет интерфейсы `ITransaction`, `ITransactionJoin`, `ITransactionLocal`, `ITransactionObject`.

Внутри сессии транзакция управляется интерфейсами `ITransactionLocal`, `ItransactionSC`, `ITransaction` и их методами `StartTransaction`, `Commit`, `Rollback`.

Во-вторых, для объекта сессии можно создать объект транзакции при помощи метода

```
function GetTransactionObject(ulTransactionLevel: UINT;
out ppTransactionObject: ITransaction): HResult; stdcall;
```

интерфейса `ITransactionObject`, который возвращает ссылку на интерфейс объекта-транзакции.

Наборы рядов

Объект-набор рядов является основным объектом ADO, обеспечивающим работу с данными. Он инкапсулирует совокупность рядов из источника данных, механизмы навигации по рядам и поддержания рядов в актуальном состоянии.

Объект сессии имеет обязательный интерфейс `IOpenRowset` с методом

```
function OpenRowset(const punkOuter: IUnknown; pTableID: PDBID; pIndexID:
PDBID; const riid: TGUID; cPropertySets: UINT; rgPropertySets:
PDBPropSetArray; ppRowset: PIUnknown): HResult; stdcall;
```

который открывает необходимый набор рядов.

В зависимости от возможностей источника данных набор рядов может поддерживать различные интерфейсы. Но пять из них являются обязательными:

- `IRowset` — обеспечивает навигацию по рядам;
- `IAccessor` — обеспечивает представление информации о формате рядов, содержащихся в буфере набора рядов;
- `IRowsetInfo` — позволяет получить информацию о наборах рядов (например, число рядов или число обновленных рядов);
- `IColumnsInfo` — позволяет получить информацию о колонках рядов (наименование, тип данных, возможность обновления и т. д.);
- `IConvertType` — содержит единственный метод `CanConvert`, позволяющий определить возможность преобразования типов данных в наборе рядов.

Примечание

В отличие от привычной практики разработки интерфейсов в рамках модели COM, интерфейсы OLE DB часто имеют всего один-два метода. В результате большая группа интерфейсов реализует несколько вполне стандартных функций.

Дополнительные возможности по управлению набором рядов предоставляют следующие интерфейсы:

- `IRowsetChange` — выполняет изменения в наборе рядов (вносит изменения, добавляет новые ряды, удаляет ряды и т. д.);
- `IRowsetIdentity` — позволяет сравнивать ряды разных рядов;
- `IRowsetIndex` — обеспечивает использование индексов;
- `IRowsetLocate` — выполняет поиск в наборе рядов;
- `IRowsetUpdate` — реализует механизм кэширования изменений.

Команды

Программные средства ADO были бы неполными, если бы не имели возможности использовать для работы с данными язык SQL. Операторы DML и DDL, ряд специальных операторов ADO носят общее название *текстовых команд*.

Объект-команда инкапсулирует саму текстовую команду и механизм обработки и передачи команды. Объект команды выполняет следующие операции:

- разбор текста команды;
- связывание команды с источником данных;
- оптимизацию команды;
- передачу команды источнику данных.

Главный интерфейс объекта команды `ICommand` имеет три метода:

- `function Cancel: HResult; stdcall;`

  отменяет выполнение команды;

- `function Execute(const punkOuter: IUnknown; const riid: TGUID; var pParams: DBPARAMS;`
  `pcRowsAffected: PInteger; ppRowset: PIUnknown): HResult; stdcall;`

  исполняет команду;

- `function GetDBSession(const riid: TGUID; out ppSession: IUnknown): HResult; stdcall;`

  возвращает ссылку на интерфейс сессии, вызвавший данную команду.

Помимо основного, объект команды обеспечивает доступ к дополнительным интерфейсам:

- `ICommandPrepare` — содержит два метода (`Prepare` и `Unprepare`) для подготовки команды;
- `ICommandProperties` — задает для команды свойства, которые должны поддерживаться возвращаемым командой набором данных;
- `ICommandText` — управляет текстом команды (этот интерфейс обязателен для объекта команды);
- `ICommandWithParameters` — обеспечивает работу с параметрами команды.

Провайдеры ADO

Провайдеры ADO обеспечивают соединение приложения, использующего данные через ADO, с источником данных (сервером SQL, локальной СУБД, файловой системой и т. д.). Для каждого типа хранилища данных должен существовать провайдер ADO.

Провайдер "знает" о местоположении хранилища данных и его содержании, умеет обращаться к данным с запросами и интерпретировать возвращаемую служебную информацию и результаты запросов с целью их передачи приложению.

Список установленных в данной операционной системе провайдеров доступен для выбора при установке соединения через компонент `TADOConnection`.

При инсталляции Microsoft ActiveX Data Objects в операционной системе устанавливаются следующие стандартные провайдеры.

- *Microsoft Jet OLE DB Provider* обеспечивает соединение с данными СУБД Access при посредстве технологии DAO.
- *Microsoft OLE DB Provider for Microsoft Indexing Service* обеспечивает доступ только для чтения к файлам и Internet-ресурсам Microsoft Indexing Service.
- *Microsoft OLE DB Provider for Microsoft Active Directory Service* обеспечивает доступ к ресурсам службы каталогов (Active Directory Service).
- *Microsoft OLE DB Provider for Internet Publishing* позволяет использовать ресурсы, предоставляемые Microsoft FrontPage, Microsoft Internet Information Server, HTTP-файлы.
- *Microsoft Data Shaping Service for OLE DB* позволяет использовать иерархические наборы данных.

- *Microsoft OLE DB Simple Provider* предназначен для организации доступа к источникам данных, поддерживающим только базисные возможности OLE DB.
- *Microsoft OLE DB Provider for ODBC drivers* обеспечивает доступ к данным, которые уже "прописаны" при помощи драйверов ODBC. Однако реальное использование столь экзотичных вариантов соединений представляется проблематичным. Драйверы ODBC и так славятся своей медлительностью, поэтому дополнительный слой сервисов здесь ни к чему.
- *Microsoft OLE DB Provider for Oracle* обеспечивает соединение с сервером Oracle.
- *Microsoft OLE DB Provider for SQL Server* обеспечивает соединение с сервером Microsoft SQL Server.

Реализация ADO в Delphi

Механизм доступа к данным через ADO и многочисленные объекты и интерфейсы реализованы в VCL Delphi в виде набора компонентов, расположенных на странице **ADO**. Все необходимые интерфейсы, обеспечивающие работу компонентов, объявлены и описаны в файлах OleDB.pas и ADODB.pas в папке \Delphi7\Source\Vcl.

Компоненты ADO

Компонент `TADOConnection` вобрал возможности перечислителя, источника данных и сессии с возможностями обслуживания транзакций.

Текстовые команды ADO реализованы в компоненте `TADOCommand`.

Наборы рядов (нотация Microsoft) можно получить при помощи компонентов `TADOTable`, `TADOQuery`, `TADOStoredProc`. Каждый из них реализует способ доступа к конкретному типу представления данных в хранилище. Далее по тексту, применительно к компонентам Delphi, совокупность возвращаемых из хранилища данных строк будем называть *набором записей*, что соответствует документации Inprise (см. **www.borland.com** или **www.borland.ru**) и стилю изложения предыдущих глав.

Набор свойств и методов компонентов ADO обеспечивает реализацию всех необходимых приложению БД функций. Способы использования компонентов ADO немногим отличаются от стандартных компонентов VCL доступа к данным *(см. гл. 11)*.

Однако при необходимости разработчик может использовать все возможности интерфейсов ADO, обращаясь к ним через соответствующие объекты ADO. Ссылки на объекты имеются в компонентах *(см. ниже)*.

Механизм соединения с хранилищем данных ADO

Компоненты доступа к данным ADO могут использовать два варианта подключения к хранилищу данных. Это стандартный метод ADO и стандартный метод Delphi.

В первом случае компоненты используют свойство `ConnectionString` для прямого обращения к хранилищу данных. Во втором случае используется специальный компонент `TADOConnection`, который обеспечивает расширенное управление соединением и позволяет обращаться к данным нескольким компонентам одновременно.

Свойство `ConnectionString` предназначено для хранения информации о соединении с объектом ADO. В нем через точку с запятой перечисляются все необходимые параметры. Как минимум, это должны быть имена провайдера соединения или удаленного сервера:

```
ConnectionString:='Remote Server=ServerName;Provider=ProviderName';
```

При необходимости указываются путь к удаленному провайдеру:

```
ConnectionString:='Remote Provider=ProviderName';
```

и параметры, необходимые провайдеру:

```
'User Name=User_Name;Password=Password';
```

Каждый компонент, обращающийся к хранилищу данных ADO самостоятельно, задавая параметры соединения в свойстве `ConnectionString`, открывает собственное соединение. Чем больше приложение содержит компонентов ADO, тем больше соединений может быть открыто одновременно.

Поэтому целесообразно реализовать механизм соединения ADO через специальный компонент — `TADOConnection`. Этот компонент открывает соединение, также заданное свойством `ConnectionString` *(см. выше)*, и предоставляет разработчику дополнительные средства управления соединением.

Компоненты, работающие с хранилищем данных ADO через данное соединение, подключаются к компоненту `TADOConnection` при помощи свойства

```
property Connection: TADOConnection;
```

которое имеет каждый компонент, инкапсулирующий набор данных ADO.

Компонент *TADOConnection*

Компонент `TADOConnection` предназначен для управления соединением с объектами хранилища данных ADO. Он обеспечивает доступ к хранилищу данных компонентам ADO, инкапсулирующим набор данных *(см. ниже)*.

Применение этого компонента дает разработчику ряд преимуществ:

- все компоненты доступа к данным ADO обращаются к хранилищу данных через одно соединение;
- возможность напрямую задать объект провайдера соединения;
- доступ к объекту соединения ADO;
- возможность выполнять команды ADO;
- выполнение транзакций;
- расширенное управление соединением при помощи методов-обработчиков событий.

Настройка соединения

Перед открытием соединения необходимо задать его параметры. Для этого предназначено свойство

```
property ConnectionString: WideString;
```

которое подробно рассматривалось в *разд. "Компонент TADOConnection"*. Добавим лишь, что набор параметров изменяется в зависимости от типа провайдера и может настраиваться как вручную, так и при помощи специального редактора параметров соединения, который вызывается двойным щелчком на компоненте `TADOConnection`, перенесенным на форму, или щелчком на кнопке в поле редактирования свойства `ConnectionString` в Инспекторе объектов (рис. 19.2).

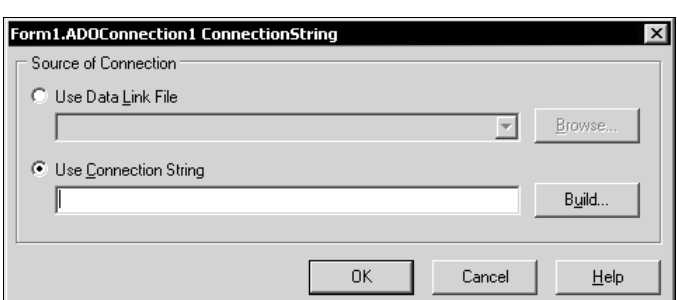


Рис. 19.2. Редактор настройки соединения ADO

Здесь можно настроить соединение через свойство `ConnectionString` (радиокнопка **Use Connection String**) или загрузить параметры соединения из файла с расширением udl (радиокнопка **Use Data Link File**).

Файл UDL (листинг 19.1) представляет собой обычный текстовый файл, в котором указывается название параметра и через знак равенства его значение. Параметры разделяются точкой с запятой.

Листинг 19.1. Демонстрационный файл DBDEMOS.UDL

```
[oledb]
Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Common
Files\Borland Shared\Data\DBDEMOS.mdb
```

Если файл параметров соединения отсутствует, настройку придется осуществлять вручную. Для этого следует нажать кнопку **Build**. В результате появляется диалоговое окно **Data Link Properties**, в котором можно настроить параметры соединения вручную. Оно представляет собой четырехстраничный блокнот, позволяющий вам этап за этапом задать все необходимые параметры (рис. 19.3).

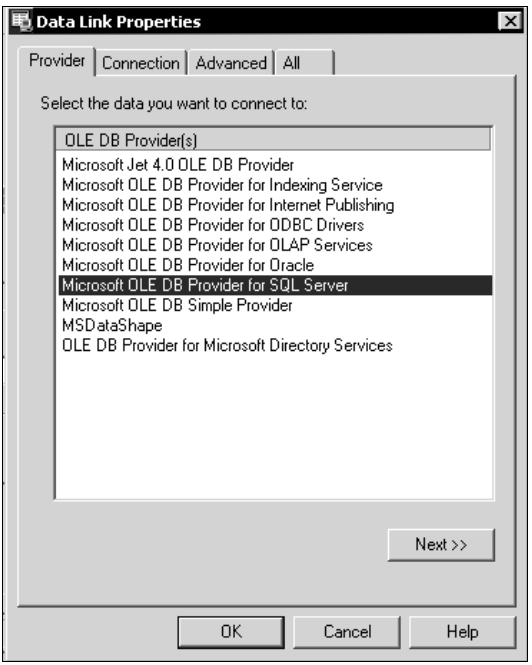


Рис. 19.3. Диалоговое окно настройки параметров соединения на странице выбора провайдера

Первая страница **Provider** позволяет выбрать провайдер OLE DB для конкретного типа источника данных из числа провайдеров, установленных в системе. Здесь вы видите провайдеры не только для серверов БД, но и служб, установленных в операционной системе. Состав элементов управления следующих страниц зависит от типа источника данных, но различается не так уж сильно. Далее практически везде необходимо задать источник

данных (имя сервера, базу данных, файл и т. д.), режим аутентификации пользователя, а также определить имя и пароль пользователя.

Рассмотрим процесс настройки на примере провайдера OLE DB для сервера Microsoft SQL Server.

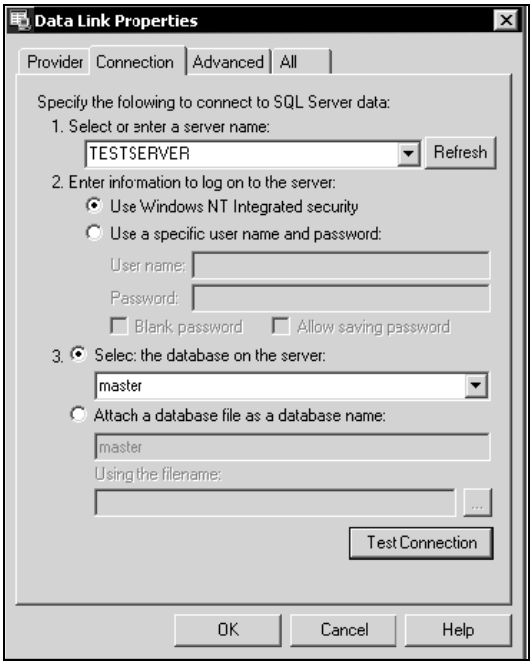


Рис. 19.4. Диалоговое окно настройки параметров соединения на странице выбора источника данных

Следующая страница **Connection** (рис. 19.4) настраивает источник данных.

На первом этапе требуется выбрать имя сервера из доступных для данного компьютера.

Второй этап определяет режим аутентификации пользователя. Это либо система безопасности Windows, либо собственная система аутентификации сервера. Здесь же надо определить имя и пароль пользователя.

Третий этап предназначен для выбора базы данных сервера.

По окончании настройки источника данных вы можете проверить соединение, нажав кнопку **Test Connection**.

Теперь перейдем на следующую страницу.

Страница **Advanced** (рис. 19.5) задает дополнительные параметры соединения. В зависимости от типа хранилища данных некоторые элементы этой страницы могут быть недоступны.

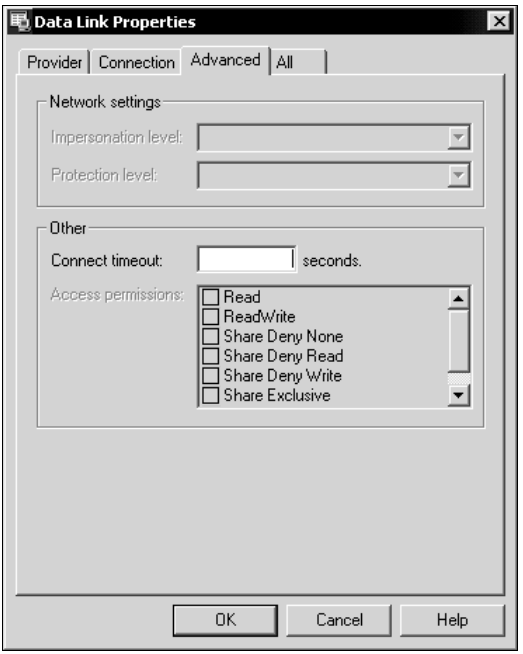


Рис. 19.5. Диалоговое окно настройки параметров соединения на странице дополнительных параметров

Список **Impersonation level** определяет возможности клиентов при подключении в соответствии с полномочиями их ролей. В списке могут быть выбраны следующие значения:

- **Anonymous** — роль клиента недоступна серверу;
- **Identify** — роль клиента опознается сервером, но доступ к системным объектам заблокирован;
- **Impersonate** — процесс сервера может быть представлен защищенным контекстом клиента;
- **Delegate** — процесс сервера может быть представлен защищенным контекстом клиента, при этом сервер может осуществлять другие подключения.

Список **Protection level** позволяет задать уровень защиты данных. В списке могут быть выбраны следующие значения:

- **None** — подтверждение не требуется;
- **Connect** — подтверждение необходимо только при подключении;
- **Call** — подтверждение источника данных при каждом запросе;
- **Pkt** — подтверждение получения от клиента всех данных;
- **Pkt Integrity** — подтверждение получения от клиента всех данных с соблюдением целостности;

- **Pkt Privacy** — подтверждение получения от клиента всех данных с соблюдением целостности и защита шифрованием.

В поле **Connect timeout** можно задать время ожидания соединения в секундах. По истечении этого времени процесс прерывается.

При необходимости список **Access permissions** задает права доступа к отдельным видам выполняемых операций. В списке можно выбрать следующие значения:

- **Read** — только чтение;
- **ReadWrite** — чтение и запись;
- **Share Deny None** — полный доступ всем на чтение и запись;
- **Share Deny Read** — чтение запрещено всем;
- **Share Deny Write** — запись запрещена всем;
- **Share Exclusive** — чтение и запись запрещена всем;
- **Write** — только запись.

Последняя страница **All** (рис. 19.6) позволяет просмотреть и при необходимости изменить все сделанные настройки (для этого предназначена кнопка **Edit Value...**) для выбранного провайдера.

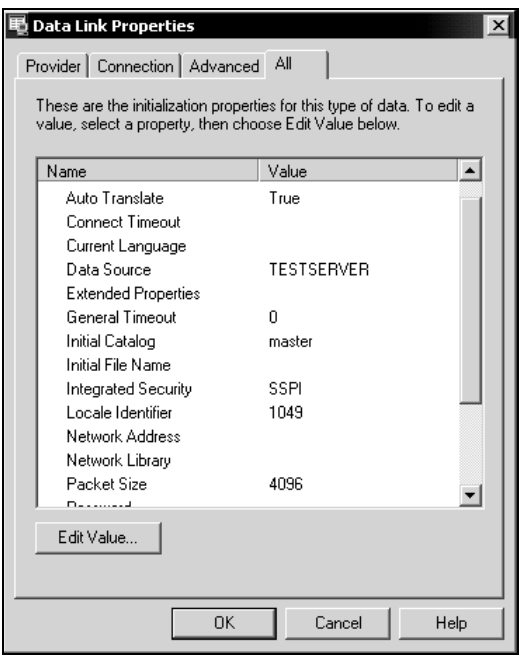


Рис. 19.6. Диалоговое окно настройки параметров соединения на странице просмотра настроек

После подтверждения сделанных в диалоге настроек из них формируется значение свойства `ConnectionString`.

Управление соединением

Соединение с хранилищем данных ADO открывается и закрывается при помощи свойства

```
property Connected: Boolean;
```

или методов

```
procedure Open; overload;
procedure Open(const UserID: WideString; const Password: WideString);
overload;
```

и

```
procedure Close;
```

Метод `Open` является перегружаемым при необходимости использования удаленного или локального соединения. Для удаленного соединения применяется вариант с параметрами `UserID` и `Password`.

До и после открытия и закрытия соединения разработчик может использовать соответствующие стандартные методы-обработчики событий:

```
property BeforeConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property AfterConnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
```

Кроме этого, компонент `TADOConnection` имеет дополнительные методы-обработчики. После получения подтверждения от провайдера о том, что соединение будет открыто, перед его реальным открытием вызывается метод

```
TWillConnectEvent = procedure(Connection: TADOConnection; var
ConnectionString, UserID, Password: WideString; var ConnectOptions:
TConnectOption; var EventStatus: TEventStatus) of object;
property OnWillConnect: TWillConnectEvent;
```

Параметр `Connection` содержит указатель на вызвавший обработчик компонент.

Параметры `ConnectionString`, `UserID` и `Password` определяют строку параметров, имя и пароль пользователя.

Соединение может быть синхронным или асинхронным, что и определяется параметром `ConnectOptions` типа `TConnectOption`:

```
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

`coConnectUnspecified` — синхронное соединение всегда ожидает результат последнего запроса;

`coAsyncConnect` — асинхронное соединение может выполнять новые запросы, не дожидаясь ответа от предыдущих запросов.

Наконец, параметр `EventStatus` позволяет определить успешность выполнения посланного запроса на соединение:

```
type
  TEventStatus = (esOK, esErrorsOccured, esCantDeny, esCancel,
esUnwantedEvent);
```

`esOK` — запрос на соединение выполнен успешно;

`esErrorsOccured` — в процессе выполнения запроса возникла ошибка;

`esCantDeny` — соединение не может быть прервано;

`esCancel` — соединение было прервано до открытия;

`esUnwantedEvent` — внутренний флаг ADO.

Например, в случае успешного соединения можно выбрать синхронный режим работы компонента:

```
procedure TForm1.ADOConnectionWillConnect(Connection: TADOConnection;
var ConnectionString, UserID, Password: WideString; var ConnectOptions:
TConnectOption; var EventStatus: TEventStatus);
begin
 if EventStatus = esOK
  then ConnectOptions := coConnectUnspecified;
end;
```

Кстати, параметр синхронности/асинхронности можно также задать при помощи свойства

```
ConnectOptions property ConnectOptions: TConnectOption;
```

После открытия соединения для выполнения собственного кода можно использовать метод-обработчик

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
var EventStatus: TEventStatus) of object;
property OnConnectComplete: TConnectErrorEvent;
```

Здесь, если в процессе открытия соединения возникла ошибка, параметр `EventStatus` будет равен `esErrorsOccured`, а параметр `Error` содержит объект ошибки ADO.

Теперь перейдем к вспомогательным свойствам и методам компонента `TADOConnection`, обеспечивающим соединение.

Для ограничения времени открытия соединения для медленных каналов связи используется свойство

```
property ConnectionTimeout: Integer;
```

задающее время ожидания открытия соединения в секундах. По умолчанию оно равно 15 сек.

Также можно определить реакцию компонента на неиспользуемое соединение. Если через соединение не подключен ни один активный компонент, свойство

```
property KeepConnection: Boolean;
```

в значении `True` сохраняет соединение открытым. Иначе, после закрытия последнего связанного компонента ADO, соединение закрывается.

При необходимости провайдер соединения ADO определяется напрямую свойством

```
property Provider: WideString;
```

Имя источника данных по умолчанию задается свойством

```
property DefaultDatabase: WideString;
```

Но если этот же параметр указан в строке соединения, то он перекрывает собой значение свойства.

При необходимости прямой доступ к объекту соединения OLE DB обеспечивает свойство

```
property ConnectionObject: _Connection;
```

При открытии соединения необходимо вводить имя пользователя и его пароль. Появление стандартного диалога управляется свойством

```
property LoginPrompt: Boolean;
```

Без этого диалога для задания данных параметров можно использовать свойство `ConnectionString`, метод `Open` *(см. выше)* или метод-обработчик

```
type TLoginEvent = procedure(Sender:TObject; Username, Password: string)
of object;
property OnLogin: TLoginEvent;
```

Свойство

```
type TConnectMode = (cmUnknown, cmRead, cmWrite, cmReadWrite,
cmShareDenyRead, cmShareDenyWrite, cmShareExclusive, cmShareDenyNone);
property Mode: TConnectMode;
```

задает доступные для соединения операции:

- `cmUnknown` — разрешение неизвестно или не может быть установлено;
- `cmRead` — разрешение на чтение;

- `cmWrite` — разрешение на запись;
- `cmReadWrite` — разрешение на чтение и запись;
- `cmShareDenyRead` — разрешение на чтение для других соединений запрещено;
- `cmShareDenyWrite` — разрешение на запись для других соединений запрещено;
- `cmShareExclusive` — разрешение на открытие для других соединений запрещено;
- `cmShareDenyNone` — открытие других соединений с разрешениями запрещено.

Доступ к связанным наборам данных и командам ADO

Компонент `TADOConnection` обеспечивает доступ ко всем компонентам, которые используют его для доступа к хранилищу данных ADO. Все открытые таким образом наборы данных доступны через индексированное свойство

```
property DataSets[Index: Integer]: TCustomADODataSet;
```

Каждый элемент этого списка содержит дескриптор компонента доступа к данным ADO (тип `TCustomADODataSet`). Общее число связанных компонентов с наборами данных возвращается свойством

```
property DataSetCount: Integer;
```

Для этих компонентов можно централизованно установить тип используемого курсора при помощи свойства

```
type TCursorLocation = (clUseServer, clUseClient);
property CursorLocation: TCursorLocation;
```

Значение `clUseClient` задает локальный курсор на стороне клиента, что позволяет выполнять любые операции с данными, в том числе не поддерживаемые сервером.

Значение `clUseServer` задает курсор на сервере, который реализует только возможности сервера, но обеспечивает быструю обработку больших массивов данных.

Например:

```
...
for i := 0 to ADOConnection.DataSetCount - 1 do
 begin
  if ADOConnection.DataSets[i].Active = True
   then ADOConnection.DataSets[i].Close;
```

```
  ADOConnection.DataSets[i].CursorLocation := clUseClient;
 end;
...
```

Помимо наборов данных компонент `TADOConnection` обеспечивает выполнение команд ADO. Команду ADO инкапсулирует специальный компонент `TADOCommand`, который рассматривается ниже. Все команды ADO, работающие с хранилищем данных через это соединение, доступны для управления через индексированное свойство

```
property Commands[Index: Integer]: TADOCommand
```

Каждый элемент этого списка представляет собой экземпляр класса `TADOCommand`.

Общее число доступных команд возвращается свойством

```
property CommandCount: Integer
```

Например, сразу после открытия соединения можно выполнить все связанные команды ADO, реализовав таким образом нечто вроде скрипта:

```
procedure TForm1.ADOConnectionConnectComplete(Connection: TADOConnection;
const Error: Error; var EventStatus: TEventStatus);
var i, ErrorCnt: Integer;
begin
 if EventStatus = esOK then
 for i := 0 to ADOConnection.CommandCount — 1 do
  try
  if ADOConnection.Commands[i].CommandText <> ''
   then ADOConnection.Commands[i].Execute;
  except
   on E: Exception do Inc(ErrorCnt);
  end;
end;
```

Однако компонент `TADOConnection` может выполнять команды ADO самостоятельно, без помощи других компонентов. Для этого используется перегружаемый метод

```
function Execute(const CommandText: WideString; ExecuteOptions:
TExecuteOptions = []): _RecordSet; overload;
procedure Execute(const CommandText: WideString; var RecordsAffected:
Integer; ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]);
overload;
```

Выполнение команды осуществляется процедурой `Execute` (если команда не возвращает набор записей) или одноименной функцией `Execute` (если команда возвращает набор записей).

Параметр `CommandText` должен содержать текст команды. Параметр `RecordsAffected` возвращает число обработанных командой записей (если они есть). Параметр

```
type
 TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking,
eoExecuteNoRecords);
 TExecuteOptions = set of TExecuteOption;
```

задает условия выполнения команды:

- `eoAsyncExecute` — команда выполняется асинхронно (соединение не будет ожидать окончания выполнения команды, а продолжит работу, обработав сигнал о завершении команды, когда он поступит);
- `eoAsyncFetch` — команда получает необходимые записи также асинхронно;
- `eoAsyncFetchNonBlocking` — команда получает необходимые записи также асинхронно, но при этом созданная нить не блокируется;
- `eoExecuteNoRecords` — команда не должна возвращать записи.

Если источник данных принял команду для выполнения и сообщил об этом соединению, вызывается метод-обработчик

```
TWillExecuteEvent = procedure(Connection: TADOConnection;
var CommandText: WideString; var CursorType: TCursorType; var LockType:
TADOLockType; var ExecuteOptions: TExecuteOptions; var EventStatus:
TEventStatus; const Command: _Command; const Recordset: _Recordset)
of object;
property OnWillExecute: TWillExecuteEvent;
```

После выполнения команды вызывается метод-обработчик

```
TExecuteCompleteEvent = procedure(Connection: TADOConnection;
RecordsAffected: Integer; const Error: Error; var EventStatus:
TEventStatus; const Command: _Command; const Recordset: _Recordset)
of object;
property OnExecuteComplete: TExecuteCompleteEvent;
```

Объект ошибок ADO

Все ошибки времени выполнения, возникающие при открытом соединении, сохраняются в специальном объекте ADO, инкапсулирующем коллекцию сообщений об ошибках. Доступ к объекту возможен через свойство

```
property Errors: Errors;
```

Подробнее об объекте ошибок ADO *см. ниже.*

Транзакции

Компонент `TADOConnection` позволяет выполнять транзакции.

Методы

```
function BeginTrans: Integer;
procedure CommitTrans;
procedure RollbackTrans;
```

обеспечивают начало, фиксацию и откат транзакции соответственно.

Методы-обработчики

```
TBeginTransCompleteEvent = procedure(Connection: TADOConnection;
TransactionLevel: Integer; const Error: Error; var EventStatus:
TEventStatus) of object;
property OnBeginTransComplete: TBeginTransCompleteEvent;
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
var EventStatus: TEventStatus) of object;
property OnCommitTransComplete: TConnectErrorEvent;
```

вызываются после начала и фиксации транзакции.

Свойство

```
type TIsolationLevel = (ilUnspecified, ilChaos, ilReadUncommitted,
ilBrowse, ilCursorStability, ilReadCommitted, ilRepeatableRead,
ilSerializable, ilIsolated);
property IsolationLevel: TIsolationLevel;
```

позволяет задать уровень изоляции транзакции:

- `ilUnspecified` — уровень изоляции не задается;
- `ilChaos` — изменения более защищенных транзакций не перезаписываются данной транзакцией;
- `ilReadUncommitted` — незафиксированные изменения других транзакций видимы;
- `ilBrowse` — незафиксированные изменения других транзакций видимы;
- `ilCursorStability` — изменения других транзакций видимы только после фиксации;
- `ilReadCommitted` — изменения других транзакций видимы только после фиксации;
- `ilRepeatableRead` — изменения других транзакций не видимы, но доступны при обновлении данных;
- `ilSerializable` — транзакция выполняется изолированно от других транзакций;

- `ilIsolated` — транзакция выполняется изолированно от других транзакций.

Свойство

```
TXactAttribute = (xaCommitRetaining, xaAbortRetaining);
property Attributes: TXactAttributes;
```

задает способ управления транзакциями при их фиксации и откате:

- `xaCommitRetaining` — после фиксации очередной транзакции автоматически начинается выполнение новой;
- `xaAbortRetaining` — после отката очередной транзакции автоматически начинается выполнение новой.

Наборы данных ADO

На странице **ADO** Палитры компонентов Delphi, кроме компонентов соединения есть стандартные компоненты, инкапсулирующие набор данных и адаптированные для работы с хранилищем данных ADO (рис. 19.7). Это компоненты:

- `TADODataSet` — универсальный набор данных;
- `TADOTable` — таблица БД;
- `TADOQuery` — запрос SQL;
- `TADOStoredProc` — хранимая процедура.

Как и положено для компонентов, инкапсулирующих набор данных, их общим предком является класс `TDataSet`, предоставляющий базовые функции управления набором данных *(см. гл. 11)*.

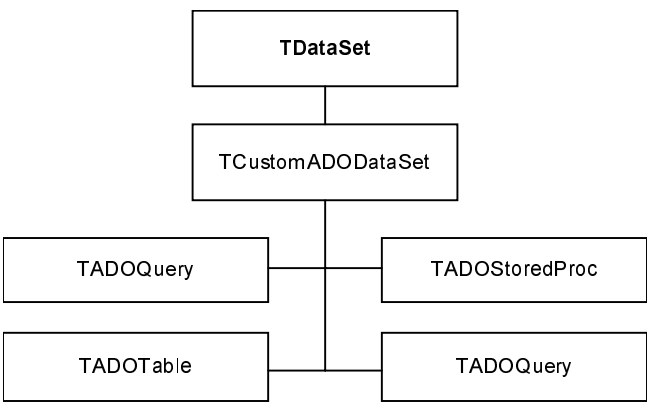


Рис. 19.7. Иерархия классов наборов данных ADO

Компоненты ADO обладают обычным набором свойств и методов, а необходимый для доступа к данным через ADO механизм наследуют от своего общего предка — класса `TCustomADODataSet`. Кроме этого, класс `TCustomADODataSet` содержит ряд общих для всех потомков свойств и методов, рассмотреть которые будет очень полезно. Поэтому сначала мы изучим класс `TCustomADODataSet` и только потом перейдем к компонентам ADO.

Класс *TCustomADODataSet*

Класс `TCustomADODataSet` инкапсулирует механизм доступа к хранилищу данных через ADO. Этот класс наполняет абстрактные методы общего предка `TDataSet` функциями конкретного механизма доступа к данным.

Поэтому здесь мы рассмотрим только уникальные свойства и методы класса `TCustomADODataSet`, обеспечивающие работу с ADO.

Соединение набора данных с хранилищем данных ADO осуществляется через компонент `TADOConnection` (свойство `Connection`) или путем задания параметров соединения через свойство `ConnectionString` *(см. выше)*.

Набор данных

Перед открытием набора данных необходимо установить тип используемой при редактировании записей блокировки. Для этого применяется свойство

```
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic,
ltOptimistic, ltBatchOptimistic);
property LockType: TADOLockType;
```

`ltUnspecified` — блокировка задается источником данных, а не компонентом;

`ltReadOnly` — набор данных откроется в режиме только для чтения;

`ltPessimistic` — редактируемая запись блокируется на все время редактирования до момента сохранения в хранилище данных;

`ltOptimistic` — запись блокируется только на время сохранения изменений в хранилище данных;

`ltBatchOptimistic` — запись блокируется на время сохранения в хранилище данных при вызове метода `UpdateBatch`.

Примечание

Для того чтобы установка блокировки сработала, свойство `LockType` должно быть обязательно модифицировано до открытия набора данных.

Набор данных открывается методом `Open` и закрывается методом `Close`. Также можно использовать свойство

```
property Active: Boolean;
```

Текущее состояние набора данных можно определить свойством

```
type
  TObjectState = (stClosed, stOpen, stConnecting, stExecuting,
stFetching);
  TObjectStates = set of TObjectState;
property RecordsetState: TObjectStates;
```

Набор данных в компонентах ADO основан на использовании объекта набора записей ADO, прямой доступ к этому объекту возможен при помощи свойства

```
property Recordset: _Recordset;
```

Но поскольку все основные методы интерфейсов объекта набора записей ADO перекрыты методами класса, в обычных случаях прямой доступ к объекту вам не понадобится.

После обновления набора данных вызывается метод-обработчик

```
TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error:
Error; var EventStatus: TEventStatus) of object;
property OnFetchComplete: TRecordsetEvent;
```

где `Error` — ссылка на объект ошибки ADO, если она возникла.

Если же набор данных работает в асинхронном режиме, при обновлении вызывается метод-обработчик

```
TFetchProgressEvent = procedure(DataSet: TCustomADODataSet; Progress,
MaxProgress: Integer; var EventStatus: TEventStatus) of object;
property OnFetchProgress: TFetchProgressEvent;
```

где параметр `Progress` показывает долю выполнения операции.

Курсор набора данных

Для набора данных ADO в зависимости от его назначения можно выбрать тип и местоположение используемого курсора.

Местоположение курсора задается свойством

```
type TCursorLocation = (clUseServer, clUseClient);
property CursorLocation: TCursorLocation;
```

Курсор может находиться на сервере (`clUseServer`) или на клиенте (`clUseClient`).

- *Серверный курсор* используется при работе с большими наборами данных, которые нецелесообразно пересылать клиенту целиком. При этом несколько снижается скорость работы клиентского набора данных.

- *Клиентский курсор* обеспечивает передачу набора данных клиенту. Это положительно сказывается на быстродействии, но такой курсор разумно использовать только для небольших наборов данных, не загружающих канал связи с сервером.

При использовании клиентского курсора необходимо дополнительно установить свойство

```
TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
property MarshalOptions: TmarshalOption
```

которое управляет обменом данных клиента с сервером. Если соединение с сервером быстрое, можно использовать значение `moMarshalAll`, разрешающее возврат серверу всех записей набора данных. В противном случае для ускорения работы компонента можно применить свойство `moMarshalModifiedOnly`, обеспечивающее возврат только модифицированных клиентом записей.

Тип курсора определяется свойством

```
TCursorType = (ctUnspecified, ctOpenForwardOnly, ctKeyset, ctDynamic,
ctStatic);
property CursorType: TCursorType;
```

`ctUnspecified` — курсор не задан, тип курсора определяется возможностями источника данных;

`ctOpenForwardOnly` — однонаправленный курсор, допускающий перемещение только вперед; используется при необходимости быстрого одиночного прохода по всем записям набора данных;

`ctKeyset` — двунаправленный локальный курсор, не обеспечивающий просмотр добавленных и удаленных другими пользователями записей;

`ctDynamic` — двунаправленный курсор, отображает все изменения, требует наибольших затрат ресурсов;

`ctStatic` — двунаправленный курсор, полностью игнорирует изменения, внесенные другими пользователями.

Примечание

Если курсор расположен на клиенте (`CursorType = clUseClient`), то для него доступен только один тип — `ctStatic`.

Соответственно до и после каждого перемещения курсора в наборе данных вызываются методы-обработчики:

```
TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; var EventStatus: TEventStatus) of object;
property OnWillMove: TRecordsetReasonEvent;
```

и

```
TRecordsetErrorEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; const Error: Error; var EventStatus: TEventStatus)
of object;
property OnMoveComplete: TRecordsetErrorEvent;
```

где параметр `Reason` позволяет узнать, какой метод вызвал это перемещение.

Локальный буфер

После передачи клиенту записи набора данных размещаются в локальном буфере, размер которого определяется свойством

```
property CacheSize: Integer;
```

Значение свойства есть число записей, помещаемых в локальный буфер, и оно не может быть меньше 1. Очевидно, что при достаточно большом размере буфера компонент будет обращаться к источнику данных не так часто, но при этом большой буфер заметно замедлит открытие набора данных.

Кроме этого, при выборе размера локального буфера необходимо учитывать доступную компоненту память. Это можно сделать путем несложных вычислений:

```
CacheSizeInMem := ADODataSet.CacheSize * ADODataSet.RecordSize;
```

где `RecordSize` — свойство

```
property RecordSize: Word;
```

возвращающее размер одной записи в байтах.

Как видите, компоненты ADO не избежали общей проблемы клиентских данных — при плохом качестве соединения работа приложения замедляется. Однако кое-что все-таки сделать можно. Если при навигации по записям вам не требуется отображать данные в визуальных компонентах пользовательского интерфейса, свойство

```
property BlockReadSize: Integer;
```

позволяет организовать блочную пересылку данных. Оно задает число записей, помещаемых в один блок. При этом набор данных переходит в состояние `dsBlockRead`. По умолчанию блочная пересылка не используется и значение свойства равно 0.

Также можно ограничить максимальный размер набора данных. Свойство

```
property MaxRecords: Integer
```

задает максимальное число записей набора данных. По умолчанию свойство имеет значение 0 и набор данных не ограничен.

Общее число записей набора данных на этот момент возвращает свойство только для чтения

```
property RecordCount: Integer;
```

При достижении последней записи набора данных вызывается метод-обработчик

```
TEndOfRecordsetEvent = procedure (DataSet: TCustomADODataSet;
var MoreData: WordBool; var EventStatus: TEventStatus) of object;
property OnEndOfRecordset: TEndOfRecordsetEvent;
```

При этом параметр `MoreData` показывает, действительно ли достигнут конец набора данных. Если `MoreData = True`, то это означает, что в хранилище данных еще имеются записи, не переданные клиенту.

Состояние записи

Класс `TCustomADODataSet` обладает дополнительными возможностями, которые позволяют отслеживать состояние каждой записи.

Для текущей записи набора данных можно определить ее состояние. Для этого предназначено свойство

```
TRecordStatus = (rsOK, rsNew, rsModified, rsDeleted, rsUnmodified,
rsInvalid, rsMultipleChanges, rsPendingChanges, rsCanceled,
rsCantRelease, rsConcurrencyViolation, rsIntegrityViolation,
rsMaxChangesExceeded, rsObjectOpen, rsOutOfMemory, rsPermissionDenied,
rsSchemaViolation, rsDBDeleted);
property RecordStatus: TRecordStatusSet;
```

где `rsOK` — запись успешно сохранена; `rsNew` — запись добавлена; `rsModified` — запись была изменена; `rsDeleted` — запись удалена; `rsUnmodified` — запись без изменений; `rsInvalid` — запись не может быть сохранена из-за неверной закладки; `rsMultipleChanges` — запись не может быть сохранена из-за множественных изменений; `rsPendingChanges` — запись не может быть сохранена из-за ссылки на несохраненные изменения; `rsCanceled` — операция с записью была отменена; `rsCantRelease` — запись заблокирована; `rsConcurrencyViolation` — запись не может быть сохранена из-за типа блокировки; `rsIntegrityViolation` — нарушена ссылочная целостность; `rsMaxChangesExceeded` — слишком много изменений; `rsObjectOpen` — конфликт с объектом базы данных; `rsOutOfMemory` — недостаток памяти, `rsPermissionDenied` — нет доступа; `rsSchemaViolation` — нарушение структуры данных; `rsDBDeleted` — запись удалена в БД.

Как видите, благодаря этому свойству состояние отдельной записи может быть определено очень точно.

Кроме этого, метод

```
type
  TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
function UpdateStatus: TUpdateStatus; override;
```

возвращает информацию о состоянии текущей записи.

Соответственно до и после изменения записи вызываются методы-обработчики

```
TWillChangeRecordEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; const RecordCount: Integer; var EventStatus:
TEventStatus) of object;
property OnWillChangeRecord: TWillChangeRecordEvent;
```

и

```
TRecordChangeCompleteEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; const RecordCount: Integer; const Error: Error; var
EventStatus: TEventStatus) of object;
property OnRecordChangeComplete: TrecordChangeCompleteEvent;
```

где параметр `Reason` позволяет узнать, какой метод изменил записи, а параметр `RecordCount` возвращает число измененных записей.

Фильтрация

Помимо обычной фильтрации, основанной на свойствах `Filter`, `Filtered` и методе-обработчике `OnFilterRecord`, класс `TCustomADODataSet` предоставляет разработчику дополнительные возможности.

Свойство

```
TFilterGroup = (fgUnassigned, fgNone, fgPendingRecords,
fgAffectedRecords, fgFetchedRecords, fgPredicate, fgConflictingRecords);
property FilterGroup: TFilterGroup;
```

задает групповой фильтр для записей, основываясь на информации о состоянии каждой записи набора данных, подобно рассмотренному выше свойству `RecordStatus`.

Фильтрация возможна по следующим параметрам:

- `fgUnassigned` — фильтр не задан;
- `fgNone` — все ограничения, заданные фильтром, снимаются, отображаются все записи набора данных;
- `fgPendingRecords` — отображаются измененные записи, несохраненные в хранилище данных при вызове метода `UpdateBatch` или `CancelBatch`;

- fgAffectedRecords — показываются записи, обработанные при последнем сохранении в хранилище данных;
- fgFetchedRecords — имеем записи, полученные при последнем обновлении из источника данных;
- fgPredicate — видны только удаленные записи;
- fgConflictingRecords — отображаются модифицированные записи, при сохранении которых в хранилище данных возникла ошибка.

Для того чтобы групповая фильтрация заработала, необходимы два дополнительных условия. Во-первых, фильтрация должна быть включена — свойство Filtered должно иметь значение True. Во-вторых, свойство LockType должно иметь значение ltBatchOptimistic.

```
with ADODataSet do
 begin
  Close;
  LockType := ltbatchOptimistic;
  Filtered := True;
  FilterGroup := fgFetchedRecords;
  Open;
 end;
```

Метод

```
procedure FilterOnBookmarks(Bookmarks: array of const);
```

включает фильтрацию по существующим закладкам. Для этого предварительно необходимо при помощи метода GetBookmark установить закладки на интересующих записях. При вызове метода FilterOnBookmarks автоматически очищается свойство Filter, а свойству FilterGroup присваивается значение gUnassigned.

Поиск

Быстрый и гибкий поиск по полям текущего индекса набора данных обеспечивает метод

```
SeekOption = (soFirstEQ, soLastEQ, soAfterEQ, soAfter, soBeforeEQ,
soBefore);
function Seek(const KeyValues: Variant; SeekOption: TSeekOption =
soFirstEQ): Boolean;
```

В параметре KeyValues должны быть перечислены необходимые значения полей индекса. Параметр SeekOption управляет процессом поиска:

- soFirstEQ — курсор устанавливается на первую найденную запись;
- soLastEQ — курсор устанавливается на последнюю найденную запись;

- soAfterEQ — курсор устанавливается на найденную запись или, если запись не найдена, сразу после того места, где она могла находиться;
- soAfter — курсор устанавливается сразу после найденной записи;
- soBeforeEQ — курсор устанавливается на найденную запись или, если запись не найдена, перед тем местом, где она могла находиться;
- soBefore — курсор устанавливается перед найденной записью.

Сортировка

Свойство

```
property Sort: WideString;
```

предоставляет простой способ сортировки по произвольному сочетанию полей. Оно должно включать через запятую имена нужных полей и признак прямого или обратного порядка сортировки:

```
ADODataSet.Sort := 'FirstField DESC';
```

Если порядок сортировки не указан, по умолчанию задается прямой порядок.

Команда ADO

Для выполнения запросов к источнику данных любой компонент ADO инкапсулирует специальный объект команды ADO.

При использовании компонентов-потомков класса TCustomADODataSet обычно нет необходимости применять объект команды напрямую. И хотя все реальное взаимодействие объекта набора данных ADO с источником данных осуществляется через объект команды, настройка и выполнение команды скрыты в свойствах и методах компонентов ADO. Тем не менее в классе TCustomADODataSet доступ к объекту команды можно получить при помощи свойства

```
property Command: TADOCommand;
```

Примечание

При необходимости выполнить команду ADO, не связанную с конкретным набором данных, разработчик может использовать отдельный компонент TADOCommand, также расположенный на странице **ADO** Палитры компонентов.

Тип команды задается свойством

```
type
  TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile,
  cmdTableDirect);
property CommandType: TCommandType;
```

`cmdUnknown` — тип команды неизвестен и будет определен источником данных;

`cmdText` — текстовая команда, интерпретируемая источником данных (например запрос SQL); текст должен быть составлен с учетом правил для конкретного источника данных;

`cmdTable` — команда на получение набора данных таблицы из хранилища данных;

`cmdStoredProc` — команда на выполнение хранимой процедуры;

`cmdFile` — команда на получение набора данных, сохраненного в файле в формате, используемым конкретным источником данных;

`cmdTableDirect` — команда на получение набора данных таблицы напрямую, например из файла таблицы.

Текст команды, представленный свойством

```
property CommandText: WideString;
```

обязательно должен быть согласован с ее типом.

Для ограничения времени ожидания выполнения команды используется свойство

```
property CommandTimeout: Integer;
```

В компонентах наборов данных ADO команды выполняется при выполнении следующих операций:

- открытие и закрытие набора данных;
- выполнение запросов и хранимых процедур;
- обновление набора данных;
- сохранение изменений;
- групповые операции.

Разработчик может повлиять на способ выполнения команды. Для этого он может изменить свойство

```
type
  TExecuteOption = (eoAsyncExecute, eoAsyncFetch,
eoAsyncFetchNonBlocking, eoExecuteNoRecords);
  TExecuteOptions = set of TExecuteOption;
property ExecuteOptions: TExecuteOptions;
```

`eoAsyncExecute` — асинхронное выполнение команды;

`eoAsyncFetch` — асинхронное выполнение команды на обновление набора данных;

`eoAsyncFetchNonBlocking` — асинхронное выполнение команды на обновление набора данных без установки блокировки;

`eoExecuteNoRecords` — выполнение команды не требует возвращения набора данных.

Групповые операции

Как уже рассказывалось выше, наборы данных ADO используют на клиентской стороне локальный кэш для хранения данных и сделанных изменений. Благодаря наличию этого кэша и появилась возможность реализовать групповые операции. В этом режиме все сделанные изменения не передаются немедленно источнику данных, а накапливаются в локальном кэше. Это повышает скорость работы и позволяет сохранять сразу группу модифицированных записей.

Из отрицательных сторон этого метода стоит отметить, что пока изменения находятся на клиенте, они недоступны другим пользователям. В данной ситуации могут возникать потери данных.

Для перевода набора данных в режим групповых операций необходимо выполнить следующие условия.

- Набор данных должен использовать клиентский курсор:

  ```
  ADODataSet.CursorLocation := clUseClient;
  ```

- Курсор должен иметь тип `ctStatic`:

  ```
  ADODataSet.CursorType := ctStatic;
  ```

- Блокировка должна иметь значение `ltBatchOptimistic`:

  ```
  ADODataSet.LockType := ltBatchOptimistic;
  ```

Для передачи сделанных изменений в хранилище данных в компонентах ADO используется метод

```
procedure UpdateBatch(AffectRecords: TAffectRecords = arAll);
```

Для отмены всех сделанных, но не сохраненных методом `UpdateBatch` изменений применяется метод

```
procedure CancelBatch(AffectRecords: TAffectRecords = arAll);
```

Используемый в методах тип `TAffectRecords` позволяет задать тип записей, к которым применяется данная операция:

```
TAffectRecords = (arCurrent, arFiltered, arAll, arAllChapters);
```

`arCurrent` — операция выполняется только для текущей записи;

`arFiltered` — операция выполняется для записей из работающего фильтра;

`rAll` — операция выполняется для всех записей;

`arAllChapters` — операция выполняется для всех записей текущего набора данных (включая невидимые из-за включенного фильтра), а также для всех вложенных наборов данных.

Параметры

Многие компоненты ADO, инкапсулирующие набор записей, должны обеспечивать применение параметров запросов. Для этого в них используется специальный класс `TParameters`.

Для каждого параметра из коллекции класса `TParameters` создается отдельный класс `TParameter`.

Этот класс является наследником класса коллекции `TCollection` и инкапсулирует индексированный список отдельных параметров *(см. ниже)*. Напомним, что для работы с параметрами обычных запросов в компонентах запросов и хранимых процедур используется класс `TParams` (например в компонентах dbExpress), также происходящий от класса коллекции.

Методы этих двух классов совпадают, а свойства имеют некоторые отличия. Для представления параметров команд в ADO имеется специальный объект параметров, который активно используется в процессе работы компонентов ADO, инкапсулирующих набор данных.

Поэтому для компонентов ADO в VCL был создан собственный класс параметров.

Класс *TParameters*

Главное, для чего предназначен класс `TParameters`, — содержать список параметров. Индексированный список параметров представлен свойством

```
property Items[Index: Integer]: TParameter;
```

Текущие значения параметров можно получить из индексированного свойства

```
property ParamValues[const ParamName: String]: Variant;
```

При этом доступ к конкретному значению осуществляется по имени параметра:

```
Edit1.Text := ADODataSet.Parameters.ParamValues['ParamOne'];
```

Список параметров можно обновлять при помощи методов

```
function AddParameter: TParameter;
```

и

```
function CreateParameter(const Name: WideString; DataType: TDataType;
Direction: TParameterDirection; Size: Integer; Value: OleVariant):
TParameter;
```

Первый метод просто создает новый объект параметра и добавляет его к списку. Затем необходимо задать все свойства нового параметра:

```
var NewParam: TParameter;
...
 NewParam := ADODataSet.Parameters.AddParameter;
 NewParam.Name := 'ParamTwo';
 NewParam.DataType := ftInteger;
 NewParam.Direction := pdInput;
 NewParam.Value := 0;
```

Метод `CreateParameter` создает новый параметр и определяет его свойства:

- `Name` — имя параметра;
- `DataType` — тип данных параметра, соответствующий типу поля таблицы БД (тип `TFieldType`);
- `Direction` — тип параметра, в дополнение к стандартным типам `dUnknown`, `pdInput`, `pdOutput`, `pdInputOutput`, тип `TParameterDirection` имеет дополнительный тип `pdReturnValue`, определяющий любое возвращаемое значение;
- `Size` — максимальный размер значения параметра;
- `Value` — значение параметра.

При работе с параметрами полезно вызывать их, используя имена, а не абсолютные индексы в списке. Для этого можно использовать метод

```
function ParamByName(const Value: WideString): TParameter;
```

Список параметров всегда должен соответствовать запросу или хранимой процедуре. Для обновления списка используется метод

```
procedure Refresh;
```

Также вы можете создать список параметров для не связанного с данным объектом параметров запроса. Для этого используется метод

```
function ParseSQL(SQL: String; DoCreate: Boolean): String;
```

где `DoCreate` определяет, удалять ли перед анализом существующие параметры.

Класс *TParameter*

Класс `TParameter` инкапсулирует отдельный параметр.

Имя параметра определяется свойством

```
property Name: WideString;
```

Тип данных, которому должно соответствовать его значение, задается свойством

```
TDataType = TFieldType;
property DataType: TDataType;
```

И так как параметры взаимодействуют с полями таблиц БД, то тип данных параметров совпадает с типами данных полей. От типа данных зависит размер параметра

```
property Size: Integer;
```

который может быть изменен для строкового или символьного типа данных и им подобных.

Само значение параметра содержится в свойстве

```
property Value: OleVariant;
```

А свойство

```
type
  TParameterAttribute = (paSigned, paNullable, paLong);
  TParameterAttributes = set of TParameterAttribute;
property Attributes: TParameterAttributes;
```

контролирует значение, присваиваемое параметру:

- `paSigned` — значение может быть символьным;
- `paNullable` — значение параметра может быть пустым;
- `paLong` — значение может содержать данные типа BLOB.

Тип параметра задается свойством

```
type TParameterDirection = (pdUnknown, pdInput, pdOutput, pdInputOutput,
pdReturnValue);
property Direction: TParameterDirection;
```

`pdUnknown` — неизвестный тип, источник данных попытается определить его самостоятельно;

`pdInput` — входной параметр, используется в запросах и хранимых процедурах;

`pdOutput` — выходной параметр, используется в хранимых процедурах;

`pdInputOutput` — входной и выходной параметр одновременно, используется в хранимых процедурах;

`pdReturnValue` — параметр для передачи значения.

Если параметр должен передавать большие бинарные массивы (например, изображения или файлы), то значение параметра можно загрузить, используя методы

```
procedure LoadFromFile(const FileName: String; DataType: TDataType);
```

и

```
procedure LoadFromStream(Stream: TStream; DataType: TDataType);
```

Компонент *TADODataSet*

Компонент `TADODataSet` предназначен для представления набора данных из хранилища данных ADO. Он прост в использовании, имея только несколько собственных свойств и методов, и применяет функции своего предка — класса `TCustomADODataSet`.

Это единственный компонент ADO, инкапсулирующий набор данных, для которого опубликованы свойства, позволяющие управлять командой ADO. Это свойства *(см. выше)*

```
property CommandText: WideString;
```

и

```
property CommandType: TCommandType;
```

В результате компонент представляет собой гибкий инструмент, который позволяет (в зависимости от типа команды и ее текста) получать данные из таблиц, запросов SQL, хранимых процедур, файлов и т. д. Например, вы выбираете нужное значение свойства `CommandType = cmdText` и заносите в свойство `CommandText` текст запроса SQL из редактора:

```
ADODataSet.CommandType = cmdText;
ADODataSet. CommandText := Memo1.Lines.Text;
```

И запрос SQL готов к выполнению.

Примечание

Для запросов SQL можно применять только язык Data Manipulation Language (использовать только SELECT).

Соединение с базой данных задается свойством `ConnectionString` или `Connection` *(см. выше)*.

Набор данных открывается и закрывается свойством `Active` или методами `Open` и `Close`.

В приложениях компонент можно применять как все обычные компоненты доступа к данным, связывая инкапсулированный в нем набор данных с визуальными компонентами отображения данных через компонент `TDataSource`.

Компонент *TADOTable*

Компонент `TADOTable` обеспечивает использование в приложениях Delphi таблиц БД, подключенных через провайдеры OLE DB. По своим функциональным возможностям и применению он подобен стандартному табличному компоненту *(см. гл. 11)*.

Как вы уже знаете, в основе компонента лежит использование команды ADO, но ее свойства настроены заранее и изменению не подлежат.

Имя таблицы БД задается свойством

```
property TableName: WideString;
```

Другие свойства и методы компонента обеспечивают применение индексов (этой возможности лишен любой компонент запроса).

Так как не все провайдеры ADO обеспечивают прямое использование таблиц БД, то для доступа к ним может понадобиться запрос SQL. Если свойство

```
property TableDirect: Boolean;
```

имеет значение `True`, осуществляется прямой доступ к таблице. В противном случае компонент генерирует соответствующий запрос.

Свойство

```
property ReadOnly: Boolean;
```

позволяет включить или отключить для таблицы режим "только для чтения".

Компонент *TADOQuery*

Компонент `TADOQuery` обеспечивает применение запросов SQL при работе с данными через ADO. По своей функциональности он подобен стандартному компоненту запроса *(см. гл. 11)*.

Текст запроса задается свойством

```
property SQL: TStrings;
```

Параметры запроса определяются свойством

```
property Parameters: TParameters;
```

Если запрос должен возвращать набор данных, для его открытия используется свойство

```
property Active: Boolean;
```

или метод

```
procedure Open;
```

В противном случае достаточно использовать метод

```
function ExecSQL: Integer; ExecSQL
```

Число обработанных запросом записей возвращает свойство

```
property RowsAffected: Integer;
```

Компонент *TADOStoredProc*

Компонент `TADOStoredProc` позволяет использовать в приложениях Delphi, обращающихся к данным через ADO, хранимые процедуры. Он подобен стандартному компоненту хранимой процедуры *(см. гл. 11)*.

Имя хранимой процедуры определяется свойством

```
property ProcedureName: WideString;
```

Для определения входных и выходных параметров используется свойство

```
property Parameters: TParameters;
```

Если процедура будет применяться без изменений многократно, имеет смысл заранее подготовить ее выполнение на сервере. Для этого свойству

```
property Prepared: Boolean;
```

присваивается значение `True`.

Команды ADO

Команде ADO, которой мы уделяли так много внимания в этой главе в VCL Delphi, соответствует компонент `TADOCommand`. Методы этого компонента во многом совпадают с методами класса `TCustomADODataSet`, хотя этот класс не является предком компонента (рис. 19.8). Он предназначен для выполнения команд, которые не возвращают наборы данных.

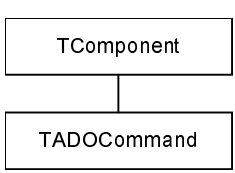


Рис. 19.8. Иерархия классов компонента `TADOCommand`

Так как компоненту `TADOCommand` нет необходимости обеспечивать работу набора записей, его непосредственным предком является класс `TComponent`. К его функциональности просто добавлен механизм соединения с БД через ADO и средства представления команды.

Команда передается в хранилище данных ADO через собственное соединение или через компонент `TADOConnection`, аналогично другим компонентам ADO *(см. выше)*.

Текстовое представление выполняемой команды должно содержаться в свойстве

```
property CommandText: WideString;
```

Однако команду можно задать и другим способом. Прямая ссылка на нужный объект команды ADO может быть задана свойством

```
property CommandObject: _Command;
```

Тип команды определяется свойством

```
type TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc,
cmdFile, cmdTableDirect);
property CommandType: TCommandType;
```

Так как тип `TCommandType` также используется в классе `TCustomADODataSet`, где необходимо представлять все возможные виды команд, по отношению к компоненту `TADOCommand` этот тип обладает избыточностью. Здесь нельзя установить значения `cmdTable`, `cmdFile`, `cmdTableDirect`, а тип `cmdStoredProc` должен обозначать только те хранимые процедуры, которые не возвращают набор данных.

Если команда должна содержать текст запроса SQL, свойство `CommandType` должно иметь значение `cmdText`.

Для вызова хранимой процедуры необходимо задать тип `cmdStoredProc`, а в свойстве `CommandText` ввести имя процедуры.

Если для выполнения команды необходимо задать параметры, используется свойство

```
property Parameters: TParameters;
```

Выполнение команды осуществляется методом `Execute`:

```
function Execute: _RecordSet; overload;
function Execute(const Parameters: OleVariant): _Recordset; overload;
function Execute(var RecordsAffected: Integer; var Parameters:
OleVariant; ExecuteOptions: TExecuteOptions = []): _RecordSet; overload;
```

Разработчик может использовать любую из представленных нотаций перегружаемого метода:

- параметр `RecordsAffected` возвращает число обработанных записей;
- параметр `Parameters` задает параметры команды;

- параметр `ExecuteOptions` определяет условия выполнения команды:

```
TExecuteOption = (eoAsyncExecute, eoAsyncFetch,
eoAsyncFetchNonBlocking, eoExecuteNoRecords);
  TExecuteOptions = set of TExecuteOption;
```

  `eoAsyncExecute` — асинхронное выполнение команды;

  `eoAsyncFetch` — асинхронная передача данных;

  `eoAsyncFetchNonBlocking` — асинхронная передача данных без блокирования потока;

  `eoExecuteNoRecords` — если команда возвращает набор записей, то они не передаются в компонент.

При работе с компонентом `TADOConnection` желательно использовать опцию `eoExecuteNoRecords`.

Для прерывания выполнения команды используется метод

```
procedure Cancel;
```

Текущее состояние команды можно определить свойством

```
type
  TObjectState = (stClosed, stOpen, stConnecting, stExecuting,
stFetching);
  TObjectStates = set of TObjectState;
property States: TObjectStates;
```

Объект ошибок ADO

При рассказе о компонентах ADO в данной главе мы довольно часто упоминали об объектах ошибок ADO. Эти объекты содержат информацию об ошибке, возникшей при выполнении операции каким-либо объектом ADO.

В Delphi для объекта ошибки не предусмотрен специальный тип, но разработчик может использовать его методы интерфейса `Error`, предоставляемого многими методами других объектов ADO. Например, тип

```
TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error:
Error; var EventStatus: TEventStatus) of object;
```

используемый для метода-обработчика, вызываемого после обновления набора данных, содержит параметр `Error`, дающий нам искомую ссылку.

Рассмотрим полезные свойства объекта ошибок ADO.

Свойство

```
property Description: WideString read Get_Description;
```

возвращает описание ошибки, переданное из объекта, в котором ошибка произошла.

Свойство

```
property SQLState: WideString read Get_SQLState;
```

содержит текст команды, вызвавшей ошибку.

Свойство

```
property NativeError: Integer read Get_NativeError;
```

возвращает код ошибки, переданный из объекта, в котором ошибка произошла.

Пример приложения ADO

Теперь попробуем применить на практике представленную в этой главе информацию о реализации ADO в Delphi. В качестве примера создадим простое приложение ADO Demo, которое "умеет" отображать пару таблиц БД, сохранять изменения при помощи групповых операций, сортировать записи и устанавливать фильтры на выбранные записи (рис. 19.9).

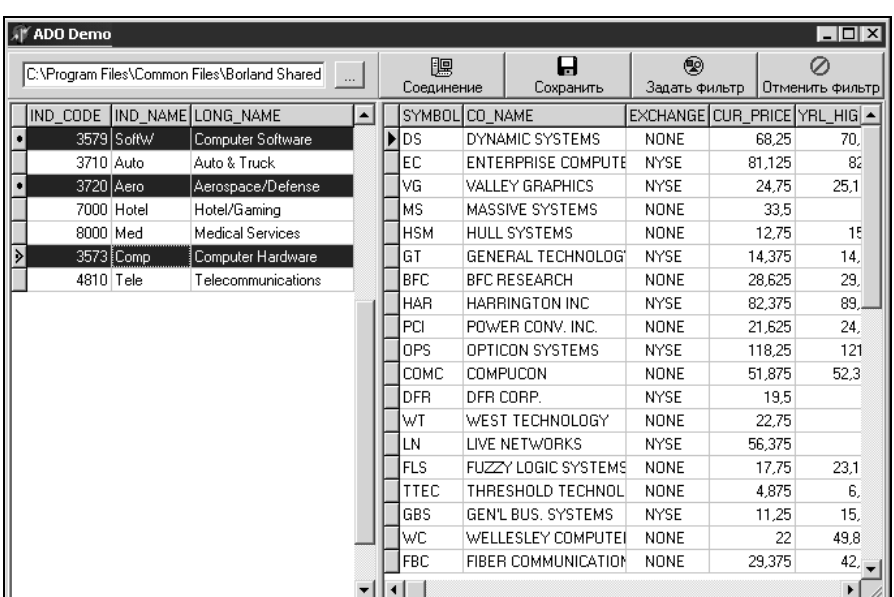


Рис. 19.9. Главное окно приложения ADO Demo

В качестве источника данных выберем файлы dBase, имеющиеся в демонстрационной базе данных Delphi \Program Files\Common Files\Borland Shared \Data. Для использования в приложении выберем две таблицы: INDUSTRY и MASTER. Они связаны между собой внешним ключом по полям `IND_CODE` и `INDUSTRY` соответственно.

Таблицу INDUSTRY можно редактировать, она инкапсулирована в компоненте `tblIndustry` типа `TADOTable` и отображается в левом компоненте `TDBGrid`. А таблица MASTER инкапсулирована в компоненте `tblMaster`, предназначена только для просмотра. Эти два компонента связаны отношением "один-ко-многим" при помощи свойств `MasterSource` и `MasterFields`.

Листинг 19.2. Секция `implementation` модуля `uMain` приложения ADO Demo

```
implementation

uses IniFiles, FileCtrl;

const sIniFileName: String = 'ADODemo.ini';
      sEmptyDefDB:  String = 'Database path is empty';
      sEmptyFilter: String = 'Records for filter is not selected';

{$R *.dfm}

procedure TfmMain.FormShow(Sender: TObject);
begin
 with TIniFile.Create(sIniFileName) do
 try
  DefDBStr := ReaDString('DefDB', 'DefDBStr', '');
  edDefDB.Text := DefDBStr;
 finally
  Free;
 end;
 SetLength(Bookmarks, 0);
end;

procedure TfmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 with TIniFile.Create(sIniFileName) do
 try
  WriteString('DefDB', 'DefDBStr', edDefDB.Text);
 finally
  Free;
 end;
end;
```

```
procedure TfmMain.sbDefDBClick(Sender: TObject);
begin
 if SelectDirectory(DefDBStr, [], 0)
  then edDefDB.Text := DefDBStr;
end;

procedure TfmMain.tbConnectClick(Sender: TObject);
begin
 ADOConn.Close;
 ADOConn.DefaultDatabase := '';
 if DefDBStr = '' then
 begin
  MessageDlg(sEmptyDefDB, mtError, [mbOK], 0);
  Abort;
 end
 else
 begin
  ADOConn.DefaultDatabase := DefDBStr;
  ADOConn.Open;
 end;
end;

procedure TfmMain.tbSaveClick(Sender: TObject);
begin
 tblIndustry.UpdateBatch();
end;

procedure TfmMain.tbFilterClick(Sender: TObject);
var i: Integer;
begin
 if dbgIndustry.SelectedRows.Count > 0 then
 begin
  SetLength(Bookmarks, dbgIndustry.SelectedRows.Count);
  for i := 0 to dbgIndustry.SelectedRows.Count — 1 do
  begin
   Bookmarks[i].VType := vtPointer;
   Bookmarks[i].VPointer := pointer(dbgIndustry.SelectedRows[i]);
  end;
  tblIndustry.FilterOnBookmarks(Bookmarks);
 end
 else
  MessageDlg(sEmptyFilter, mtWarning, [mbOK], 0);
end;
```

```
procedure TfmMain.tbUnFilterClick(Sender: TObject);
begin
 tblIndustry.Filtered := False;
 dbgIndustry.SelectedRows.Clear;
end;

procedure TfmMain.dbgIndustryTitleClick(Column: TColumn);
begin
 if tblIndustry.Active then
  if (Pos(Column.FieldName, tblIndustry.Sort) > 0)and(Pos('ASC',
tblIndustry.Sort) > 0)
   then tblIndustry.Sort := Column.FieldName + ' DESC'
   else tblIndustry.Sort := Column.FieldName + ' ASC';
end;

procedure TfmMain.ADOConnAfterConnect(Sender: TObject);
var i: Integer;
begin
  for i := 0 to adoConn.DataSetCount - 1 do
   ADOConn.DataSets[i].Open;
end;

procedure TfmMain.ADOConnBeforeDisconnect(Sender: TObject);
var i: Integer;
begin
  for i := 0 to adoConn.DataSetCount - 1 do
   ADOConn.DataSets[i].Close;
end;

end.
```

Соединение с источником данных

Для связывания приложения с источником данных используем компонент `TADOConnection` и настроим соединения, щелкнув на кнопке свойства `ConnectionString` в Инспекторе объектов.

Перейдя в редактор **Data Link Properties**, выберем провайдер Microsoft OLE DB Provider for OLE DB Drivers (см. рис. 19.3). Как правило, он имеется в операционной системе, если вы не предпринимали специальных усилий по его удалению.

Далее, на странице **Connection** (см. рис. 19.4) выберем радиокнопку **Use data source name** и в списке — файлы dBase. Для создания соединения с провайдером ODBC этого вполне достаточно.

Прокомментируем другие свойства компонента соединения ADO.

Свойство `LoginPrompt` должно иметь значение `False`, чтобы запретить показ диалога авторизации пользователя, ненужный для файлов dBase.

Свойство `DefaultDatabase` пока останется пустым. Мы применим его для указания пути к файлам базы данных, используя элементы пользовательского интерфейса приложения.

Свойство `CursorLocation` имеет значение `clUseClient`, чтобы обеспечить использование курсоров наборов данных на стороне клиента.

Свойство `ConnectOptions` имеет значение по умолчанию `coConnectUnspecified`. Это означает, что все команды будут выполняться синхронно — соединение будет ожидать ответ на каждую команду.

Для свойства `Mode` установим значение `cmShareDenyNone`, что запрещает другим соединениям устанавливать любые ограничения — ведь в данном случае мы не планируем многопользовательскую работу с источником данных.

Для открытия соединения после запуска приложения необходимо задать путь к хранилищу данных. Для этого предназначена кнопка и однострочный редактор на Панели управления. После выбора пути его значение заносится в переменную `DefDBStr` и в текст редактора `edDefDB`. Переменная используется для установления соединения. Для включения соединения необходимо нажать кнопку `tbConnect`. Ее метод-обработчик проверяет состояние переменной `DefDBStr` и заполняет свойство `DefaultDatabase` компонента соединения.

Примечание

Так как во время настройки соединения выше мы не задавали путь к хранилищу данных, то свойство `DefaultDatabase` сработает. Иначе его значение будет перекрыто настройками свойства `ConnectionString`.

Открытие наборов данных ADO в приложении выполняется в методе-обработчике `ADOConnAfterConnect`, который вызывается после полного открытия соединения. Аналогичным образом наборы данных закрываются перед закрытием соединения в методе-обработчике `ADOConnBeforeDisconnect`.

Текущее значение пути к хранилищу данных сохраняется в файле DemoADO.ini и загружается при открытии приложения.

Групповые операции

Компонент `tblIndustry` предназначен для выполнения групповых операций. Поэтому его свойство `LoclType` имеет значение `ltBatchOptimistic`. Для свойства `CursorLocation` установлено значение `clUseClient`, чтобы обеспечить

использование набора данных на клиенте. Тип курсора (свойство CursorType) должен быть ctStatic.

Сохранение изменений в хранилище данных обеспечивает метод UpdateBatch в методе-обработчике нажатия кнопки tbSave.

Фильтрация

Для фильтрации записей в наборе данных tblIndustry используется метод FilterOnBookmark. Пользователь должен выбрать интересующие его записи в компоненте dbgIndustry (он работает в режиме dgMultiSelect). Затем, при нажатии кнопки tbFilter, созданные в свойстве SelectedRows компонента dbgIndustry закладки передаются в массив Bookmarks типа TVarRec, который потом передается в качестве параметра метода FilterOnBookmark для фильтрации.

Массив Bookmarks служит здесь лишь промежуточным звеном для приведения типа закладок компонента dbgIndustry к параметру метода FilterOnBookmark.

Сортировка

Сортировка создана также для набора данных tblIndustry. При щелчке на заголовке колонки компонента dbgIndustry вызывается метод-обработчик dbgIndustryTitleClick. В нем, в зависимости от текущего состояния свойства сортировки tblIndustry.Sort (какое поле сортируется и в каком порядке), задается новое значение свойства Sort.

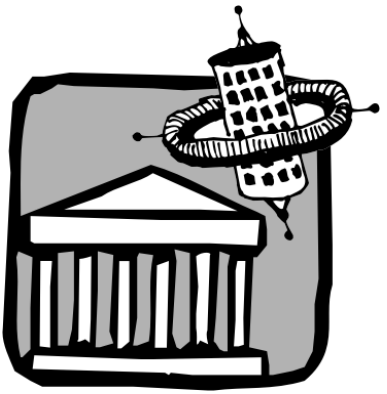
Резюме

Технология ADO обеспечивает универсальный способ доступа к гетерогенным источникам данных. Благодаря тому, что функции ADO реализованы на основе интерфейсов OLE DB и COM, приложению для доступа к данным не требуется дополнительных библиотек, кроме инсталлированного ADO.

Компонент TADOConnection обеспечивает соединение с источниками данных через провайдеры OLE DB. Компоненты TADODataSet, TADOTable, TADOQuery, TADOStoredProc обеспечивают использование наборов записей в приложении. Свойства и методы компонентов позволяют создавать полнофункциональные приложения.

Компонент TADOCommand инкапсулирует текстовую команду ADO.

В дополнение к стандартным возможностям работы с данными, из компонентов можно напрямую обращаться к необходимым объектам и интерфейсам ADO.



ЧАСТЬ V

Распределенные приложения баз данных

Глава 20. Технология DataSnap. Механизмы удаленного доступа

Глава 21. Сервер приложения

Глава 22. Клиент многозвенного распределенного приложения

ГЛАВА 20

Технология DataSnap. Механизмы удаленного доступа

В главах *части IV* мы рассматривали вопросы создания обычных приложений БД, работающих с базами данных на локальных компьютерах или в пределах локальной сети. Однако, как быть, если необходимо создать приложение, которое может с одинаковым успехом работать как в локальной сети, так и на удаленном компьютере.

Очевидно, что в этом случае модель доступа к данным должна быть расширена, т. к. наличие большого числа удаленных клиентов делает традиционные схемы создания приложений БД малоэффективными.

В этой главе мы рассмотрим модель распределенного приложения БД, которая называется *многозвенной* (multitiered), и, в частности, ее наиболее простой вариант — *трехзвенное распределенное приложение*. Тремя частями такого приложения являются:

- собственно сервер базы данных;
- сервер приложений (серверная часть приложения);
- клиентская часть приложения.

Все они объединены в единое целое единым механизмом взаимодействия (транспортный уровень) и обработки данных (уровень бизнес-логики).

Компоненты и объекты Delphi, обеспечивающие разработку многозвенных приложений, объединены общим названием DataSnap.

Примечание

В предыдущих версиях Delphi (Delphi 4 и 5) эти компоненты объединялись под названием MIDAS (Multi-tier Distributed Applications Services — сервисы многозвенных распределенных приложений).

Палитра компонентов Delphi содержит специальную страницу **DataSnap**, на которой доступно большинство рассматриваемых в главах этой части ком-

понентов. Однако при разработке многозвенных приложений нам понадобятся и многие другие компоненты, которым также уделено достаточное внимание.

В этой главе рассматриваются следующие вопросы:

- структура многозвенных приложений;
- механизм удаленного доступа к данным DataSnap;
- удаленные модули удаленных данных;
- компоненты-провайдеры;
- транспортные компоненты удаленных соединений DataSnap;
- вспомогательные компоненты — брокеры соединений.

Структура многозвенного приложения в Delphi

Многозвенная архитектура приложений баз данных вызвана к жизни необходимостью обрабатывать на стороне сервера запросы от большого числа удаленных клиентов. Казалось бы, с этой задачей вполне могут справиться и приложения клиент/сервер, основные элементы которых представлены в *части III*.

Однако в этом случае при большом числе клиентов вся вычислительная нагрузка ложится на сервер БД, который обладает довольно скудным набором средств для реализации сложной бизнес-логики (хранимые процедуры, триггеры, просмотры и т. д.). И разработчики вынуждены существенно усложнять программный код клиентского ПО, а это крайне нежелательно при наличии большого числа удаленных клиентских компьютеров. Ведь с усложнением клиентского ПО возрастает вероятность ошибок и усложняется его обслуживание.

Многозвенная архитектура приложений БД призвана исправить перечисленные недостатки.

Итак, в рамках этой архитектуры *"тонкие" клиенты* представляют собой простейшие приложения, обеспечивающие лишь передачу данных, их локальное кэширование, представление средствами пользовательского интерфейса, редактирование и простейшую обработку.

Клиентские приложения обращаются не к серверу БД напрямую, а к специализированному ПО промежуточного слоя. Это может быть и одно звено (простейшая трехзвенная модель) и более сложная структура.

ПО промежуточного слоя называется *сервером приложений*, принимает запросы клиентов, обрабатывает их в соответствии с запрограммированными

правилами бизнес-логики, при необходимости преобразует в форму, удобную для сервера БД и отправляет серверу.

Сервер БД выполняет полученные запросы и отправляет результаты серверу приложений, который адресует данные клиентам.

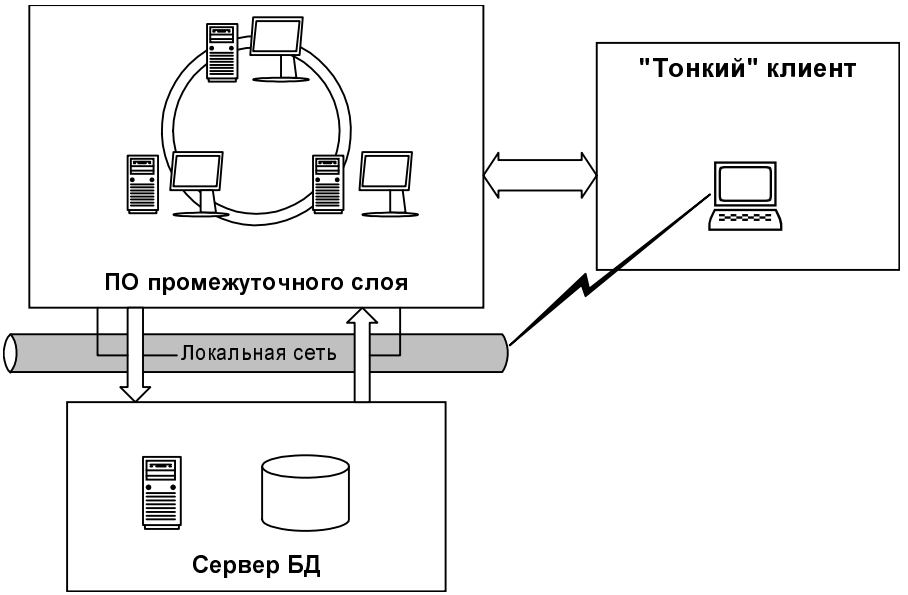


Рис. 20.1. Многозвенная архитектура приложений БД

Таким образом, многозвенное приложение БД состоит из (рис. 20.1):

- "тонких" клиентских приложений, обеспечивающих лишь передачу, представление, редактирование и простейшую обработку данных;
- одного или нескольких звеньев ПО промежуточного слоя (сервер приложений), которые могут функционировать как на одном компьютере, так и распределенно — в локальной сети;
- сервера БД (Oralce, Sybase, MS SQL, InterBase и т. д.), поддерживающего функционирование базы данных и обрабатывающего запросы.

Более простая трехзвенная модель содержит следующие элементы:

- "тонкие" клиенты;
- сервер приложений;
- сервер БД.

Далее мы будем рассматривать именно трехзвенную модель. В среде разработки Delphi имеется набор инструментов и компонентов для создания кли-

ентского ПО и ПО промежуточного слоя. Серверная часть — сервер приложений описывается в *гл. 21*, вопросы создания клиентского ПО — в *гл. 22*.

Сервер приложений взаимодействует с сервером БД, используя одну из технологий доступа к данным, реализованным в Delphi *(см. часть IV)*. Это технологии ADO, BDE, InterBase Express и dbExpress. Разработчик может выбрать наиболее подходящую, исходя из поставленной задачи и параметров сервера БД.

Удаленные клиентские приложения создаются с использованием специального набора компонентов, объединенных общим названием DataSnap. Эти компоненты инкапсулируют стандартные транспорты (DCOM, HTTP, сокеты) и обеспечивают соединение удаленного клиентского приложения с сервером приложения. Также компоненты DataSnap обеспечивают доступ клиента к функциям сервера приложений за счет использования интерфейса `IAppServer` *(см. гл. 21)*.

Важную роль при разработке клиентских приложений играет компонент, инкапсулирующий клиентский набор данных. Его реализации также зависят от технологий доступа к данным и рассматриваются в *гл. 22*.

Наряду с перечисленными выше преимуществами, наличие дополнительного звена — сервера приложений — дает некоторые дополнительные бонусы, которые могут быть весьма существенным подспорьем с точки зрения повышения надежности и эффективности системы.

Так как зачастую клиентские компьютеры — это достаточно слабые машины, реализация сложной бизнес-логики на сторону сервера позволяет существенно повысить быстродействие системы в целом. И не только за счет более мощной техники, но и за счет оптимизации выполнения однородных запросов пользователей.

Например, при чрезмерной загрузке сервера, сервер приложений может самостоятельно обрабатывать запросы пользователей (ставить их в очередь или отменять) без дополнительной загрузки сервера БД.

Наличие сервера приложений повышает безопасность системы, т. к. вы можете организовать здесь авторизацию пользователей, да и любые другие функции безопасности без прямого доступа к данным.

Кроме того, вы легко сможете использовать защищенные каналы передачи данных, например HTTPS.

Трехзвенное приложение в Delphi

Теперь рассмотрим составные части трехзвенного распределенного приложения в Delphi (рис. 20.2). Как говорилось выше, в Delphi целесообразно разрабатывать клиентскую часть трехзвенного приложения и ПО промежуточного слоя — сервер приложений.

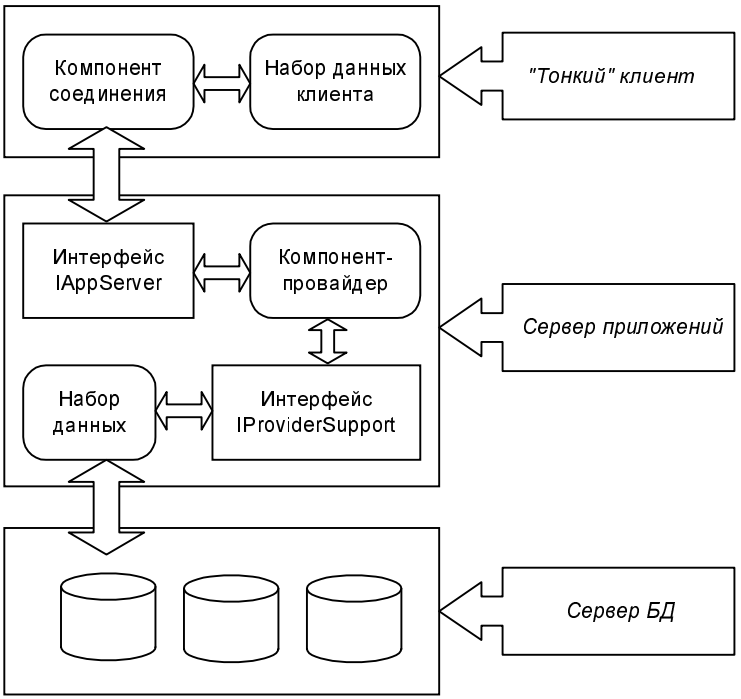


Рис. 20.2. Схема трехзвенного распределенного приложения

Части трехзвенных приложений разрабатываются с использованием компонентов DataSnap, а также некоторых других специализированных компонентов, в основном обеспечивающих функционирование клиента. Для доступа к данным применяется одна из четырех технологий, реализованных в Delphi *(см. часть IV)*.

Примечание

Разработку трехзвенных приложений целесообразно вести, используя в среде разработки группу проектов вместо одиночных проектов. Для этого используется утилита Project Manager (меню **View | Project Manager**).

Для передачи данных между сервером приложений и клиентами используется интерфейс `IAppServer`, предоставляемый удаленным модулем данных сервера приложений. Этот интерфейс используют компоненты-провайдеры `TDataSetProvider` на стороне сервера и компоненты `TClientDataSet` на стороне клиента.

Сервер приложений

Сервер приложений инкапсулирует большую часть бизнес-логики распределенного приложения и обеспечивает доступ клиентов к базе данных.

Основной частью сервера приложений является *удаленный модуль данных*.

Во-первых, подобно обычному модулю данных *(см. гл. 11)* он является платформой для размещения невизуальных компонентов доступа к данным и компонентов-провайдеров. Размещенные на нем компоненты соединений, транзакций и компоненты, инкапсулирующие наборы данных, обеспечивают трехзвенное приложение связью с сервером БД. Это могут быть наборы компонентов для технологий ADO, BDE, InterBase Express, dbExpress.

Во вторых, удаленный модуль данных реализует основные функции сервера приложений на основе предоставления клиентам интерфейса `IAppServer` или его потомка. Для этого удаленный модуль данных должен содержать необходимое число компонентов-провайдеров `TDataSetProvider`. Эти компоненты передают пакеты данных клиентскому приложению, а точнее компонентам `TClientDataSet`, а также обеспечивают доступ к методам интерфейса.

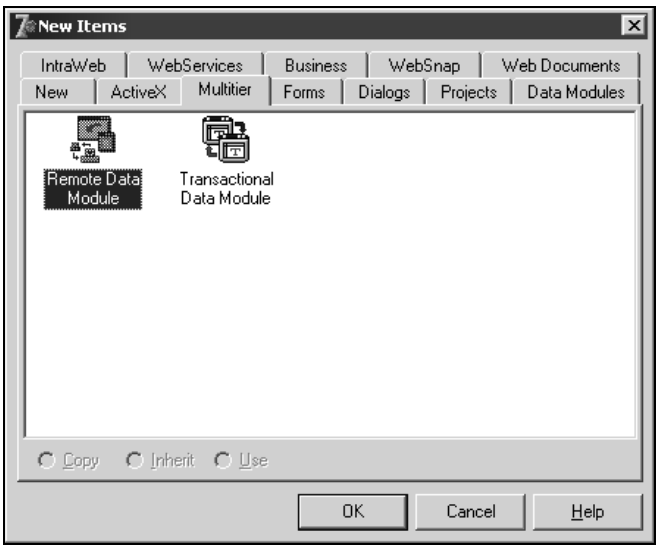


Рис. 20.3. Выбор удаленных модулей данных в Репозитории Delphi

В состав Delphi входят удаленные модули данных. Для их создания используйте страницы **Multitier**, **WebSnap** и **WebServices** Репозитория Delphi (рис. 20.3).

- **Remote Data Module** — удаленный модуль данных, инкапсулирующий сервер Автоматизации. Используется для организации соединений через DCOM, HTTP, сокеты *(см. гл. 21)*.
- **Transactional Data Module** — удаленный модуль данных, инкапсулирующий сервер MTS (Microsoft Transaction Server).

- **SOAP Server Data Module** — удаленный модуль данных, инкапсулирующий сервер SOAP (Simple Object Access Protocol).
- **WebSnap Data Module** — удаленный модуль данных, использующий Web-службы и Web-браузер в качестве сервера.

Помимо удаленного модуля данных неотъемлемой частью сервера приложений являются компоненты-провайдеры `TDataSetProvider`. С каждым компонентом, инкапсулирующим набор данных, предназначенным для передачи клиенту, в модуле данных должен быть связан компонент-провайдер.

Клиентское приложение

Клиентское приложение в трехзвенной модели должно обладать лишь минимально необходимым набором функций, делегируя большинство операций по обработке данных серверу приложений.

В первую очередь удаленное клиентское приложение должно обеспечить соединение с сервером приложений. Для этого используются компоненты соединений DataSnap:

- `TDCOMConnection` — использует DCOM;
- `TSocketConnection` — использует сокеты Windows;
- `TWebConnection` — использует HTTP.

Компоненты соединения DataSnap предоставляют интерфейс `IAppServer`, используемый компонентами-провайдерами на стороне сервера и компонентами `TClientDataSet` на стороне клиента для передачи пакетов данных.

Для работы с наборами данных используются компоненты `TClientDataSet`, работающие в режиме кэширования данных.

Для представления данных и создания пользовательского интерфейса в клиентском ПО применяются стандартные компоненты со страницы **Data Controls** Палитры компонентов.

Подробнее о разработке клиентского ПО для распределенных многозвенных приложений БД рассказывается в *гл. 22*.

Механизм удаленного доступа к данным DataSnap

Для передачи пакетов данных между компонентом-провайдером и клиентским набором данных (см. рис. 20.2) (между клиентом и сервером) должен существовать некий транспортный канал, обеспечивающий физическую передачу данных. Для этого могут использоваться разнообразные транспортные протоколы, поддерживаемые операционной системой.

Различные типы соединений, позволяющие настроить транспорт и начать передачу и прием данных, инкапсулированы в нескольких компонентах DataSnap. Для создания соединения с тем или иным транспортным протоколом разработчику достаточно перенести соответствующий компонент на форму и правильно настроить несколько свойств. Ниже рассматриваются варианты настройки транспортных протоколов для компонентов, использующих DCOM, сокеты TCP/IP, http.

Компонент *TDCOMConnection*

Компонент `TDCOMConnection` предоставляет транспорт на основе технологии Distributed COM и применяется в основном для организации транспорта в рамках локальной сети.

Для настройки соединения DCOM в первую очередь необходимо задать имя компьютера, на котором функционирует сервер приложений. Для компонента `TDCOMConnection` это должен быть зарегистрированный сервер Автоматизации. Имя компьютера задается свойством

```
property ComputerName: string;
```

Если оно задано правильно, в списке свойства

```
property ServerName: string;
```

в Инспекторе объектов можно выбрать один из доступных серверов.

При выборе сервера также автоматически заполняется свойство

```
property ServerGUID: string;
```

Причем для успешного соединения клиента с сервером приложений оба свойства должны быть заданы в обязательном порядке. Только имя сервера или только его GUID не обеспечат правильный доступ к удаленному объекту COM.

Открытие и закрытие соединения осуществляется свойством

```
property Connected: Boolean;
```

или методами

```
procedure Open;
procedure Close;
```

соответственно.

Для организации передачи данных между клиентом и сервером компонент `TDCOMConnection` предоставляет интерфейс `IAppServer`

```
property AppServer: Variant;
```

который также может быть получен методом

```
function GetServer: IAppServer; override;
```

Свойство

```
property ObjectBroker: TCustomObjectBroker;
```

позволяет использовать экземпляр компонента `TSimpleObjectBroker` для получения списка доступных серверов по время выполнения *(см. ниже)*.

Методы-обработчики компонента `TDCOMConnection` представлены в табл. 20.1.

Таблица 20.1. *Методы-обработчики событий компонента* `TDCOMConnection`

Объявление	Описание
`property AfterConnect: TNotifyEvent;`	Вызывается после установления соединения
`property AfterDisconnect: TNotifyEvent;`	Вызывается после разрыва соединения
`property BeforeConnect: TNotifyEvent;`	Вызывается перед установлением соединения
`property BeforeDisconnect: TNotifyEvent;`	Вызывается перед разрывом соединения
`type TGetUsernameEvent = procedure(Sender: TObject; var Username: string) of object;` `property OnGetUsername: TGetUsernameEvent;`	Вызывается непосредственно перед появлением диалога удаленной авторизации пользователя. Для этого свойство `LoginPrompt` должно иметь значение `True`. Параметр `Username` может содержать имя пользователя по умолчанию, которое появится в диалоге
`type TLoginEvent = procedure(Sender:TObject; Username, Password: string) of object;` `property OnLogin: TLoginEvent;`	Вызывается после открытия соединения, если свойство `LoginPrompt` имеет значение `True`. Параметры `Username` и `Password` содержат имя пользователя и пароль, введенные при авторизации

Компонент *TSocketConnection*

Компонент `TSocketConnection` обеспечивает соединение клиента с сервером приложений за счет использования сокетов TCP/IP. Для успешного открытия соединения на стороне сервера должен работать сокет-сервер (приложение ScktSrvr.exe, рис. 20.4).

Для успешного соединения свойство

```
property Host: String;
```

должно содержать имя компьютера сервера.

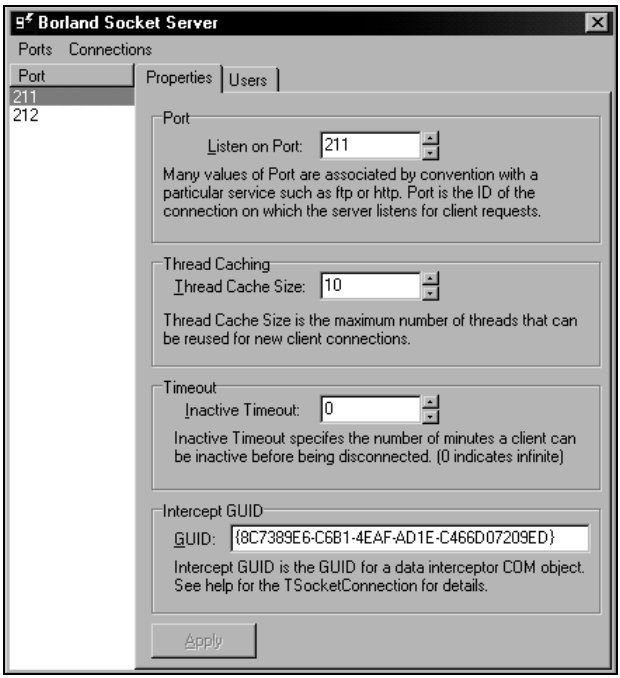


Рис. 20.4. Сокет-сервер ScktSrvr.exe

Дополнительно, свойство

```
property Address: String;
```

должно содержать IP-адрес сервера.

Для открытия соединения должны быть заданы оба этих свойства.

Свойство

```
property Port: Integer;
```

устанавливает номер используемого порта. По умолчанию это порт 211, но разработчик волен изменить порт, например, для использования различными категориями пользователей или для создания защищенного канала.

После правильного выбора компьютера в списке свойства

```
property ServerName: string;
```

в Инспекторе объектов появляется перечень доступных серверов Автоматизации. И после выбора сервера свойство

```
property ServerGUID: string;
```

которое содержит имя компьютера `GUID` зарегистрированного сервера, задается автоматически, хотя его можно задать и вручную.

Метод

```
function GetServerList: OleVariant; virtual;
```

возвращает список зарегистрированных серверов Автоматизации.

Открытие и закрытие соединения осуществляется свойством

```
property Connected: Boolean;
```

или методами

```
procedure Open;
procedure Close;
```

соответственно.

Канал сокета TCP/IP может быть зашифрован. Для этого используется свойство

```
property InterceptName: string;
```

содержащее программный идентификатор объекта COM, обеспечивающего шифрование/дешифрование данных в канале, и свойство

```
property InterceptGUID: string;
```

содержащее имя компьютера `GUID` этого объекта.

Этот объект COM перехватывает данные в канале и осуществляет их обработку, предусмотренную собственным программным кодом. Это может быть шифрование, сжатие, обработка шумов и т. д.

Примечание

Создание объекта COM, обеспечивающего дополнительную обработку данных в канале, ложится на плечи разработчика. Объект-перехватчик должен поддерживать стандартный интерфейс `IDataIntercept`.

Естественно, на стороне сервера должен быть зарегистрирован объект COM, выполняющий обратную операцию. Для этого также используется сокет-сервер (рис. 20.5). Строка **Interceptor** на странице должна содержать имя компьютера `GUID` объекта-перехватчика COM.

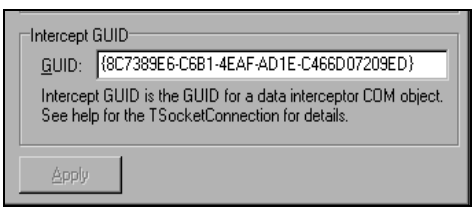


Рис. 20.5. Регистрация объекта-перехватчика COM в сокет-сервере

Метод

```
function GetInterceptorList: OleVariant; virtual;
```

возвращает список зарегистрированных на сервере объектов-перехватчиков.

Для организации передачи данных между клиентом и сервером компонент `TSocketConnection` предоставляет интерфейс `IAppServer`

```
property AppServer: Variant;
```

который также может быть получен методом

```
function GetServer: IAppServer; override;
```

Свойство

```
property ObjectBroker: TCustomObjectBroker;
```

позволяет использовать экземпляр компонента `TSimpleObjectBroker` для получения списка доступных серверов во время выполнения *(см. ниже)*.

Методы-обработчики событий компонента `TSocketConnection` полностью совпадают с методами-обработчиками компонента `TDCOMConnection` (см. табл. 20.1).

Компонент *TWebConnection*

Компонент `TWebConnection` предоставляет клиенту соединение на основе транспорта HTTP. Для работы компонента на клиентском компьютере должна быть зарегистрирована библиотека wininet.dll. Обычно это не требует специальных усилий, т. к. этот файл уже имеется в системной папке Windows, если на компьютере установлен Internet Explorer.

На компьютере сервера должен быть инсталлирован Internet Information Server версии не ниже 4.0 или Netscape Enterprise версии не ниже 3.6. Перечисленное ПО обеспечивает доступ компонента `TWebConnection` к динамической библиотеке HTTPsrvr.dll, которая также должна находиться на сервере.

Например, если файл HTTPsrvr.dll расположен в папке Scripts IIS 4.0 на Web-сервере **www.someserver.com**, то свойство

```
property URL: string;
```

должно содержать следующее значение:

http://someserver.com/scripts/httpsrvr.dll

Если URL задан верно и сервер настроен правильно, то в списке свойства

```
property ServerName: string;
```

в Инспекторе объектов появляется перечень зарегистрированных серверов приложений. Имя одного из них должно содержаться в свойстве `ServerName`.

После выбора имени сервера в свойстве

```
property ServerGUID: string;
```

автоматически появляется GUID сервера.

Свойства

```
property UserName: string;
```

и

```
property Password: string;
```

при необходимости могут содержать имя и пароль пользователя, которые будут использованы при авторизации.

Свойство

```
property Proxy: string;
```

содержит имя используемого прокси-сервера.

В заголовок сообщений HTTP можно поместить имя приложения. Для этого используется свойство

```
property Agent: string;
```

Соединение открывается и закрывается при помощи свойства

```
property Connected: Boolean;
```

Аналогичные операции выполняют методы

```
procedure Open;
procedure Close;
```

Доступ к интерфейсу `IAppServer` предоставляет свойство

```
property AppServer: Variant;
```

или метод

```
function GetServer: IAppServer; override;
```

Список доступных соединению серверов приложений возвращает метод

```
function GetServerList: OleVariant; virtual;
```

Свойство

```
property ObjectBroker: TCustomObjectBroker;
```

позволяет использовать экземпляр компонента `TSimpleObjectBroker` для получения списка доступных серверов во время выполнения *(см. ниже)*.

Методы-обработчики событий компонента `TWebConnection` полностью совпадают с методами-обработчиками компонента `TDCOMConnection` (см. табл. 20.1).

Провайдеры данных

Компонент-провайдер TDataSetProvider представляет собой мост между набором данных сервера приложений и клиентским набором данных. Он обеспечивает формирование и передачу пакетов данных клиентскому приложению и прием от него сделанных изменений (см. рис. 20.2).

Все необходимые операции компонент выполняет автоматически. Разработчику необходимо лишь разместить компонент TDataSetProvider и связать его с набором данных сервера приложений. Для этого предназначено свойство

```
property DataSet: TDataSet;
```

Если соединение в клиентском приложении настроено правильно *(см. выше)*, то в списке выбора свойства ProviderName компонента TClientDataSet в Инспекторе объектов появляются имена всех компонентов-провайдеров сервера приложений. Если связать клиентский набор данных с компонентом-провайдером, а затем открыть его, в клиентский набор данных будут переданы записи из набора данных сервера приложений, указанного в свойстве DataSet компонента-провайдера TDataSetProvider.

Компонент также содержит свойства, помогающие настроить процесс обмена данными.

Свойство

```
property ResolveToDataSet: Boolean;
```

управляет передачей данных от клиента серверу БД. Если оно имеет значение True, все изменения передаются в набор данных сервера приложений, заданный свойством DataSet. Иначе изменения направляются напрямую серверу БД. Если сервер приложений не должен отображать сделанные клиентом изменения, то свойству ResolveToDataSet можно присвоить значение False, что ускорит работу приложения.

Свойство

```
property Constraints: Boolean;
```

управляет передачей ограничений серверного набора данных клиентскому. Если свойство имеет значение True, ограничения передаются.

Свойство

```
property Exported: Boolean;
```

позволяет использовать в клиентском наборе данных интерфейс IAppServer. Для этого свойство должно иметь значение True.

Параметры компонента-провайдера задаются свойством

```
type
  TProviderOption = (poFetchBlobsOnDemand, poFetchDetailsOnDemand,
                     poIncFieldProps, poCascadeDeletes, poCascadeUpdates,
                     poReadOnly, poAllowMultiRecordUpdates,
                     poDisableInserts, poDisableEdits, poDisableDeletes,
                     poNoReset, poAutoRefresh, poPropogateChanges,
                     poAllowCommandText, poRetainServerOrder);
  TProviderOptions = set of TProviderOption;
```

Набор параметров свойства задается присвоением элементам значения `True`.

```
property Options: TProviderOptions;
```

`poFetchBlobsOnDemand` — включает передачу в клиентский набор данных значений полей типа BLOB. По умолчанию эта возможность отключена для ускорения работы;

`poFetchDetailsOnDemand` — включает передачу в клиентский набор данных подчиненных записей для отношения "один-ко-многим". По умолчанию эта возможность отключена для ускорения работы;

`poIncFieldProps` — включает передачу в клиентский набор данных нескольких свойств для объектов полей: `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, `DisplayValues`;

`poCascadeDeletes` — включает автоматическое удаление подчиненных записей в отношении "один-ко-многим" на стороне сервера, если главная запись была удалена в клиентском наборе данных;

`poCascadeUpdates` — включает автоматическое обновление подчиненных записей в отношении "один-ко-многим" на стороне сервера, если главная запись была изменена в клиентском наборе данных;

`poReadOnly` — включает режим "только для чтения" для набора данных сервера;

`poAllowMultiRecordUpdates` — включает режим внесения изменений сразу в несколько записей одновременно. Иначе все записи изменяются последовательно, одна за одной;

`poDisableInserts` — запрещает клиенту вносить в набор данных сервера новые записи;

`poDisableEdits` — запрещает клиенту вносить в набор данных сервера изменения;

`poDisableDeletes` — запрещает клиенту удалять записи в наборе данных сервера;

poNoReset — запрещает обновление набора данных сервера перед передачей записей клиенту (перед вызовом метода AS_GetRecords интерфейса IAppServer);

poAutoRefresh — включает автоматическое обновление записей клиентского набора данных. По умолчанию эта возможность отключена для ускорения работы;

poPropogateChanges — если в методах-обработчиках BeforeUpdateRecord или AfterUpdateRecord клиентского набора данных были сделаны дополнительные изменения, то после их записи в наборе данных сервера, изменения снова направляются клиенту для обновления записи. Во включенном состоянии эта возможность позволяет полностью контролировать сохранение изменений на сервере;

poAllowCommandText — позволяет изменять текст запроса SQL, имена хранимых процедур или таблиц в компоненте набора данных на сервере приложений;

poRetainServerOrder — включает запрет на изменение порядка сортировки записей клиентом. Если этот параметр отключить, возможны ошибки отображения набора данных, проявляющиеся в появлении двойных записей.

Методы-обработчики компонента-провайдера данных представлены в табл. 20.2.

Таблица 20.2. *Методы-обработчики событий компонента* `TDataSetProvider`

Объявление	Описание
`property AfterApplyUpdates: TRemoteEvent;`	Вызывается после сохранения изменений, переданных от клиента, в наборе данных сервера
`property AfterExecute: TRemoteEvent;`	Вызывается после выполнения запроса SQL или хранимой процедуры на сервере
`property AfterGetParams: TRemoteEvent;`	Вызывается после того, как компонент-провайдер сформировал набор параметров набора данных сервера для их передачи клиенту
`property AfterGetRecords: TRemoteEvent;`	Вызывается после того, как компонент-провайдер сформировал пакет данных для передачи набора данных сервера клиенту
`property AfterRowRequest: TRemoteEvent;`	Вызывается после обновления текущей записи клиента компонентом-провайдером
`property AfterUpdateRecord: TAfterUpdateRecordEvent;`	Вызывается сразу после обновления единичной записи на сервере
`property BeforeApplyUpdates: TRemoteEvent;`	Вызывается перед сохранением изменений, переданных от клиента, в наборе данных сервера

Таблица 20.2 *(окончание)*

Объявление	Описание
`property BeforeExecute: TRemoteEvent;`	Вызывается перед выполнением запроса SQL или хранимой процедуры на сервере
`property BeforeGetParams: TRemoteEvent;`	Вызывается перед тем, как компонент-провайдер сформировал набор параметров набора данных сервера для их передачи клиенту
`property BeforeGetRecords: TRemoteEvent;`	Вызывается перед тем, как компонент-провайдер сформировал пакет данных для передачи набора данных сервера клиенту
`property BeforeRowRequest: TRemoteEvent;`	Вызывается перед обновлением текущей записи клиента компонентом-провайдером
`property BeforeUpdateRecord: TBeforeUpdateRecordEvent;`	Вызывается непосредственно перед обновлением единичной записи на сервере
`property OnDataRequest: TDataRequestEvent;`	Вызывается при обработке запроса на получение данных клиентом
`property OnGetData: TProviderDataEvent;`	Вызывается после получения данных от набора данных сервера, но перед их отправкой клиенту
`property OnGetDataSetProperties: TGetDSProps;`	Вызывается при создании структуры параметров набора данных сервера для их передачи клиенту
`property OnGetTableName: TGetTableNameEvent;`	Вызывается при получении компонентом-провайдером имени таблицы, подлежащей обновлению
`property OnUpdateData: TProviderDataEvent;`	Вызывается при сохранении изменений в наборе данных сервера
`property OnUpdateError: TResolverErrorEvent;`	Вызывается при возникновении ошибки сохранения изменений в наборе данных сервера

Вспомогательные компоненты — брокеры соединений

В состав компонентов DataSnap входит ряд дополнительных компонентов, облегчающих работу с соединениями удаленных клиентов с сервером приложений. Рассмотрим их.

Компонент *TSimpleObjectBroker*

Компонент `TSimpleObjectBroker` инкапсулирует список серверов, доступных для клиентов данного многозвенного распределенного приложения. Список серверов создается на этапе разработки. При необходимости (отключение

сервера, его перегрузка и т. д.) компонент соединения клиентского ПО может использовать один из запасных серверов из списка компонента `TSimpleObjectBroker` непосредственно во время выполнения.

Для этого необходимо заполнить список серверов компонента `TSimpleObjectBroker` и указать ссылку на него в свойстве `ObjectBroker` компонента соединения *(см. выше)*. И тогда при "переоткрытии" соединения имя сервера будет запрашиваться из списка компонента `TSimpleObjectBroker`.

Список серверов задается свойством

```
property Servers: TServerCollection;
```

На этапе разработки список серверов заполняется специализированным редактором (рис. 20.6), который вызывается при щелчке на кнопке свойства в Инспекторе объектов.

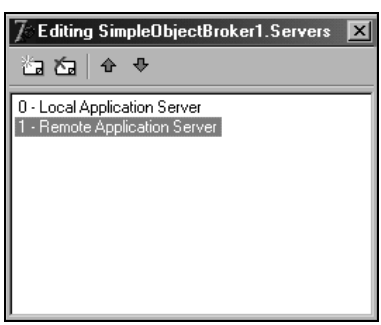


Рис. 20.6. Редактор списка серверов компонента `TSimpleObjectBroker`

Свойство `Servers` представляет собой коллекцию *(см. гл. 7)* объектов класса `TServerItem`. Этот класс имеет несколько свойств, позволяющих описать основные параметры сервера (табл. 20.3). При использовании в соединении значения этих свойств подставляются в соответствующие свойства компонента соединения.

Таблица 20.3. *Свойства класса* `TServerItem`

Объявление	Описание
`property ComputerName: string;`	Имя компьютера, на котором функционирует сервер
`property DisplayName: String;`	Содержит имя сервера для представления в списке серверов
`property Enabled: Boolean;`	Управляет доступностью записи о сервере для выбора при подключении. При значении `True` компоненты соединений могут использовать данную запись списка для подключения

Таблица 20.3 *(окончание)*

Объявление	Описание
`property HasFailed: Boolean;`	После неудачной попытки использовать данную запись списка при подключении свойству присваивается значение `True` и в дальнейшем эта запись не используется
`property Port: Integer;`	Содержит номер порта, используемого при подключении к серверу

Помимо списка серверов компонент имеет лишь несколько вспомогательных свойств и методов.

Метод

```
function GetComputerForGUID(GUID: TGUID): string; override;
```

возвращает имя компьютера, на котором зарегистрирован сервер с `GUID`, заданным параметром.

Метод

```
function GetComputerForProgID(const ProgID): string; override;
```

возвращает имя компьютера, на котором зарегистрирован сервер с именем, заданным параметром `ProgID`.

Свойство

```
property LoadBalanced: Boolean;
```

управляет выбором сервера из списка. При значении `True` запись о сервере выбирается случайным образом, иначе для соединения предлагается первая доступная запись о сервере.

Компонент *TLocalConnection*

Компонент `TLocalConnection` используется локально для получения доступа к существующим компонентам-провайдерам.

Свойство

```
property Providers[const ProviderName: string]: TCustomProvider;
```

содержит ссылки на все компоненты-провайдеры, размещенные с компонентом `TLocalConnection` на одной форме. Индексация в списке осуществляется по имени компонента-провайдера.

Общее число компонентов-провайдеров в списке возвращает свойство

```
property ProviderCount: Integer;
```

Кроме этого, при помощи компонента `TLocalConnection` можно получить доступ к интерфейсу `IAppServer` локально. Для этого используется свойство

```
property AppServer: IAppServer;
```

или метод

```
function GetServer: IAppServer; override;
```

Компонент *TSharedConnection*

Если интерфейс `IAppServer` удаленного модуля данных имеет метод, возвращающий ссылку на аналогичный интерфейс другого удаленного модуля данных, то первый модуль называется *главным*, а второй — *дочерним (см. гл. 21)*. Компонент `TSharedConnection` используется для соединения клиентского приложения с дочерним удаленным модулем данных сервера приложений.

Свойство

```
property ParentConnection: TDispatchConnection;
```

должно содержать ссылку на компонент соединения с главным удаленным модулем данных сервера приложений. Дочерний удаленный модуль данных определяется свойством

```
property ChildName: string;
```

которое должно содержать его имя. Если интерфейс главного удаленного модуля данных настроен правильно, то в списке выбора свойства в Инспекторе объектов появляются имена всех дочерних удаленных модулей данных.

Интерфейс `IAppServer` дочернего удаленного модуля данных возвращает свойство

```
property AppServer: Variant;
```

или метод

```
function GetServer: IAppServer; override;
```

Методы-обработчики компонента `TSharedConnection` унаследованы от класса предка `TCustomConnection` (см. табл. 20.1).

Компонент *TConnectionBroker*

Компонент `TConnectionBroker` обеспечивает централизованное управление соединением клиентских наборов данных с сервером приложений. Для этого свойство `ConnectionBroker` клиентских наборов данных должно ссылаться на экземпляр компонента `TConnectionBroker`. Тогда для изменения соединения (например, при переходе с транспорта HTTP на сокеты TCP/IP) нет

необходимости изменять значение свойства `RemoteServer` всех компонентов `TClientDataSet`, а достаточно изменить свойство

```
property Connection: TCustomRemoteServer;
```

компонента `TConnectionBroker`.

Доступ к интерфейсу `IAppServer` обеспечивает свойство

```
property AppServer: Variant;
```

или метод

```
function GetServer: IAppServer; override;
```

Методы-обработчики компонента `TConnectionBroker` полностью соответствуют табл. 20.1.

Резюме

Многозвенные распределенные приложения обеспечивают эффективное взаимодействие большого числа удаленных "тонких" клиентов с сервером БД при помощи ПО промежуточного слоя. Наиболее распространенной моделью является трехзвенная модель, где ПО промежуточного слоя состоит только из сервера приложений.

В Delphi для создания трехзвенных распределенных приложений используются компоненты DataSnap и удаленные модули данных. Все эти инструменты реализованы для различных типов транспортных протоколов.

Также в трехзвенных приложениях применяются компоненты-провайдеры `TDataSetProvider` и компоненты `TClientDataSet`, инкапсулирующие наборы данных на клиентской стороне.

ГЛАВА 21

Сервер приложения

Многозвенные распределенные приложения обеспечивают эффективный доступ удаленных клиентов к базе данных, так как в них для управления доступом к данным применяется специализированное ПО промежуточного слоя. В наиболее распространенной схеме — трехзвенном приложении — это сервер приложения, который выполняет следующие функции:

- обеспечивает авторизацию пользователей;
- принимает и передает запросы пользователей и пакеты данных;
- регулирует доступ клиентских запросов к серверу БД, балансируя нагрузку сервера БД;
- может содержать часть бизнес-логики распределенного приложения, обеспечивая существование "тонких" клиентов.

Delphi обеспечивает разработку серверов приложений на основе использования ряда технологий:

- Web;
- Автоматизация;
- MTS;
- SOAP.

В этой главе рассматриваются следующие вопросы:

- программные элементы сервера приложения Delphi;
- структура сервера приложения;
- типы удаленных модулей данных;
- создание и настройка удаленных модулей данных;
- роль компонентов-провайдеров в передаче данных клиентам;

- методы интерфейса `IAppServer`;
- регистрация сервера приложения.

Структура сервера приложения

Итак, сервер приложения — это ПО промежуточного слоя трехзвенного распределенного приложения (см. рис. 20.2). Его основой является удаленный модуль данных. В Delphi предусмотрено использование удаленных модулей данных пяти типов *(см. ниже)*.

Далее в этой главе мы детально рассмотрим вопросы использования удаленных модулей данных, инкапсулирующих функции серверов Автоматизации. Другие типы удаленных модулей данных рассматриваются в следующих частях книги.

Каждый удаленный модуль данных инкапсулирует интерфейс `IAppServer`, методы которого используются в механизме удаленного доступа клиентов к серверу БД *(см. гл. 20)*.

Для обмена данными с сервером БД модуль данных может содержать некоторое количество компонентов доступа к данным (компонентов соединений и компонентов, инкапсулирующих набор данных).

Для обеспечения передачи данных клиентам удаленный модуль данных обязательно должен содержать необходимое количество компонентов `TDataSetProvider`, каждый из которых должен быть связан с соответствующим набором данных.

Внимание!

Обмен данными сервера приложения с клиентами обеспечивает динамическая библиотека MIDAS.DLL, которая должна быть зарегистрирована на компьютере сервера приложения.

Для создания нового сервера приложения достаточно выполнить несколько простых операций.

1. Создать новый проект, выбрав в качестве типа проекта обычное приложение (пункт меню **File | New | Application**) и сохранить его.
2. В зависимости от используемой технологии, выбрать из Репозитория Delphi необходимый тип удаленного модуля данных (см. рис. 20.3). Удаленные модули данных располагаются на страницах **Multitier**, **WebSnap** и **WebServices**.
3. Настроить параметры создаваемого удаленного модуля данных *(см. ниже)*.
4. Разместить в удаленном модуле данных компоненты доступа к данным и настроить их. Здесь разработчик может выбрать один из имеющихся

наборов компонентов *(см. часть IV)* в зависимости от используемого сервера БД и требуемых характеристик создаваемого приложения.

5. Разместить в удаленном модуле данных необходимое число компонентов `TDataSetProvider` и связать их с компонентами, инкапсулирующими наборы данных.
6. При необходимости создать для потомка интерфейса `IAppServer`, используемого в удаленном модуле данных, дополнительные методы. Для этого создается новая библиотека типов *(см. ниже)*.
7. Скомпилировать проект и создать исполняемый файл сервера приложения.
8. Зарегистрировать сервер приложения и при необходимости настроить дополнительное ПО.

Весь механизм удаленного доступа, инкапсулированный в удаленных модулях данных и компонентах-провайдерах, работает автоматически, без создания разработчиком дополнительного программного кода.

Далее в этой главе на простом примере рассматриваются все перечисленные этапы создания сервера приложения.

Интерфейс *IAppServer*

Интерфейс `IAppServer` является основной механизма удаленного доступа клиентских приложений к серверу приложения. Набор данных клиента использует его для общения с компонентом-провайдером на сервере приложения. Наборы данных клиента получают экземпляр `IAppServer` от компонента соединения в клиентском приложении (см. рис. 20.2).

При создании удаленных модулей данных *(см. ниже)* каждому такому модулю ставится в соответствие вновь создаваемый интерфейс, предком которого является интерфейс `IAppServer`.

Разработчик может добавить к новому интерфейсу собственные методы, которые, благодаря возможностям механизма удаленного доступа многозвенных приложений, становятся доступны приложению-клиенту.

Свойство

```
property AppServer: Variant;
```

в клиентском приложении имеется как в компонентах удаленного соединения, так и клиентском наборе данных.

По умолчанию интерфейс является несохраняющим состояние (stateless). Это означает, что вызовы методов интерфейса независимы и не привязаны

к предыдущему вызову. Поэтому интерфейс `IAppServer` не имеет свойств, которые бы хранили информацию о состоянии между вызовами.

Обычно разработчику ни к чему использовать методы интерфейса напрямую, однако его значение для многозвенных приложений трудно переоценить. И при детальной работе с механизмом удаленного доступа интерфейс понадобится так или иначе.

Методы интерфейса `IAppServer` представлены в табл. 21.1

Таблица 21.1. *Методы интерфейса* `IAppServer`

Объявление	Описание
`function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant): OleVariant; safecall;`	Передает изменения, полученные от клиентского набора данных, компоненту-провайдеру, определяемому параметром `ProviderName`. Изменения содержатся в параметре `Delta`. Параметр `MaxErrors` задает максимальное число ошибок, пропускаемых при сохранении данных перед прерыванием операции. Реальное число возникших ошибок возвращается параметром `ErrorCount`. Параметр `OwnerData` содержит дополнительную информацию, передаваемую между клиентом и сервером (например, значения параметров методов-обработчиков). Функция возвращает пакет данных, содержащий все записи, которые не были сохранены в базе данных по какой-либо причине
`function AS_DataRequest(const ProviderName: WideString; Data: OleVariant): OleVariant; safecall;`	Генерирует событие `OnDataRequest` для указанного провайдера `ProviderName`
`procedure AS_Execute(const ProviderName: WideString; const CommandText: WideString; var Params: OleVariant; var OwnerData: OleVariant); safecall;`	Выполняет запрос или хранимую процедуру, определяемые параметром `CommandText` для провайдера, указанного параметром `ProviderName`. Параметры запроса или хранимой процедуры содержатся в параметре `Params`
`function AS_GetParams(const ProviderName: WideString; var OwnerData: OleVariant): OleVariant; safecall;`	Передает провайдеру `ProviderName` текущие значения параметров клиентского набора данных
`function AS_GetProviderNames: OleVariant; safecall;`	Возвращает список всех доступных провайдеров удаленного модуля данных

Таблица 21.1 *(окончание)*

Объявление	Описание
`function AS_GetRecords(const ProviderName: WideString, Count: Integer; out RecsOut: Integer; Options: Integer; const CommandText: WideString; var Params: OleVariant; var OwnerData:OleVariant): OleVariant; safecall;`	Возвращает пакет данных с записями набора данных сервера, связанного с компонентом-провайдером. Параметр `CommandText` содержит имя таблицы, текст запроса или имя хранимой процедуры, откуда необходимо получить записи. Но он работает только в случае, если для провайдера в параметре `Options` включена опция `poAllowCommandText`. Параметры запроса или процедуры помещаются в параметре `Params`. Параметр задает требуемое число записей, начиная с текущей, если его значение больше нуля. Если параметр равен нулю — возвращаются только метаданные, если он равен –1 — возвращаются все записи. Параметр `RecsOut` возвращает реальное число переданных записей
`function AS_RowRequest(const ProviderName: WideString; Row: OleVariant; RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;`	Возвращает запись набора данных (предоставляемого провайдером `ProviderName`), определяемую параметром `Row`. Параметр `RequestType` содержит значение типа `TfetchOptions`

Большинство методов интерфейса используют параметры `ProviderName` и `OwnerData`. Первый определяет имя компонента-провайдера, а второй содержит набор параметров, передаваемых для использования в методах-обработчиках.

Внимательный читатель обратил внимание, что использование метода `AS_GetRecords` подразумевает сохранение информации при работе интерфейса, т. к. метод возвращает записи, начиная с текущей, хотя интерфейс `IAppServer` имеет тип `stateless`. Поэтому перед использованием метода рекомендуется обновлять набор данных клиента.

Тип

```
TFetchOption = (foRecord, foBlobs, foDetails);
TFetchOptions = set of TFetchOption;
```

используется в параметре `RequestType` метода `AS_RowRequest`.

`foRecord` — возвращает значения полей текущей записи;

`foBlobs` — возвращает значения полей типа BLOB текущей записи;

`foDetails` — возвращает все подчиненные записи вложенных наборов данных для текущей записи.

Интерфейс *IProviderSupport*

Для организации взаимодействия клиентов с сервером БД удаленный модуль данных сервера приложения должен содержать компоненты-провайдеры `TDataSetProvider` *(см. гл. 20)*. При этом используются методы интерфейса `IAppServer`.

Для обмена данными с набором данных на сервере компонент-провайдер применяет интерфейс `IProviderSupport` (см. рис. 20.2), который включен в любой компонент набора данных, произошедший от класса `TDataSet`. В зависимости от используемой технологии доступа к данным каждый компонент, инкапсулирующий набор данных, имеет собственную реализацию методов интерфейса `IProviderSupport`.

Методы интерфейса могут понадобится разработчику только при создании собственных компонентов, инкапсулирующих набор данных и наследующих от класса `TDataSet`.

Удаленные модули данных

Удаленный модуль данных является основой сервера приложения (см. рис. 20.2) для многозвенного распределенного приложения. Во-первых, он выполняет функции обычного модуля данных — на нем можно размещать компоненты доступа к данным. Во-вторых, удаленный модуль данных инкапсулирует интерфейс `IAppServer`, обеспечивая тем самым выполнение функций сервера и обмен данными с удаленными клиентами.

В зависимости от используемой технологии в Delphi можно использовать удаленные модули данных пяти типов.

- **Remote Data Module**. Класс `TRemoteDataModule` инкапсулирует сервер Автоматизации.
- **Transactional Data Module**. Класс `TMTSDataModule` является потомком класса `TRemoteDataModule` и к функциям обычного сервера Автоматизации добавляет возможности MTS.
- **WebSnap Data Module**. Класс `TWebDataModule` создает сервер приложения, использующий возможности Internet-технологий.
- **Soap Server Data Module**. Класс `TSOAPDataModule` инкапсулирует сервер SOAP.
- **CORBA Data Module**. Класс `TCORBADataModule` является потомком класса `TRemoteDataModule` и реализует функции сервера CORBA.

Ниже мы рассмотрим процесс создания сервера приложения на основе удаленного модуля данных `TRemoteDataModule`. Остальные модули данных (за исключением удаленного модуля данных для CORBA) детально рассматриваются далее в этой книге.

Удаленный модуль данных для сервера Автоматизации

Для создания удаленного модуля данных `TRemoteDataModule` используется Репозиторий Delphi (команда **File | New | Other**). Значок класса `TRemoteDataModule` находится на странице **Multitier** (см. рис. 20.3). Перед созданием экземпляра удаленного модуля данных появляется диалоговое окно (рис. 21.1), в котором необходимо предустановить три параметра.

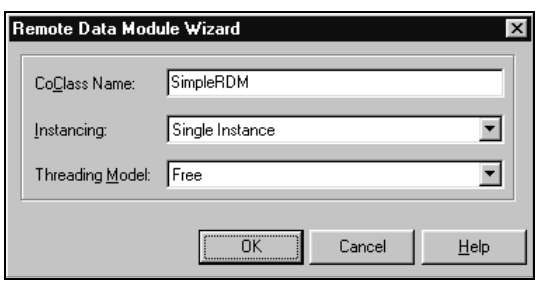


Рис 21.1. Мастер создания удаленного модуля данных `TRemoteDataModule`

Строка **CoClass Name** должна содержать имя нового модуля данных, которое будет также использовано для именования нового класса, создаваемого для поддержки нового модуля данных.

Список **Instancing** позволяет задать способ создания модуля данных.

- **Internal** — модуль данных обеспечивает функционирование лишь внутреннего сервера Автоматизации.
- **Single Instance** — для каждого клиентского соединения создается собственный экземпляр удаленного сервера Автоматизации в собственном процессе.
- **Multiple Instance** — для каждого клиентского соединения создается собственный экземпляр удаленного сервера Автоматизации в одном общем процессе.

Список **Threading Model** задает механизм обработки запросов клиентов.

- **Single** — поток запросов клиентов обрабатывается строго последовательно.
- **Apartment** — модуль данных одновременно обрабатывает один запрос. Однако если DLL для выполнения запросов создает экземпляры COM

объектов, то для запросов могут создаваться отдельные нити, в которых обработка ведется параллельно.

- **Free** — модуль данных может создавать нити для параллельного выполнения запросов.
- **Both** — аналогична модели Free, за исключением того, что все ответы клиентам возвращаются строго один за другим.
- **Neutral** — запросы клиентов могут направляться модулям данных в нескольких нитях одновременно. Используется только для технологии COM+.

При создании нового удаленного модуля данных создается специальный класс — наследник класса `TRemoteDataModule`. И фабрика класса на основе класса `TComponentFactory`

Примечание

Класс `TComponentFactory` представляет собой фабрику класса для компонентов Delphi, инкапсулирующих интерфейсы. Поддерживает интерфейс `IClassFactory`.

Создадим, например, удаленный модуль данных `SimpleRDM`. В мастере создания модуля данных в качестве способа создания выберем **Single Instance**, а **Free** — как модель обработки запросов.

Листинг 21.1. Исходный код нового удаленного модуля данных и его фабрики класса

```
type
  TSimpleRDM = class(TRemoteDataModule, ISimpleRDM)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean; const ClassID,
    ProgID: string); override;
  public
    { Public declarations }
  end;

implementation

{$R *.DFM}

class procedure TSimpleRDM.UpdateRegistry(Register: Boolean; const
ClassID, ProgID: string);
begin
  if Register then
```

```
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;

initialization
  TComponentFactory.Create(ComServer, TSimpleRDM,
    Class_SimpleRDM, ciMultiInstance, tmApartment);
end.
```

Обратите внимание, что параметры модуля данных, заданные при создании, использованы в фабрике класса `TComponentFactory` в секции `initialization`.

Примечание

Фабрика класса `TComponentFactory` обеспечивает создание экземпляров компонентов Delphi, поддерживающих использование интерфейсов.

Метод класса `UpdateRegistry` создается автоматически и обеспечивает регистрацию и аннулирование регистрации сервера Автоматизации. Если параметр `Register` имеет значение `True`, выполняется регистрация, иначе — отмена регистрации.

Разработчик не должен использовать этот метод, т. к. его вызов осуществляется автоматически.

Одновременно с модулем данных создается и его интерфейс — потомок интерфейса `IAppServer`. Его исходный код содержится в библиотеке типов проекта сервера приложения. Для удаленного модуля данных `SimpleRDM` созданный интерфейс `ISimpleRDM` представлен в листинге 21.2. Для удобства из листинга удалены автоматически добавляемые комментарии.

Листинг 21.2. Вновь созданная библиотека типов для сервера приложения с исходным кодом интерфейса удаленного модуля данных

```
LIBID_SimpleAppSrvr: TGUID = '{93577575-0F4F-43B5-9FBE-A5745128D9A4}';

IID_ISimpleRDM: TGUID = '{E2CBEBCB-1950-4054-B823-62906306E840}';
CLASS_SimpleRDM: TGUID = '{DB6A6463-5F61-485F-8F23-EC6622091908}';
```

```
type

  ISimpleRDM = interface;
  ISimpleRDMDisp = dispinterface;

  SimpleRDM = ISimpleRDM;

  ISimpleRDM = interface(IAppServer)
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
  end;

  ISimpleRDMDisp = dispinterface
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
    function  AS_ApplyUpdates(const ProviderName: WideString; Delta:
    OleVariant; MaxErrors: Integer; out ErrorCount: Integer;
    var OwnerData: OleVariant): OleVariant; dispid 20000000;
    function  AS_GetRecords(const ProviderName: WideString; Count:
    Integer; out RecsOut: Integer; Options: Integer; const CommandText:
    WideString; var Params: OleVariant; var OwnerData: OleVariant):
    OleVariant; dispid 20000001;
    function  AS_DataRequest(const ProviderName: WideString; Data:
OleVariant): OleVariant; dispid 20000002;
    function  AS_GetProviderNames: OleVariant; dispid 20000003;
    function  AS_GetParams(const ProviderName: WideString;
    var OwnerData: OleVariant): OleVariant; dispid 20000004;
    function  AS_RowRequest(const ProviderName: WideString; Row:
OleVariant; RequestType: Integer; var OwnerData: OleVariant): OleVariant;
dispid 20000005;
    procedure AS_Execute(const ProviderName: WideString; const
    CommandText: WideString; var Params: OleVariant;
    var OwnerData: OleVariant); dispid 20000006;
  end;

  CoSimpleRDM = class
    class function Create: ISimpleRDM;
    class function CreateRemote(const MachineName: string): ISimpleRDM;
  end;

implementation

uses ComObj;

class function CoSimpleRDM.Create: ISimpleRDM;
begin
  Result := CreateComObject(CLASS_SimpleRDM) as ISimpleRDM;
end;

class function CoSimpleRDM.CreateRemote(const MachineName: string):
ISimpleRDM;
```

```
begin
  Result := CreateRemoteComObject(MachineName, CLASS_SimpleRDM)
  as ISimpleRDM;
end;

end.
```

Обратите внимание, что интерфейс `ISimpleRDM` является потомком интерфейса `IAppServer`, рассмотренного выше.

Так как удаленный модуль данных реализует сервер Автоматизации, дополнительно к основному дуальному интерфейсу `ISimpleRDM` автоматически создан интерфейс диспетчеризации `ISimpleRDMDisp`. При этом для интерфейса диспетчеризации созданы методы, соответствующие методам интерфейса `IAppServer`.

Класс `CoSimpleRDM` обеспечивает создание COM-объектов, поддерживающих использование интерфейса. Для него автоматически созданы два метода класса.

Метод

```
class function Create: ISimpleRDM;
```

используется при работе с локальным и внутренним сервером (in process).

Метод

```
class function CreateRemote(const MachineName: string): ISimpleRDM;
```

используется в удаленном сервере.

Оба метода возвращают ссылку на интерфейс `ISimpleRDM`.

Теперь, если проект с созданным модулем данных сохранить и зарегистрировать, он станет доступен в удаленных клиентских приложениях как сервер приложения.

После создания удаленный модуль данных становится платформой для размещения компонентов доступа к данным и компонентов провайдеров *(см. гл. 20)*, которые, наряду с модулем данных, реализуют основные функции сервера приложения.

Дочерние удаленные модули данных

Один сервер приложения может содержать несколько удаленных модулей данных, которые, например, выполняют различные функции или обращаются к разным серверам БД. В этом случае процесс разработки серверной части не претерпевает изменений. При выборе имени сервера в компоненте удаленного соединения на стороне клиента *(см. гл. 22)* будут доступны имена всех удаленных модулей данных, включенных в состав сервера приложения.

Однако тогда для каждого модуля понадобится собственный компонент соединения. Если это нежелательно, можно использовать компонент `TSharedConnection`, но в этом случае в интерфейсы удаленных модулей данных необходимо внести изменения.

Для того чтобы несколько модулей данных были доступны в рамках одного удаленного соединения, необходимо выделить один главный модуль данных, а остальные сделать дочерними.

Рассмотрим, что же это означает для практики создания удаленных модулей данных. Суть идеи проста. Интерфейс главного модуля данных (разработчик назначает модуль главным, исходя из собственных соображений) должен содержать свойства, указывающие на интерфейсы всех других модулей данных, которые также необходимо использовать в рамках одного соединения на клиенте. Такие модули данных и называются дочерними.

Если такие свойства (свойство должно иметь атрибут только для чтения) существуют, все дочерние модули данных будут доступны в свойстве `ChildName` компонента `TSharedConnection` *(см. гл. 20)*.

Например, если дочерний удаленный модуль данных носит название `Secondary`, главный модуль данных должен содержать свойство `Secondary`:

```
ISimpleRDM = interface(IAppServer)
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
    function  Get_Secondary: Secondary; safecall;
    property Secondary: Secondary read Get_Secondary;
  end;
```

Реализация метода `Get_Secondary` выглядит так:

```
function TSimpleRDM.Get_Secondary: Secondary;
begin
 Result := FSecondaryFactory.CreateCOMObject(nil) as ISecondary;
end;
```

Как видите, в простейшем случае достаточно вернуть ссылку на вновь созданный дочерний интерфейс.

Полностью пример создания дочернего удаленного модуля данных рассматривается далее в этой главе.

Регистрация сервера приложения

Для того чтобы клиент мог "увидеть" сервер приложения, он должен быть зарегистрирован на компьютере сервера. В зависимости от используемой технологии процесс регистрации имеет особенности. Регистрация серверов MTS, Web и SOAP рассматривается далее в этой книге.

Здесь же мы остановимся на регистрации сервера приложения, использующего удаленный модуль данных `TRemoteDataModule` (сервер Автоматизации), который чрезвычайно прост.

Для исполняемых файлов достаточно запустить сервер с ключом `/regserver` или даже просто запустить исполняемый файл.

В среде разработки ключ можно поместить в диалоге команды меню **Run Parameters** (рис. 21.2).

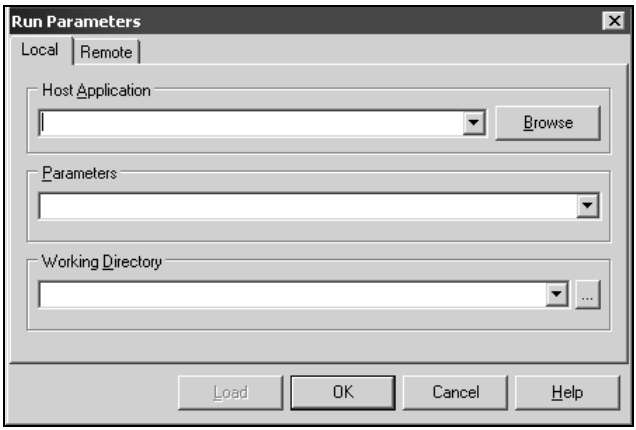


Рис. 21.2. Диалог параметров запуска приложения

Для удаления регистрации используется ключ `/unregserver`, но только в командной строке.

Для регистрации динамических библиотек применяется ключ `/regsvr32`.

Пример простого сервера приложения

В качестве примера рассмотрим процесс создания простого сервера приложения на основе удаленного модуля данных `TRemoteDataModule`. Для начала создадим новый проект — простое исполняемое приложение и сохраним его под именем `SimpleAppSrvr` (табл. 21.2). Этот проект входит в состав группы проектов `SimpleRemote`, в нее впоследствии будет добавлено клиентское приложение.

Таблица 21.2. *Файлы проекта* `SimpleAppSrvr`

Файл	Назначение
uSimpleAppSrvr.pas	Стандартный файл проекта
SimpleAppSrvr_TLB.pas	Библиотека типов. Содержит объявления всех используемых в проекте интерфейсов

Таблица 21.2 *(окончание)*

Файл	Назначение
uSimpleRDM.pas	Файл главного удаленного модуля данных `SimpleRDM`
uSecondary.pas	Файл дочернего удаленного модуля данных `Secondary`

Пример создания клиента для сервера приложения `SimpleAppSrvr` рассматривается в *гл. 22*.

Главный удаленный модуль данных

Добавим в проект новый удаленный модуль данных, используя для этого Репозиторий Delphi (см. рис. 20.3). Затем в появившемся диалоге (см. рис. 21.1) зададим имя модуля — `SimpleRDM` и его параметры:

- способ создания — `Single Instance` — для каждого клиента создается собственный модуль данных;
- способ обработки запросов — `Free` *(см. выше)*.

Метод класса `UpdateRegistry` для модуля данных создается автоматически и обеспечивает регистрацию и аннулирование регистрации сервера Автоматизации (см. листинг 21.1).

Одновременно с удаленным модулем данных автоматически создается библиотека типов и в ней дуальный интерфейс `ISimpleRDM` и интерфейс диспетчеризации `ISimpleRDMDisp` (см. листинг 21.2).

Примечание

Для каждого вновь созданного интерфейса автоматически назначается GUID.

Разместим в модуле `SimpleRDM` компоненты для доступа к файлам демонстрационной базы данных (\Program Files\Common Files\Borland Shared\Data) через драйвер BDE и псевдоним DBDEMOS, который создается автоматически при инсталляции Delphi. Это компонент `TDatabase`, обеспечивающий соединение и три табличных компонента `TTable`, инкапсулирующих наборы данных из таблиц Orders.db, Customer.db, Employee.db.

Компонент соединения настроен на псевдоним DBDEMOS (свойство `AliasName`). В параметрах соединения заданы имя пользователя и пароль, а свойство `LoginPrompt = False` запрещает отображение диалога регистрации при открытии соединения.

Каждый табличный компонент связан с компонентом-провайдером `TDataSetProvider`. Свойство провайдера `ResolveToDataSet = False` запрещает передачу изменений, полученных от клиента, в набор данных связанного

компонента. Вместо этого данные напрямую сохраняются в базе данных. Это увеличивает быстродействие приложения.

Дочерний удаленный модуль данных

Дополнительно к основному модулю данных создадим дочерний модуль данных `Secondary`. Для того чтобы связать главный модуль данных с дочерним, необходимо добавить к интерфейсу `ISimpleRDM` метод, возвращающий ссылку на интерфейс дочернего модуля данных. В нашем примере это метод `Get_Secondary`.

Для его создания воспользуемся библиотекой типов сервера (рис. 21.3).

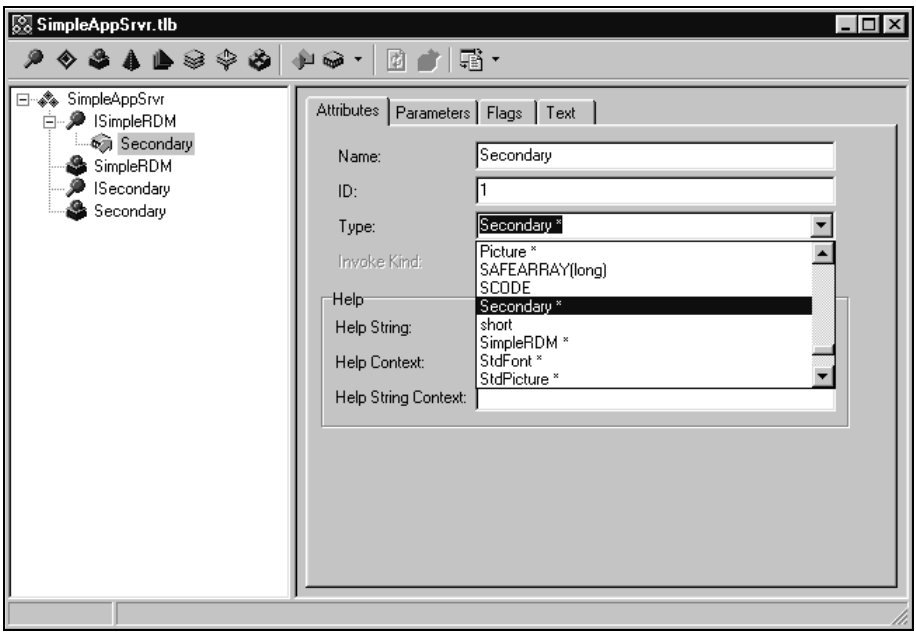


Рис. 21.3. Библиотека типов сервера приложения `SimpleAppSrvr`

В дереве в левой части окна выберем интерфейс `ISimpleRDM` и создадим для него новое свойство только для чтения, переименуем его в `Secondary`. Одновременно со свойством будет создан метод, обеспечивающий чтение свойства. Переименуем его в `Get_Secondary`. Метод должен возвращать тип `Secondary`. Для его установки воспользуемся списком **Type** на странице **Attributes** в правой части панели окна библиотеки типов (см. рис. 21.3).

После обновления исходного кода библиотеки типов (кнопка **Refresh Implementation**) описание нового свойства и метода интерфейса `ISimpleRDM`

появится в файле SimpleAppSrvr_TLB.pas. Теперь объявление интерфейса `ISimpleRDM` выглядит так:

```
ISimpleRDM = interface(IAppServer)
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
    function Get_Secondary: Secondary; safecall;
    property Secondary: Secondary read Get_Secondary;
  end;
```

Одновременно в объявлении удаленного модуля данных `SimpleRDM` в файле uSimpleRDM появится метод `Get_Secondary`. Его исходный код должен выглядеть следующим образом:

```
function TSimpleRDM.Get_Secondary: Secondary;
begin
 Result := FSecondaryFactory.CreateCOMObject(nil) as ISecondary;
end;
```

Теперь модуль данных `Secondary` стал дочерним для модуля `SimpleRDM`.

Модуль `Secondary` содержит компоненты для доступа к локальному серверу InterBase. База данных mastsql.gdb, используемая в этом примере, поставляется вместе с Delphi. Соединение обеспечивается компонентом `TIBDatabase`, который настроен на базу данных при помощи свойства `DatabaseName`.

Внимание!

Перед компиляцией проекта необходимо правильно настроить свойство `DatabaseName`, если местоположение файла mastsql.gdb отличается от обычного.

Два табличных компонента `TIBTable` инкапсулируют таблицы Vendors и Parts из базы данных mastsql.gdb. Дополнительно между этими двумя компонентами установлено отношение "один-ко-многим". Свойство `MasterSource` компонента `tblParts` указывает на компонент `dsVendors` (класс `TDataSource`), связанный с компонентом `tblVendors`. Свойства `MasterFields` и `IndexFieldNames` компонента `tblParts` содержат имя общего для двух таблиц поля `VendorNo` *(подробнее о создании отношения "один-ко-многим" см. гл. 14)*.

Отношение "один-ко-многим", созданное для двух таблиц, позволит продемонстрировать в примере клиентского приложения использование вложенных наборов данных *(см. гл. 22)*.

Регистрация сервера приложения

Теперь, когда сервер приложения готов, остался последний этап — регистрация сервера. Для этого достаточно запустить исполняемый файл проекта на компьютере, который будет использоваться для работы сервера приложе-

ния. Но обратите внимание, что в этом случае настройки доступа к используемым в примере базам данных должны быть скорректированы с учетом переноса на другой компьютер. И естественно, на нем должны быть установлен BDE и клиент InterBase.

После регистрации сервер приложения доступен из всех клиентских приложений, компоненты соединения DataSnap настроены на компьютер сервера приложения.

Резюме

Сервер приложения представляет собой ПО промежуточного слоя для трехзвенных распределенных приложений. Он обеспечивает связь удаленных клиентов с сервером БД и реализует большую часть бизнес-логики распределенного приложения.

В Delphi сервер приложения создается на основе удаленных модулей данных, реализация которых различается для различных технологий удаленного доступа. Удаленные модули данных имплементируют интерфейс `IAppServer`. Непосредственный доступ к данным обеспечивают компоненты-провайдеры `TDataSetProvider` при помощи интерфейса `IProviderSupport`.

ГЛАВА 22

Клиент многозвенного распределенного приложения

Клиентское ПО в распределенном многозвенном приложении имеет особенности архитектуры, продиктованные его ролью — ведь большая часть бизнес-логики и функций обработки данных сосредоточены в сервере приложений *(см. гл. 21)*. Такая схема призвана обеспечить более высокую эффективность обработки запросов многочисленных удаленных клиентов, а также упрощает обслуживание клиентского ПО. Клиенты, выполняющие лишь необходимый минимум операций, называются "тонкими".

Клиенты многозвенных приложений обеспечивают выполнение следующих функций:

- соединение с сервером приложений, прием и передача данных;
- отображение средствами пользовательского интерфейса;
- простейшие операции редактирования;
- сохранение локальных копий данных.

При разработке клиентских частей многозвенных приложений в Delphi используются компоненты DataSnap *(см. гл. 20)*, а также компонент `TClientDataSet`, роль которого трудно переоценить.

Помимо новых компонентов в процессе разработки применяются стандартные компоненты отображения данных, подробно рассматриваемые в *гл. 15*, а также обычная схема связывания визуальных компонентов с набором данных через компонент `TDataSource` *(см. гл. 11)*.

В этой главе рассматриваются следующие вопросы:

- структура клиентского приложения;
- соединение удаленного клиента с сервером приложений;
- набор данных клиента в компоненте `TClientDataSet`, локальное кэширование данных;

- основные операции обработки данных, выполняемые клиентским набором данных;
- вложенные наборы данных;
- обработка локальных ошибок клиентского набора данных и ошибок сервера приложений.

Структура клиентского приложения

По своей структуре (рис. 22.1) клиентское приложение подобно обычному приложению баз данных, рассматриваемому в *гл. 11*.

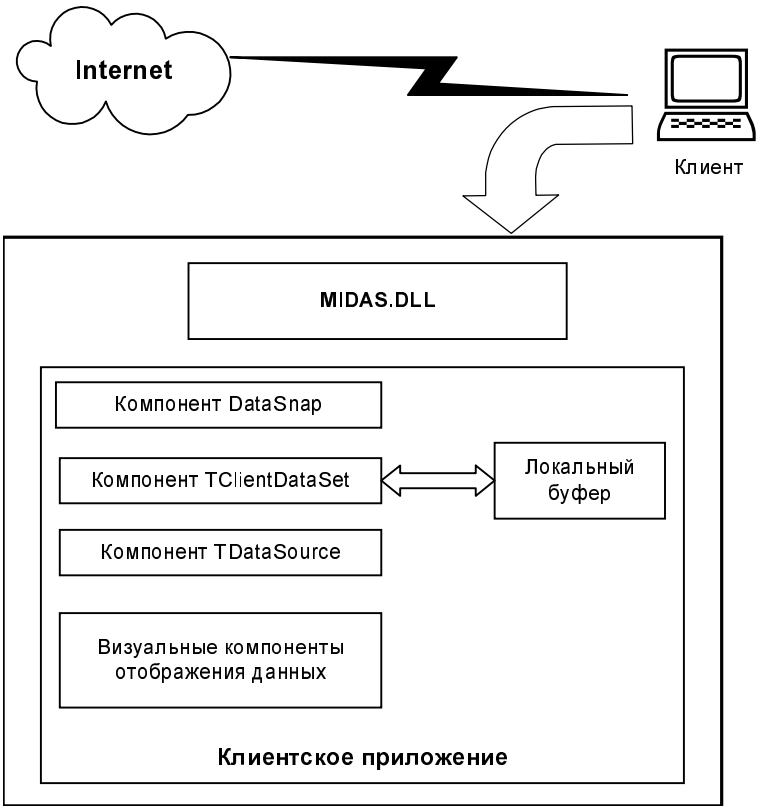


Рис. 22.1. Структура клиентской части многозвенного приложения Delphi

Соединение клиента с сервером приложений осуществляется специализированными компонентами DataSnap *(см. гл. 20)*. Эти компоненты взаимодействуют с удаленным модулем данных, входящим в состав сервера, при помощи методов интерфейса `IAppServer`.

Также в клиентском приложении могут использоваться дополнительные, определенные разработчиком, методы интерфейса удаленного модуля данных, унаследованного от интерфейса `IAppServer`. Подробнее об этих компонентах и способах их настройки на удаленный сервер приложений *см. гл. 21.*

Внимание!

Соединение с сервером приложений обеспечивает динамическая библиотека MIDAS.DLL, которая должна быть зарегистрирована на компьютере клиента.

Как и обычное приложение БД, клиент многозвенного распределенного приложения должен содержать компоненты, инкапсулирующие набор данных, которые связаны с визуальными компонентами отображения данных посредством компонентов `TDataSource`.

Очевидно, что набор данных сервера должен быть скопирован клиентским приложением в некий локальный буфер. При этом должен использоваться эффективный механизм загрузки данных сравнительно небольшими порциями, что позволяет значительно разгрузить транспортный канал между клиентом и сервером приложений.

Кэширование и редактирование данных в клиентском приложении обеспечивает специализированный компонент `TClientDataSet`, отдаленным предком которого является класс `TDataSet`. Помимо унаследованных от предков методов, класс `TClientDataSet` инкапсулирует ряд дополнительных функций, облегчающих управление данными.

Примечание

Подобно обычному приложению БД, в "тонком" клиенте для размещения невизуальных компонентов доступа к данным необходимо использовать модули данных.

Для получения набора данных сервера компонент `TClientDataSet` взаимодействует с компонентом `TDataSetProvider`, используя методы интерфейса `IProviderSupport` *(см. гл. 21).*

По существу все уникальные функции клиентского приложения сосредоточены в компоненте `TClientDataSet`, изучением которого мы и займемся далее в этой главе. В остальном клиентское приложение не отличается от обычного приложения БД и при его разработке могут применяться стандартные методы.

Клиентские наборы данных

В Палитре компонентов Delphi представлено несколько компонентов, инкапсулирующих клиентский набор данных. В то же время при разработке настоящих удаленных клиентских приложений применяется компонент

TClientDataSet. Внесем ясность в этот вопрос. Итак, помимо компонента TClientDataSet, расположенного на странице **Data Access**, существуют еще два компонента:

- TSimpleDataSet — разработан для технологии доступа к данным dbExpress и, по существу, является единственным полноценным средством для работы с набором данных в рамках этой технологии;
- TIBClientDataSet — используется в технологии доступа к данным сервера InterBase — InterBase Express.

Все перечисленные компоненты произошли от общего предка — класса TCustomClientDataSet (рис. 22.2). Они обеспечивают локальное кэширование данных и взаимодействие с серверным набором данных при посредстве интерфейса IProviderSupport.

Основное различие между компонентом TClientDataSet и другими клиентскими компонентами заключается в том, что первый предназначен для использования с *внешним* компонентом-провайдером данных. А значит, он может взаимодействовать с удаленным провайдером данных.

Остальные перечисленные компоненты инкапсулируют *внутренний* провайдер данных, предоставляя тем самым для использования в рамках соответствующих технологий доступа к данным эффективный механизм локального кэширования данных. Использование внутреннего провайдера данных обеспечивает общий класс-предок TCustomCachedDataSet.

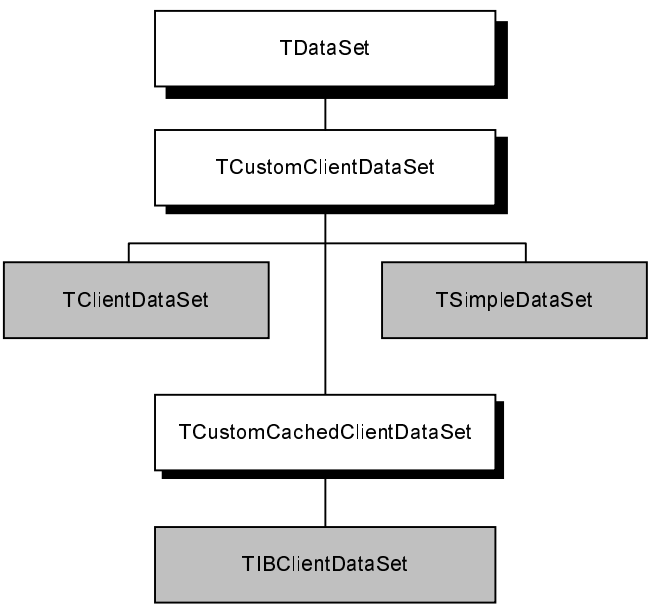


Рис. 22.2. Иерархия классов клиентских наборов данных

Для этого он имеет защищенное свойство

```
property Provider: TDataSetProvider;
```

Соединение с источником данных осуществляется не свойством `RemoteServer` (будет рассмотрено ниже применительно к компоненту `TClientDataSet`), задающим удаленный сервер, а стандартными средствами соответствующей технологии доступа к данным.

Таким образом, для работы с удаленными данными (т. е. внешними по отношению к клиенту) пригоден только компонент `TClientDataSet`, умеющий работать с внешним провайдером данных.

Компонент *TClientDataSet*

Компонент `TClientDataSet` используется в клиентской части многозвенного распределенного приложения. Он инкапсулирует набор данных, переданный при помощи компонента-провайдера из удаленного набора данных. Компонент обеспечивает выполнение следующих основных функций:

- получение данных от удаленного сервера и передача ему сделанных изменений с использованием удаленного компонента-провайдера;
- представление набора данных при помощи локального буфера и поддержка основных операций, унаследованных от класса `TDataSet`;
- объединение записей набора данных при помощи агрегатных функций для получения суммарных данных;
- локальное сохранение набора данных в файле и последующее восстановление набора данных из файла;
- представление набора данных в формате XML.

Предком компонента `TClientDataSet` является класс `TDataSet`, поэтому `TClientDataSet` обладает таким же набором функций, что и обычный компонент, инкапсулирующий набор данных. Основное же отличие заключается в том, источник данных для него доступен только через удаленный компонент-провайдер. Это означает, что сохранение изменений и обновление набора данных осуществляется локально, без обращения к источнику данных.

Например, выполнение метода `Post` приведет лишь к сохранению текущей записи набора данных в локальном кэше. Все изменения отсылаются на сервер только при необходимости и легко управляются разработчиком.

Как и обычный компонент, компонент `TClientDataSet` может использоваться совместно с визуальными компонентами отображения данных. Для этого нужен компонент `TDataSource`.

Рассмотрим основные функции, реализуемые компонентом `TClientDataSet`.

Получение данных от компонента -провайдера

Компонент `TClientDataSet` получает доступ к удаленным данным через компонент соединения DataSnap *(см. гл. 20)*. В зависимости от используемой технологии это могут быть компоненты `TDCOMConnection`, `TSocketConnection`, `TWebConnection` или `TCorbaConnection`.

Компонент `TClientDataSet` связывается с компонентом соединения при помощи свойства

```
property RemoteServer: TCustomRemoteServer;
```

Если соединение настроено правильно, то ссылка на интерфейс `IAppServer` в свойстве

```
property AppServer: IAppServer;
```

совпадает со свойством

```
ClientDataSet.RemoteServer.AppServer;
```

После настройки соединения в свойстве

```
property ProviderName: string;
```

можно выбрать один из компонентов-провайдеров, которые доступны на сервере приложений, выбранном в компоненте соединения.

Если провайдер был подключен правильно, свойство только для чтения

```
property HasAppServer: Boolean;
```

автоматически принимает значение `True`.

Теперь компонент готов к приему данных. При использовании метода

```
procedure Open;
```

или свойства

```
property Active: Boolean;
```

компонент получает от провайдера первый пакет данных.

Размер пакета определяется свойством

```
property PacketRecords: Integer;
```

которое задает число записей, передаваемое в одном пакете. Если свойство имеет значение −1 (это значение по умолчанию), передаются все записи набора данных. Если оно равно 0 — клиенту передаются только метаданные о наборе данных.

Если соединение клиента с сервером медленное, число записей в пакете можно уменьшить, но желательно так, чтобы при использовании компонентов `TDBGrid` полученные в одном пакете записи полностью заполняли рабочую область этого компонента.

Одновременно разработчик имеет возможность управлять доставкой следующих пакетов. Для этого используется метод

```
function GetNextPacket: Integer;
```

Например, это можно сделать следующим образом:

```
procedure TDataModule1.ClientDataSetAfterScroll(DataSet: TDataSet);
begin
 if ClientDataSet.EOF then ClientDataSet.GetNextPacket;
end;
```

Свойство

```
property FetchOnDemand: Boolean;
```

должно иметь значение `False`. При значении `True` оно разрешает компоненту получать новые пакеты данных по мере надобности, например, при необходимости прокрутки записей в компоненте `TDBGrid`.

До и после получения очередного пакета соответственно выполняются обработчики событий:

```
type
  TRemoteEvent = procedure(Sender: TObject; var OwnerData: OleVariant)
of object;

property BeforeGetRecords: TRemoteEvent;
property AfterGetRecords: TRemoteEvent;
```

Содержимое очередного пакета представлено свойством

```
property Data: OleVariant;
```

Данные в нем хранятся в транспортном формате, готовые для пересылки. Причем его можно использовать не только для чтения, но и для записи, формируя пакет данных для отправки провайдеру:

```
var  OwnerData: OleVariant;
     MaxErrors, ErrorCount: Integer;
...
MaxErrors := 0;
  ResultDataSet.Data := SourceDataSet.AppServer.AS_ApplyUpdates('',
SourceDataSet.Delta, MaxErrors, ErrorCount, OwnerData);
...
```

Метод `AS_ApplyUpdates` передает данные, содержащиеся в буфере `Delta`, провайдеру на сервер и возвращает записи, сохранить которые не удалось. Подробнее о методе `AS_ApplyUpdates` см. табл. 21.1.

Размер буфера `Data` в байтах возвращает свойство

```
property DataSize: Integer;
```

Кэширование и редактирование данных

После получения записей от провайдера набор данных сохраняется в локальном буфере памяти. И все вносимые изменения после применения метода `Post` также сохраняются локально и не пересылаются на сервер. Буфер изменений доступен при помощи свойства

```
property Delta: OleVariant;
```

Для передачи изменений на сервер используется метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

где параметр `MaxErrors` задает число ошибок, которые игнорируются при сохранении данных на сервере. Если параметр равен −1, сохранение на сервере прерывается при первой же ошибке. Метод возвращает число сохраненных записей.

После выполнения метода `ApplyUpdates` все записи, сохранить которые не удалось, возвращаются клиенту в локальный буфер `Delta`.

Если клиентское приложение будет редко изменять свои наборы данных, сохранение изменений на сервере можно связать с методом-обработчиком `AfterPost`:

```
procedure TForm1.ClientDataSetAfterPost(DataSet: TDataSet);
begin
 ClientDataSet.ApplyUpdates(-1);
end;
```

Свойство только для чтения

```
property ChangeCount: Integer;
```

возвращает общее число изменений, содержащееся в буфере `Delta`.

Для очистки буфера изменений используется метод

```
procedure CancelUpdates;
```

После вызова метода свойство `ChangeCount` принимает значение 0.

До и после сохранения изменений на сервере соответственно вызываются методы-обработчики

```
property BeforeApplyUpdates: TRemoteEvent;
property AfterApplyUpdates: TRemoteEvent;
```

Несмотря на сделанные локально многократные изменения, запись может быть восстановлена в первоначальном виде. Метод

```
procedure RefreshRecord;
```

получает от провайдера первоначальный вариант текущей записи, сохраненный на сервере.

При этом (и при всех других случаях, когда компонент запрашивает обновление текущей записи) вызываются методы-обработчики

```
property BeforeRowRequest: TRemoteEvent;
property AfterRowRequest: TRemoteEvent;
```

Но что делать, если необходимо восстановить удаленную запись? В обычном наборе данных после сохранения это невозможно. В компоненте `TClientDataSet` существует метод

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

который возвращает набор данных к состоянию до последней выполненной операции редактирования, добавления или удаления записи. Если параметр `FollowChange` имеет значение `True`, курсор набора данных будет установлен на восстановленную запись.

О состоянии текущей записи позволяет судить метод

```
function UpdateStatus: TUpdateStatus; override;
```

который возвращает значение типа

```
TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

означающее состояние текущей записи:

`usUnmodified` — запись осталась неизменной;

`usModified` — запись была изменена;

`usInserted` — запись была добавлена;

`usDeleted` — запись была удалена.

Например, при закрытии набора данных можно выполнить проверку:

```
if ClientDataSet.UpdateStatus = usModified
 then ShowMessage('Record was changed');
```

На основе типа можно управлять видимостью записей в наборе данных. Свойство

```
property StatusFilter: TUpdateStatusSet;
```

определяет, какой тип записей будет отображаться в наборе данных. Например:

```
ClientDataSet.StatusFilter := usDeleted;
```

отобразит в наборе данных только удаленные записи (при этом изменения не сохранены на сервере).

Управление запросом на сервере

Компонент `TClientDataSet` может не только эффективно управлять своим набором данных, но и влиять на выполнение серверного компонента, с которым он связан через провайдер.

Свойство

```
property CommandText: string;
```

содержит текст запроса SQL, имя таблицы или хранимой процедуры в зависимости от типа серверного компонента.

Изменив значение этого свойства на клиенте, можно, например, модифицировать запрос SQL на сервере. Но для этого в свойстве `Options` соответствующего компонента-провайдера `TDataSetProvider` должно быть установлено значение

```
poAllowCommandText := True;
```

Новое значение свойства `CommandText` отправляется на сервер только после открытия клиентского набора данных или выполнения метода

```
procedure Execute; virtual;
```

Для запросов или хранимых процедур можно задавать параметры, которые сохраняются в свойстве

```
property Params: TParams;
```

До выполнения запроса присваиваются значения входным параметрам. После выполнения хранимой процедуры в выходных параметрах размещаются полученные от сервера значения.

Обратите внимание, что при выполнении запросов или хранимых процедур может измениться порядок следования параметров. Поэтому обращаться к параметрам желательно по их именам. Например, так:

```
Edit1.Text := ClientDataSet.Params.ParamByName('OutputParam').AsString;
```

Для того чтобы получить текущие значения параметров компонента набора данных на сервере, достаточно использовать метод

```
procedure FetchParams;
```

Перед и после получения параметров от провайдера, клиентский набор данных вызывает методы-обработчики событий:

```
property BeforeGetParams: TRemoteEvent;
property AfterGetParams: TRemoteEvent;
```

Использование индексов

Обычно использование индексов — прерогатива сервера БД. Из компонентов Delphi только табличные компоненты могут в какой-то степени управлять использованием индексов. Очевидно, что удаленное соединение не способствует эффективному управлению индексами набора данных на сервере. Поэтому компонент `TClientDataSet` предоставляет разработчику возможность создавать и использовать локальные индексы.

Правильно созданные и используемые локальные индексы могут существенно ускорить выполнение операций с набором данных. В то же время их невозможно сохранить вместе с набором данных локально, их необходимо перестраивать при каждом новом открытии набора данных и его обновлении с сервера.

Для создания локального индекса используется метод

```
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions;
const DescFields: string = ''; const CaseInsFields: string = '';
const GroupingLevel: Integer = 0);
```

Параметр `Name` определяет имя нового индекса. Параметр `Fields` должен содержать имена полей, которые разработчик хочет включить в индекс. Имена полей должны разделяться точкой с запятой. Параметр `Options` позволяет задать тип индекса:

```
TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive,
ixExpression, ixNonMaintained);
  TIndexOptions = set of TIndexOption;
```

`ixPrimary` — первичный индекс;

`ixUnique` — значения индекса уникальны;

`ixDescending` — индекс сортирует записи в обратном порядке;

`ixCaseInsensitive` — индекс сортирует записи без учета регистра символов;

`ixExpression` — в индексе используется выражение (для индексов dBASE);

`ixNonMaintained` — индекс не обновляется при открытии таблицы.

При этом можно задать поля, порядок сортировки которых будет обратным. Для этого их необходимо перечислить через точку с запятой в параметре `DescFields`. Параметр `CaseInsFields` аналогичным образом позволяет задать поля, на сортировку которых не влияет регистр символов.

Параметры `DescFields` и `CaseInsFields` используются вместо параметра `Options`.

Параметр `GroupingLevel` задает уровень группировки полей индекса. Подробнее об этом *см. ниже в разд. "Агрегаты" этой главы*.

Основные свойства компонента, обеспечивающие управление индексами, совпадают с аналогичными свойствами табличных компонентов *(подробнее об этом см. гл. 12)*. Поэтому лишь кратко перечислим их.

При работе с компонентом разработчик имеет возможность управлять индексами.

Созданный индекс подключается к набору данных свойством

```
property IndexName: String;
```

которое должно включать имя индекса или использовать свойство

```
property IndexFieldNames: String;
```

в котором можно задать произвольное сочетание имен индексированных полей таблицы. Имена полей разделяются точкой с запятой. Свойства `IndexName` и `IndexFieldNames` нельзя использовать одновременно.

Число полей, используемых в текущем индексе табличного компонента, возвращает свойство

```
property IndexFieldCount: Integer;
```

Свойство

```
property IndexFields: [Index: Integer]: TField;
```

представляет собой индексированный список полей, входящих в текущий индекс.

Параметры созданных индексов доступны в свойстве

```
property IndexDefs: TIndexDefs;
```

Класс `TIndexDefs` подробно рассматривается в *гл. 12*.

После создания и подключения индекса записи набора данных "переупорядочиваются" в соответствии со значениями индексированных полей.

Удаление локального индекса обеспечивает метод

```
procedure DeleteIndex(const Name: string);
```

После удаления текущего индекса или его отмены (обнуления свойства `IndexName`) записи набора данных "переупорядочиваются" в исходном порядке, соответствующем порядку записей набора данных на сервере.

Имена всех существующих в наборе данных индексов можно загрузить в список при помощи метода

```
procedure GetIndexNames(List: TStrings);
```

Например:

```
Memo1.Lines.Clear;
ClientDataSet.GetIndexNames(Memo1.Lines);
```

Сохранение набора данных в файлах

Клиентское приложение может использовать одну очень удобную функцию компонента `TClientDataSet`. Представим, что соединение между сервером и клиентом обладает малой пропускной способностью и к тому же часто обрывается. Что в этом случае делать пользователю, который внес много изменений и не может сохранить их на сервере?

В этом случае можно сохранить набор данных клиента в файле на локальном диске, а при удобной возможности — загрузить обратно и переслать на сервер.

Для сохранения данных (по существу это буфер `Data`) в файле используется метод

```
procedure SaveToFile(const FileName: string = '';
Format: TDataPacketFormat=dfBinary);
```

Причем, если параметр `FileName` пуст, имя файла берется из свойства

```
property FileName: string;
```

Также можно передать данные в поток:

```
procedure SaveToStream(Stream: TStream; Format: TDataPacketFormat=dfBinary);
```

Формат, в котором данные будут сохранены, определяется параметром `Format`:

```
type TDataPacketFormat = (dfBinary, dfXML, dfXMLUTF8);
```

где `dfBinary` — бинарный вид, `dfXML` — формат XML, `dfXMLUTF8` — формат XML в кодировке UTF8.

Обратная загрузка данных, соответственно, выполняется методами:

```
procedure LoadFromFile(const FileName: string = '');
```

и

```
procedure LoadFromStream(Stream: TStream);
```

После загрузки набор данных полностью готов к работе:

```
if LoadFileDialog.Execute then
begin
 ClientDataSet.LoadFromFile(LoadFileDialog.FileName);
 ClientDataSet.Open;
end;
```

Работа с данными типа BLOB

Если набор данных сервера содержит большие поля (например, изображения), передача данных по медленному каналу займет очень много времени,

что, несомненно, снизит эффективность приложения. Простейшее решение проблемы — передача клиенту данных типа BLOB только в том случае, когда это ему действительно необходимо — т. е. исключительно по его запросу.

В компоненте `TClientDataSet` процессом передачи полей типа BLOB можно управлять, используя свойство

```
property FetchOnDemand: Boolean;
```

По умолчанию оно равно значению `True` и клиентский набор данных "выкачивает" данные BLOB по мере необходимости автоматически. Это означает, что приложение будет останавливаться и заново получать данные при любом просмотре данных, прокрутке и т. д. Если свойство имеет значение `False`, для получения данных клиент должен явно вызвать метод

```
procedure FetchBlobs;
```

Но, кроме этого, в свойстве `Options` компонента-провайдера `TDataSetProvider` обязательно должно быть установлено значение:

```
poFetchBlobsOnDemand := True;
```

Представление данных в формате XML

Набор данных клиента легко можно представить в формате XML. Для этого достаточно использовать свойство

```
property XMLData: OleVariant;
```

которое возвращает данные, содержащиеся в буфере `Data` *(см. выше)* в бинарном виде, в формате XML.

Например, клиентский набор данных можно сохранить в файле формата XML:

```
if SaveDialog.Execute then
 with TFileStream.Create(SaveDialog.FileName, fmCreate) do
 try
  Write(Pointer(ClientDataSet.XMLData)^, Length(ClientDataSet.XMLData));
 finally
  Free;
 end;
```

Агрегаты

Наличие локального буфера данных позволяет компоненту `TClientDataSet` реализовать ряд дополнительных функций, основанных на использовании агрегатных функций применительно к полям всего набора данных, загруженного в локальный буфер.

К агрегатным функциям относятся:

- `AVG` — вычисляет среднее значение;
- `COUNT` — возвращает число записей;
- `MIN` — вычисляет минимальное значение;
- `MAX` — вычисляет максимальное значение;
- `SUM` — вычисляет сумму.

Для их применения в компоненте `TClientDataSet` предусмотрены:

- индексированный список объектов, инкапсулирующих агрегатные выражения — агрегаты;
- агрегатные поля, обеспечивающие получение новых значений подобно вычисляемым полям, но с группированием записей на основе использования агрегатных функций.

Объекты-агрегаты

Для вычисления агрегатных выражений для всех записей набора данных используются объекты класса `TAggregate`. Индексированный список этих объектов содержится в свойстве

```
property Aggregates: TAggregates;
```

компонента `TClientDataSet`. Прямым предком класса `TAggregates` является класс `TCollection`, поэтому для него можно использовать все основные приемы работы с коллекциями *(см. гл. 7)*.

Для создания нового агрегата необходимо щелкнуть на кнопке свойства в Инспекторе объектов и, в появившемся Редакторе агрегатов, выбрать пункт **Add** во всплывающем меню или щелкнуть на кнопке **Add New** (рис. 22.3).

Новый агрегат может быть добавлен и динамически:

```
var NewAgg: TAggregate;
...
NewAgg := ClientDataSet.Aggregates.Add;
...
```

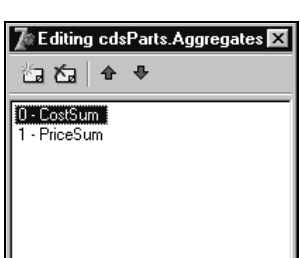


Рис. 22.3. Редактор агрегатов компонента `TClientDataSet`

Рассмотрим свойства класса `TAggregate`.

Имя агрегата содержится в свойстве

```
property AggregateName: string;
```

которое может быть использовано при отображении агрегата в визуальных компонентах.

Вычисляемое выражение с применением агрегатных функций должно находиться в свойстве

```
property Expression: String;
```

Например, для таблицы COUNTRY.DB из демонстрационной базы данных Delphi можно вычислять общую площадь государств Северной и Южной Америки (площадь государства содержится в поле `Area`):

```
ClientDataSet.Aggregates[SomeIndex].Expression := 'SUM(Area)';
```

Вычислением агрегата управляет свойство

```
property Active: Boolean;
```

а вычисленное значение возвращает функция

```
function Value: Variant;
```

Если пользователь редактирует набор данных, то для всех включенных агрегатов (`Active = True`) возвращаемое значение автоматически пересчитывается.

Например, после сохранения изменений в наборе данных можно визуализировать новое значение агрегата:

```
SomeLabel.Caption := ClientDataSet.Aggregates[0].AggregateName;
SomeEdit.Text := ClientDataSet.Aggregates[0].Value;
```

Для проверки активности агрегата, помимо проверки значения свойства `Active`, можно также использовать свойство

```
property InUse: Boolean;
```

Если оно возвращает значение `True` — вычисляемое выражение агрегата рассчитывается.

Видимость агрегата в визуальных компонентах управляется свойством

```
property Visible: Boolean;
```

Для того чтобы снизить вычислительную нагрузку на набор данных, можно отключить все агрегаты одновременно. Для этого свойству

```
property AggregatesActive: Boolean;
```

необходимо присвоить значение `False`.

Если же `AggregatesActive = True`, вычисляются только активные агрегаты, для которых их свойство `Active` имеет значение `True`.

Если вам необходимо использовать все активные агрегаты, то вместо их последовательного перебора с проверкой свойства `Active` можно использовать свойство

```
property ActiveAggs[Index: Integer] : TList;
```

компонента `TClientDataSet`, которое представляет собой список активных агрегатов.

Агрегатные поля

Агрегатные поля не входят в структуру полей набора данных, т. к. агрегатные функции подразумевают объединение записей таблицы для получения результата. Следовательно, значение агрегатного поля нельзя связать с какой-то одной записью, оно относится ко всем или группе записей.

Агрегатные поля не отображаются вместе со всеми полями в компонентах `TDBGrid`, в Редакторе полей они расположены в отдельном списке. Для представления значения агрегатного поля можно воспользоваться одним из компонентов отображения данных, который визуализирует значение одного поля (например, `TDBText` или `TDBEdit`) или свойствами самого поля:

```
Label1.Caption := MyDataSetAGGRFIELD1.AsString;
```

Подробно вопросы создания агрегатных полей рассмотрены в *гл. 13*.

Класс `TAggregateField` предназначен для инкапсуляции свойств и методов агрегатных полей.

Его свойство

```
property Expression: string;
```

задает вычисляемое выражение.

Вычисление значения проводится только для тех агрегатных полей, свойство

```
property Active: Boolean;
```

которых имеет значение `True`.

Вычисление включенных свойством `Active` агрегатных полей выполняется только в том случае, если булевское свойство `AggregatesActive` клиентского компонента набора данных имеет значение `True`.

По умолчанию экземпляр класса `TAggregateField` создается со свойством `Visible = False`.

Группировка и использование индексов

Каждый агрегат (объект или поле) имеет свойство

```
property GroupingLevel: Integer;
```

которое задает уровень группировки полей набора данных при вычислении. При значении 0 расчет проводится для всех записей набора данных. При значении 1 записи группируются по первому полю набора данных и расчет осуществляется для каждой группы. При значении 2 записи разбиваются на группы по первому и второму полям и т. д.

Однако группировка по уровням выше нулевого возможна, только если в наборе данных используется индекс по группирующим полям. Например, если свойство `GroupingLevel = 2` и набор данных начинается с полей `CustNo` и `OrderNo`, в свойстве `IndexName` компонента `TClientDataSet` и свойстве

```
property IndexName: String;
```

агрегата (объекта или поля) должно быть имя индекса, включающего оба эти поля.

Вложенные наборы данных

В *гл. 14* рассматривался вопрос организации между таблицами отношения "один-ко-многим", когда через одинаковое значение поля внешнего ключа одна запись главной таблицы связывается с несколькими записями подчиненной таблицы. Этот широко распространенный в практике программирования приложений БД механизм реализован и в компоненте `TClientDataSet`. Для этого используются класс поля `TDataSetField`.

На стороне клиента для создания отношения "один-ко-многим" необходимо использовать как минимум два компонента `TClientDataSet`, главный из которых инкапсулирует основной набор данных, а подчиненный — вложенный набор данных.

Итак, на стороне сервера есть два табличных компонента, связанных отношением "один-ко-многим" при помощи свойств `MasterSource` и `MasterFields` *(см. гл. 14)*. Также это могут быть и два компонента запросов SQL, связанные параметрами подчиненного запроса с одноименными полями главного запроса и свойством `DataSource`.

Теперь на стороне клиента необходимо при помощи компонента-провайдера связать компонент `TClientDataSet` с главным серверным компонентом отношения "один-ко-многим" и создать для него статические объекты для всех полей. Для этого достаточно дважды щелкнуть на компоненте и в окне Редактора полей (см. рис. 22.3) из всплывающего меню выбрать пункт **Add Field**. В результате в окне Редактора полей появятся имена объектов для

всех полей серверного набора данных, а также еще одно дополнительное поле объектного типа `TDataSetField`. Его имя совпадает с именем подчиненного серверного компонента отношения "один-ко-многим".

Это поле связано с подчиненным компонентом на сервере. Чтобы убедиться в этом, достаточно просмотреть значение его свойства только для чтения

```
property NestedDataSet: TDataSet;
```

Индексированный список всех полей, передаваемых из серверного подчиненного компонента, содержится в свойстве только для чтения

```
property Fields: TFields;
```

В дальнейшем связь между компонентами на клиенте настраивается именно через это поле. В подчиненном компоненте `TClientDataSet` в Инспекторе объектов необходимо выбрать свойство

```
property DataSetField: TDataSetField;
```

В списке этого свойства вы увидите имя только что созданного поля объектного типа `TDataSetField`. Выберите его и отношение "один-ко-многим" для клиентских наборов данных готово. При этом в компоненте вложенного набора данных автоматически очищаются свойства `RemoteServer` и `ProviderName`, т. к. их значения утрачивают значение и компонент оказывается связан только с главным компонентом отношения "один-ко-многим".

Теперь при навигации по записям основного набора данных во вложенном наборе данных автоматически будут появляться связанные записи. Также вы можете использовать все возможности, предоставляемые компонентом `TClientDataSet` как для основного, так и для вложенного набора данных.

Дополнительные свойства полей клиентского набора данных

Как известно, все классы полей имеют одного общего предка — класс `TField`. Подробно эти классы рассматриваются в *гл. 13*. Здесь же остановимся лишь на нескольких дополнительных свойствах полей, которые работают только в режиме кэширования в обычных компонентах, инкапсулирующих набор данных, и в компоненте `TClientDataSet`. Причем в компоненте `TClientDataSet` реализация этих свойств обеспечена локальным кэшем.

Итак, для разработчика могут быть полезны свойства объектов полей, содержащие не только текущее, но и предыдущее значение поля.

Свойство

```
property CurValue: Variant;
```

возвращает текущее значение поля.

Свойство

```
property OldValue: Variant;
```

содержит значение поле, которое было до начала редактирования.

Свойство

```
property NewValue: Variant;
```

содержит новое значение, которое может быть присвоено при обработке ошибки сервера методом-обработчиком `OnReconcileError` *(см. ниже)*.

Обработка ошибок

Особенности использования компонента `TClientDataSet` распространяются также и на обработку ошибок. Ведь клиентский набор данных должен реагировать не только на ошибки, возникшие локально, но и на ошибки сохранения изменений на сервере.

В первом случае разработчик может применить стандартные способы. Это использование блоков `try..except` или методов обработчиков, унаследованных от класса `TDataSet`:

- `property OnDeleteError: TDataSetErrorEvent;` — вызывается при ошибках удаления записей;
- `property OnEditError: TDataSetErrorEvent;` — вызывается при ошибках редактирования записей;
- `property OnPostError: TDataSetErrorEvent;` — вызывается при ошибках локального сохранения записей.

Все они используют процедурный тип

```
type
  TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
var Action: TDataAction) of object;
```

Здесь, помимо параметров `DataSet` и `E`, определяющих соответственно набор данных и тип ошибки, параметром `Action` можно задать вариант реакции на ошибку:

```
type TDataAction = (daFail, daAbort, daRetry);
```

`daFail` — прервать операцию и показать сообщение об ошибке;

`daAbort` — прервать операцию без сообщения об ошибке;

`daRetry` — повторить операцию.

Например, при возникновении ошибки редактирования набора данных код обработчика может выглядеть следующим образом:

```
procedure TForm1.ClientDataSetEditError(DataSet: TDataSet;
E: EDatabaseError; var Action: TDataAction);
begin
 if Not (DataSet.State in [dsEdit, dsInsert]) then
 begin
  DataSet.Edit;
  Action := daRetry;
 end
 else Action := daAbort;
end;
```

Здесь, если набор данных не находится в состоянии редактирования, это упущение исправляется и операция повторяется.

Итак, с локальными ошибками все обстоит достаточно просто. А как клиентский набор данных "узнает" об ошибке на удаленном сервере? Очевидно, при помощи своего компонента-провайдера. Действительно, компонент `TDataSetProvider` не только возвращает клиенту несохраненные изменения в пакете `Delta` *(см. выше)*, но и обеспечивает генерацию события, реакцией на которое является метод-обработчик

```
type
  TReconcileErrorEvent = procedure(DataSet: TCustomClientDataSet;
E: EReconcileError; UpdateKind: TUpdateKind; var Action:
TReconcileAction) of object;
property OnReconcileError: TReconcileErrorEvent;
```

Обратите внимание, что все параметры похожи на соответствующие параметры локальных обработчиков, но имеют собственные типы. Рассмотрим их.

Параметр `UpdateKind` содержит указание на тип операции, вызвавшей ошибку на сервере:

```
type
  TUpdateKind = (ukModify, ukInsert, ukDelete);
```

`ukModify` — изменение данных;

`ukInsert` — добавление записей;

`ukDelete` — удаление записей.

Параметр `Action` позволяет разработчику предусмотреть реакцию клиентского набора данных на ошибку:

```
type
  TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,
raRefresh);
```

`raSkip` — отменить операцию для записей, вызвавших ошибку, с их сохранением в буфере;

`raAbort` — отменить все изменения для операции, вызвавшей ошибку;

`raMerge` — совместить измененные записи с аналогичными записями сервера;

`raCorrect` — сохранить изменения, сделанные в данном методе-обработчике;

`raCancel` — отменить изменения, вызвавшие ошибку, заменив их исходными локальными значениями клиентского набора данных;

`raRefresh` — отменить изменения, вызвавшие ошибку, заменив их исходными значениями серверного набора данных.

Как видите, выбор возможных реакций на ошибку сервера несколько шире, чем на локальные ошибки.

Тип ошибки возвращается параметром `E`, для которого предусмотрен специальный класс `EReconcileError`, имеющий несколько полезных свойств.

Свойство

```
property ErrorCode: DBResult;
```

возвращает код ошибки. Используемые коды ошибок можно найти в файле \Source\Vcl\DSIntf.pas. Код предыдущей ошибки возвращается свойством

```
property PreviousError: DBResult;
```

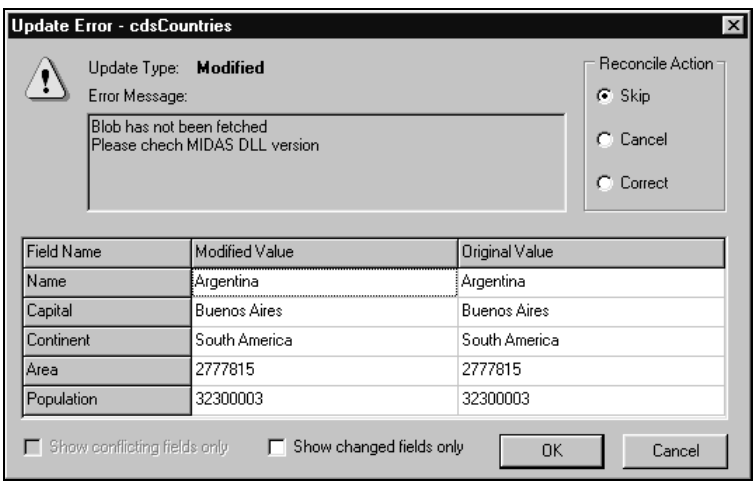


Рис. 22.4. Стандартный диалог обработки ошибок сервера

Используя представленную здесь информацию, вы можете самостоятельно управлять обработкой ошибок сервера на клиенте. Но можно поступить и более просто — использовать стандартный диалог обработки удаленных ошибок (рис. 22.4). Этот диалог можно подключить к вашему проекту (он содержится в модуле \ObjRepos\RecError.pas) и вызвать при помощи процедуры:

```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
ReconcileError: EReconcileError): TReconcileAction;
```

В параметры этой функции подставляются параметры метода-обработчика `OnReconcileError`, а возвращает данная функция действие, выбранное пользователем в диалоге (см. рис. 22.4). Таким образом, ее использование очень просто:

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action:
TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Пример "тонкого" клиента

Пример клиентского приложения является частью группы проектов SimpleRemote.bpg и предназначен для взаимодействия с сервером приложений `SimpleAppSrvr` (рис. 22.5), процесс создания которого подробно рассматривался в *гл. 21*.

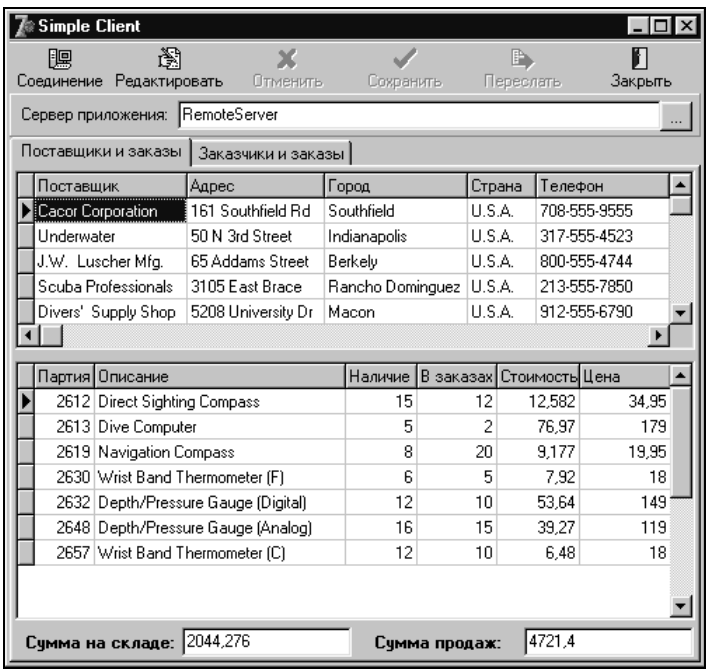


Рис. 22.5. Окно клиентского приложения Simple Client

Проект клиента Simple Client состоит из двух файлов.

- Компоненты, обеспечивающие соединение с удаленным сервером приложения и работу с наборами данных, сосредоточены в модуле данных `DataModule` (файл uDataModule.pas). Обратите внимание, что это "обычный" модуль данных, используемый в приложениях баз данных *(см. гл. 11)*.
- Главная форма клиентского приложения `fmMain` (файл uMain.pas), содержащая визуальные компоненты пользовательского интерфейса.

Листинг 22.1. Секция `implementation` модуля данных `DataModule`

```
implementation

uses uMain, Variants, Dialogs;

{$R *.dfm}

procedure TDM.SrvrConAfterConnect(Sender: TObject);
var i: Integer;
begin
 for i := 0 to SrvrCon.DataSetCount - 1 do
  SrvrCon.DataSets[i].Open;
 cdsVendors.Open;
end;

procedure TDM.SrvrConBeforeDisconnect(Sender: TObject);
var i: Integer;
begin
 for i := 0 to SrvrCon.DataSetCount - 1 do
  SrvrCon.DataSets[i].Close;
 cdsVendors.Close;
end;

procedure TDM.cdsVendorsAfterScroll(DataSet: TDataSet);
begin
 fmMain.edCostSum.Text := VarToStr(cdsParts.Aggregates[0].Value);
 fmMain.edPriceSum.Text := VarToStr(cdsParts.Aggregates[1].Value);
end;

procedure TDM.cdsPartsReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
 cdsParts.CancelUpdates;
 MessageDlg(E.Message, mtError, [mbOK], 0);
end;

end.
```

Соединение клиента с сервером приложения

Для соединения клиентского приложения с сервером в локальной сети использован компонент `SrvrCon` класса `TDCOMConnection`. Данный тип соединения выбран как наиболее простой и требующий лишь наличия локальной сети или даже не требующий ничего — в демонстрационном приложении можно использовать сервер приложения, установленный на этом же компьютере.

Для настройки соединения компонента `SrvrCon` в свойстве `ComputerName` было указано имя компьютера сервера. После этого в списке свойства `ServerName` можно выбрать один из доступных зарегистрированных серверов. В нашем случае это сервер `SimpleAppSrvr.SimpleRDM`, имя которого состоит из имени приложения сервера и имени главного удаленного модуля данных.

Обратите внимание, что в этом же списке имеется и дочерний модуль `Secondary`. Однако для получения доступа к наборам данных дочернего модуля данных мы не будем создавать еще одно соединение, а воспользуемся компонентом `TSharedConnection`, т. к. он специально предназначен для подобных случаев. Для его настройки достаточно указать в свойстве `ParentConnection` компонент соединения. В нашем случае — это `SrvrCon`.

Для компонента `SrvrCon` предусмотрены два метода-обработчика (см. листинг 22.1) — после подключения и перед отключением соединения. В них открываются и закрываются все наборы данных клиентского приложения.

Теперь в клиентском приложении доступны наборы данных обоих удаленных модулей данных сервера приложений.

Непосредственно подключение к серверу осуществляется кнопкой **Соединение**. При ее нажатии выполняется следующий простой код:

```
procedure TfmMain.tbConnectClick(Sender: TObject);
begin
 try
  DM.SrvrCon.Close;
  DM.SrvrCon.ComputerName := edServerName.Text;
  DM.SrvrCon.Open;
 except
  on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
 end;
 SetCtrlState;
end;
```

Соединение закрывается, задается новое имя компьютера сервера, соединение открывается. Специально созданный метод формы `SetCtrlState` управляет доступностью кнопок формы, анализируя текущее состояние наборов данных.

Наборы данных клиентского приложения

Каждый из компонентов TClientDataSet модуля данных DataModule связан с соответствующим компонентом-провайдером сервера.

Компонент cdsOrders предназначен для просмотра данных о заказах. Вспомогательные компоненты cdsEmployees и cdsCustomers содержат списки заказчиков и работников, используемые в главном наборе данных. В компоненте cdsOrders определено агрегатное поле PaidSum, рассчитывающее сумму платежей по всем заказам.

Компонент cdsParts предназначен для просмотра и редактирования данных о поступлениях. Компонент cdsVendors представляет список поставщиков. Так как в сервере приложения связанный с cdsVendors набор данных является главным в отношении "один-ко-многим", то одновременно с обычными полями для компонента cdsVendors автоматически создается поле tblParts типа TDataSetField. Это поле позволяет настроить вложенный набор данных. Для этого достаточно в свойстве DataSetField вложенного компонента cdsParts задать поле tblParts. Теперь при перемещении по записям главного набора данных cdsVendors вложенный набор данных компонента cdsParts будет отображать записи, связанные с текущим поставщиком.

Примечание

В целях сохранения простоты и наглядности исходного кода редактирование предусмотрено только для одного компонента cdsParts. В реальной работе аналогичные методы могут использоваться для всех наборов данных.

Для компонента cdsParts созданы два агрегата, суммирующие данные о поступлениях и продажах. При перемещении по записям этого набора данных в методе-обработчике AfterScroll предусмотрено обновление значений агрегатов (см. листинг 22.1).

Так как компонент cdsParts предназначен и для редактирования данных, то для него необходимо предусмотреть обработку исключительных ситуаций, возникающих не только на клиенте, но и на сервере. Для этого используется метод-обработчик cdsPartsReconcileError (см. листинг 22.1). Сама операция очень проста и скорее служит лишь демонстрацией возможности создавать собственную обработку серверных исключений вместо использования стандартной функции HandleReconcileError (см. рис. 22.4). Здесь все изменения в проблемных записях отменяются методом CancelUpdates и выводится сообщение об ошибке.

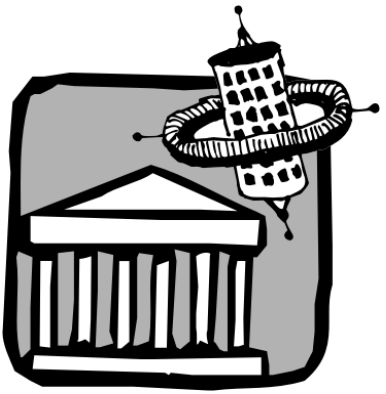
Локальное редактирование, сохранение или отмена изменений для компонента cdsParts выполняется стандартными методами набора данных *(см. гл. 12)*. Дополнительно при отмене изменений используется метод UndoLastChange, позволяющий полностью восстановить последнюю модифицированную запись даже после локального сохранения изменений.

Для передачи изменений серверу использован метод `ApllyUpdates`. Параметр –1 означает, что клиенту будет возвращено сообщение о первой же ошибке.

Резюме

В многозвенных распределенных приложениях в основном используются "тонкие" клиенты, делегирующие большинство функций ПО промежуточного слоя. В трехзвенных приложениях — это сервер приложений.

Основой клиентского приложения является компонент `TClientDataSet`, который инкапсулирует набор данных и обеспечивает его использование при помощи локального буфера. Соединение с удаленным сервером приложений осуществляется при помощи компонентов DataSnap.



ЧАСТЬ VI

Генератор отчетов Rave Reports 5.0

Глава 23. Компоненты Rave Reports и отчеты в приложении Delphi

Глава 24. Визуальная среда создания отчетов

Глава 25. Разработка, просмотр и печать отчетов

Глава 26. Отчеты для приложений баз данных

ГЛАВА 23

Компоненты Rave Reports и отчеты в приложении Delphi

На первый взгляд кажется, что в сфере создания и печати отчетов в Delphi 7 произошла небольшая революция. Просматривая первый раз Палитру компонентов, вы не найдете в ней хорошо знакомой по прошлым версиям Delphi страницы **QReport**. Вместо старого генератора отчетов в состав Delphi 7 включен продукт Rave Reports 5.0 от фирмы Nevrona. "Ну и почему же это событие не дотягивает до революции в отчетах?" — спросит читатель. Авторы могут обосновать свою точку зрения.

Во-первых, компоненты QReport по-прежнему доступны разработчику — пакет DCLQRT70.BPL все так же занимает прочное место в папке \Delphi7\Bin и может быть установлен в Палитру компонентов обычным способом. Да и было бы странно ожидать другого от фирмы Borland, которая бдительно следит за обратной совместимостью приложений. Посмотрите к примеру на страницу **Win 3.1** Палитры компонентов — новые поколения программистов никогда не видели "прабабушку" Windows XP, и все же исторические компоненты занимают свое исконное место!

Во-вторых, схема создания и внедрения отчетов в приложения Delphi практически не изменилась. В Rave Reports имеются и глобальный класс отчета, и классы полос, и компоненты преобразования данных. Существенным нововведением можно считать только визуальную среду создания отчетов, что несомненно облегчит жизнь создателей отчетов и сделает их работу эффективнее и приятнее.

Тем не менее, в Delphi 7 генератор отчетов Rave Reports является основным средством создания отчетов и его компоненты устанавливаются в Палитре компонентов по умолчанию на странице **Rave**. Поэтому главы этой части посвящены разнообразным аспектам разработки отчетов в Rave Reports.

В данной главе рассматриваются следующие вопросы:

- какие компоненты входят в состав Rave Reports и на какие функциональные группы они делятся;

- что такое проект отчета и его структура;
- как включить отчет в состав приложения и какие компоненты для этого необходимы;
- компоненты управления отчетами.

Генератор отчетов Rave Reports 5.0

Генератор отчетов Rave Reports 5.0 разработан фирмой Nevrona и входит в состав Delphi 7 в качестве основного средства для создания отчетов. Он состоит из трех частей:

- ядро генератора отчетов обеспечивает управление отчетом и его предварительный просмотр, и отправку на печать. Исполняемый код ядра сервера включается в приложение Delphi, делая его полностью автономным при работе с отчетами на компьютере клиента;
- визуальная среда разработки отчетов Rave Reports предназначена для разработки самих отчетов. Она позволяет добавлять к отчету страницы, размещать на них графические и текстовые элементы управления, подключать к отчетам источники данных и т. д. Отчеты сохраняются в файлах с расширением rav и должны распространяться совместно с приложениями, использующими их;
- компоненты Rave Reports расположены на странице **Rave** Палитры компонентов Delphi. Они обеспечивают управление отчетами в приложении.

Генератор отчетов устанавливается при инсталляции Delphi в папку \Delphi7\Rave5. Исходные коды компонентов разработчикам в Delphi недоступны.

Безусловно, визуальная среда разработки заметно упрощает процесс создания отчетов и позволяет добиться лучших результатов меньшими усилиями, чем в генераторе отчетов Quick Report, который использовался в предыдущих версиях Delphi. Тем не менее при первом знакомстве с продуктом заметны и его недостатки. Система помощи оставляет тягостное впечатление не только своей крайней лаконичностью, но и фактическими ошибками. Многие свойства и методы остались недокументированными, и наоборот — имеющиеся в статьях подсказки описания не имеют реальных аналогов в коде компонентов.

Однако будем надеяться, что недостатки будут со временем исправлены. А мы займемся детальным знакомством с процессом создания отчетов.

Компоненты Rave Reports и их назначение

Компоненты для создания отчетов и управления расположены на странице **Rave** Палитры компонентов. Они делятся на следующие функциональные группы.

- *Компонент отчета* `TRvProject`, с точки зрения приложения, и есть отчет. Он обеспечивает загрузку заранее созданного в визуальной среде Rave Reports отчета из файла с расширением rav.

  Подробнее об использовании компонента `TRvProject` рассказывается в ниже в этой главе.

- *Компонент управления отчетом* `TRvSystem` обеспечивает работу приложения с отчетом. Взаимодействуя с компонентом отчета, с одной стороны, и сервером отчета Rave Reports, с другой, этот компонент обеспечивает просмотр и печать отчетов.

  Подробнее об использовании компонента `TRvSystem` рассказывается в ниже в этой главе.

- *Компоненты соединения с источниками данных* предназначены для подключения различных источников данных к отчетам. При этом могут использоваться технологии доступа к данным ADO, BDE, dbExpress *(см. часть IV)*.

  К этой группе относятся компоненты:

  - `TRvCustomConnection`;
  - `TRvDataSetConnection`;
  - `TRvTableConnection`;
  - `TRvQueryConnection`.

  Подробнее об использовании этих компонентов рассказывается в *гл. 25*.

- *Компоненты преобразования данных* позволяют конвертировать отчеты из формата данных Rave Reports в другие форматы (текстовый, PDF, HTML, RTF), а также распечатывать или просматривать отчеты.

  К этой группе относятся компоненты:

  - `TRvNDRWriter`;
  - `TRvRenderPreview`;
  - `TRvRenderPrinter`;
  - `TRvRenderPDF`;
  - `TRvRenderHTML`;
  - `TRvRenderRTF`;
  - `TRvRenderText`.

  Подробнее об использовании компонентов преобразования данных рассказывается ниже в этой главе.

Отчет в приложении Delphi

Завершив обзор нового генератора отчетов, давайте обратимся к деталям программирования и посмотрим, что нужно сделать, чтобы приложение могло работать с отчетами.

Основой отчета является файл отчета с расширением rav. Он создается в визуальной среде разработки Rave Reports и может содержать произвольное число страниц. Каждая страница может быть оформлена графическими или текстовыми элементами или отображать данные из какой-либо базы данных. Другими словами, файл RAV — это проект будущего отчета, содержащий общую информацию об отчете, оформление его страниц и правила их заполнения.

После создания проект отчета необходимо связать с приложением Delphi. Для этого используется компонент `TRvProject` (рис. 23.1). Этот компонент обеспечивает представление отчета в приложении.

Но этого недостаточно, чтобы просмотреть или напечатать отчет. Для выполнения этих операций используется код ядра генератора отчета, который автоматически прикомпилируется к исполняемому коду приложения при переносе на любую форму проекта компонентов `TRvProject` и `TRvSystem`. Для управления операциями печати и просмотра в проекте должен присутствовать компонент `TRvSystem` (рис. 23.1).

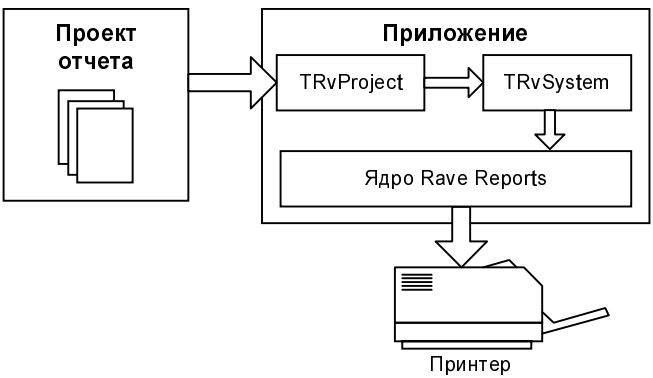


Рис. 23.1. Компоненты Rave Reports в приложении Delphi

Для того чтобы приложение Delphi могло выполнять функции печати отчетов, разработчик должен выполнить следующий набор операций.

1. При помощи визуальной среды разработки Rave Reports необходимо создать проект отчета и сохранить его *(см. гл. 24)*.
2. Перенести в проект приложения в Delphi компонент `TRvProject` и связать его с файлом проекта отчета *(см. ниже)* при помощи свойства `ProjectFile`.

3. Перенести в проект приложения в Delphi компонент `TRvSystem` и связать его с компонентом `TRvProject`. Для этого используется свойство `Engine` компонента `TRvProject` *(см. ниже)*.
4. Написать код приложения, обеспечивающий просмотр и печать отчета (при необходимости и другие операции), используя методы компонента `TRvProject` *(см. ниже)*.

Конечно же, это наиболее простой способ включения отчета в приложения. Для решения более сложных задач необходимо изучить использованные выше компоненты более детально.

Компонент отчета *TRvProject*

Компонент `TRvProject` обеспечивает представление в приложении отчета. Для того чтобы связать проект отчета Rave Reports с компонентом, используется свойство

```
property ProjectFile: string;
```

До начала печати необходимо связать компонент `TRvProject` с компонентом управления отчетом `TRvSystem`. Для этого достаточно передать в свойстве

```
property Engine: TRpComponent;
```

ссылку на компонент `TRvSystem`.

При необходимости вы можете загрузить отчет из внешнего файла или потока:

```
procedure LoadFromFile(FileName: String);
procedure LoadFromStream(Stream: TStream);
```

Загруженный отчет становится текущим.

Кроме этого существует и пара методов для сохранения отчета:

```
function SaveToFile(FileName: String);
procedure SaveToStream(Stream: TStream);
```

В процессе работы приложения может потребоваться напечатать несколько различных отчетов. Для этого можно использовать требуемое число компонентов `TRvProject` или загружать нужные отчеты по мере необходимости.

Забегая немного вперед *(см. гл. 24)*, скажем, что один файл проекта отчета может содержать несколько независимых отчетов. Каждый из них идентифицируется в компоненте `TRvProject` тремя свойствами. Имя, полное имя и описание отчета содержатся соответственно в трех свойствах только для чтения:

```
property ReportName: String;
property ReportFullName: String;
property ReportDesc: String;
```

При этом эти три свойства возвращают параметры текущего отчета. Сразу после загрузки из файла текущим становится отчет, являющийся отчетом по умолчанию в среде разработки. При необходимости сменить текущий отчет используется метод

```
function SelectReport(ReportName: string; FullName: boolean): boolean;
```

В параметре `ReportName` передается имя нужного отчета. Если параметр `FullName` имеет значение `True`, то это полное имя отчета, иначе — имя отчета.

В случае, если проект содержит несколько отчетов, их имена доступны при помощи метода

```
procedure GetReportList(ReportList: TStrings;FullName: boolean);
```

Список имен будет возвращен в список строк `ReportList`, а параметр `FullName` определяет, какие именно имена будут занесены в список. При значении параметра `True` метод возвращает полные имена отчетов (соответствует свойству `ReportFullName`), иначе — имена (соответствует свойству `ReportlName`).

Например, код

```
var ReportList: TStringList;
    i: Integer;
...
 ReportList := TStringList.Create;
 RvProject1.Open;
 try
  RvProject1.GetReportList(ReportList, False);
  for i := 0 to ReportList.Count — 1
   do RvProject1.ExecuteReport(ReportList[I]);
 finally
  RvProject1.Close;
  ReportList.Free;
 end;
```

последовательно печатает все отчеты, входящие в состав файла проекта отчета.

Файл проекта отчета можно включить в состав исполняемого файла приложения. Для этого используется свойство

```
property StoreRAV: Boolean;
```

При щелчке на кнопке в строке этого свойства в Инспекторе объектов открывается специализированный редактор **Load Into exe** (рис. 23.2).

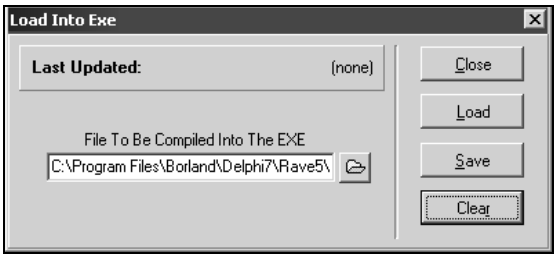


Рис. 23.2. Редактор свойства `StoreRAV` компонента `TRvProject`

Здесь можно задать файл проекта отчета. После этого в Инспекторе объектов в строке свойства `StoreRAV` появятся дата и время загрузки проекта отчета. Это же время и дата будут сохранены в свойстве

```
property RaveBlobDateTime: TDateTime;
```

Отправить отчет на печать можно методом

```
procedure Execute;
```

или же методом

```
procedure ExecuteReport(ReportName: string);
```

который позволяет направить на печать отчет, заданный параметром `ReportName`. Он должен соответствовать имени отчета, хранящемуся в свойстве `ReportName` компонента `TRvProject`.

Отчет, содержащийся в компоненте `TrvProject`, может быть открыт для редактирования методом

```
procedure Open;
```

Не открывая отчет, вы не сможете использовать большинство свойств и методов компонента. Дело в том, что при открытии компонент загружает отчет из файла проекта или прикомпилированного кода (в случае использования свойства `StoreRAV`).

Сохранение и закрытие отчета соответственно выполняются методами

```
procedure Save;
procedure Close;
```

Кроме этого, действия, аналогичные методам `Open` и `Close`, выполняются свойством

```
property Active: Boolean;
```

Если свойству присвоить значение `True` — отчет открывается, иначе — закрывается.

До и после открытия и закрытия отчета вызывается четверка методов-обработчиков:

```
property BeforeOpen: TNotifyEvent;
property AfterOpen: TNotifyEvent;
property BeforeClose: TNotifyEvent;
property AfterClose: TNotifyEvent;
```

Компонент управления отчетом *TRvSystem*

Компонент управления отчетом `TRvSystem` обеспечивает выполнение основных операций с отчетом из приложения. В приложении он должен быть связан с компонентом `TRvProject` *(см. выше разд. "Компонент отчета TRvProject" данной главы)*. Этого вполне достаточно, чтобы компонент `TRvSystem` выполнил свою работу. У разработчика нет необходимости вызывать какие-либо методы компонента, чтобы направить отчет на печать.

В его составе инкапсулированы объекты, обеспечивающие вывод отчета из компонента `TRvProject` в один из трех системных приемников:

- файл (объект класса `TSystemFiler`);
- предварительный просмотр (объект класса `TSystemPreview`);
- принтер (объект класса `TSystemPrinter`).

За это отвечает свойство

```
type
  TReportDest = (rdPreview, rdPrinter, rdFile);
property ReportDest: TReportDest;
```

которое может принимать одно из трех значений типа `TReportDest`.

Соответственно, для каждого типа системного приемника имеется свойство, позволяющее задать все его основные параметры.

Для вывода в файл это комплексное свойство

```
property SystemFiler: TSystemFiler;
```

Внутри него задается имя файла во вложенном свойстве

```
property FileName: string;
```

но при этом вложенное свойство

```
type
  TStreamMode = (smMemory, smTempFile, smFile, smUser);
property StreamMode: TStreamMode;
```

должно иметь значение `smFile`.

При выводе отчета для предварительного просмотра используется экземпляр класса `TSystemPreview`, который доступен через свойство

```
property SystemPreview: TSystemPreview;
```

Его свойства совпадают со свойствами компонента `TRvRenderPreview`.

Стандартное диалоговое окно предварительного просмотра отчета Rave Reports представлено на рис. 23.3.

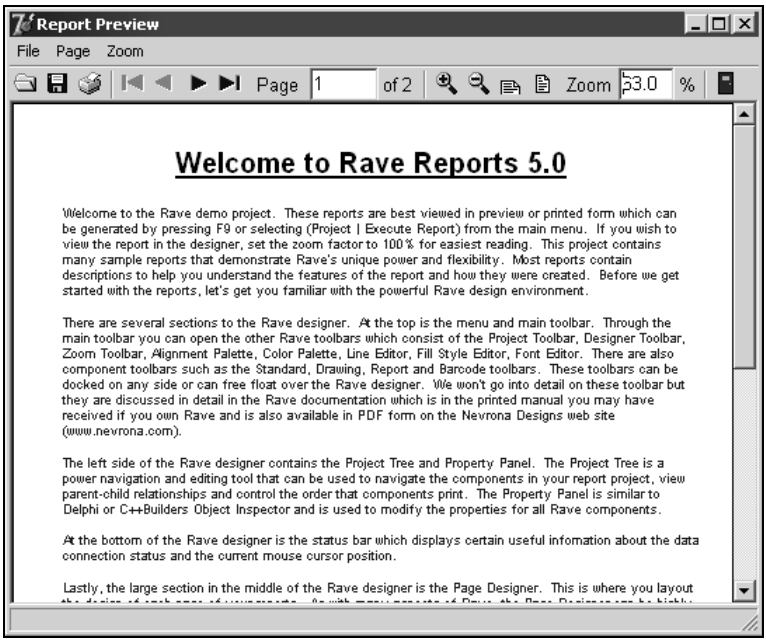


Рис. 23.3. Стандартное диалоговое окно предварительного просмотра компонента `TRvSystem`

Заголовок этого окна задается свойством

```
property TitlePreview: TFormatString;
```

Перед открытием окна предварительного просмотра вызывается метод-обработчик

```
property OnPreviewShow: TNotifyEvent;
```

За вывод отчета на печать отвечает инкапсулированный в компоненте объект типа `TSystemPrinter`. К нему можно обратиться при помощи свойства

```
property SystemPrinter: TSystemPrinter;
```

Его свойства совпадают со свойствами компонента `TRvRenderPrinter`.

Перед тем как отправить отчет одному из трех системных приемников, компонент открывает диалог настройки печати (рис. 23.4).

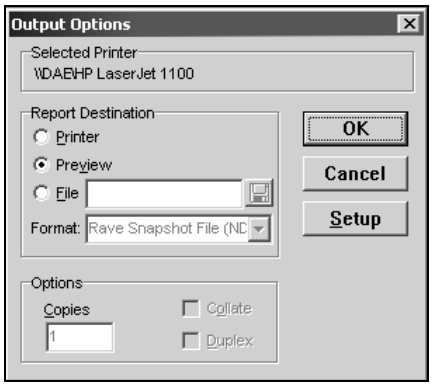


Рис. 23.4. Диалог настройки печати компонента `TRvSystem`

Его заголовок определяется свойством

```
property TitleSetup: TFormatString;
```

Перед открытием этого окна вызывается метод-обработчик

```
property OnPreviewSetup: TNotifyEvent;
```

Кроме этого, для диалога настройки печати можно задать ряд дополнительных параметров. Это делается в свойстве

```
type
  TSystemSetup = (ssAllowSetup, ssAllowCopies, ssAllowCollate,
  ssAllowDuplex, ssAllowDestPreview, ssAllowDestPrinter, ssAllowDestFile,
  ssAllowPrinterSetup);
  TSystemSetups = set of TSystemSetup;
property SystemSetups: TSystemSetups;
```

Элементы множества `TSystemSetup` означают следующее:

- `ssAllowSetup` — разрешает или запрещает использование диалога настройки печати компонента;
- `ssAllowCopies` — управляет доступностью установки числа копий отчета;
- `ssAllowCollate` — разрешает или запрещает настройку режима печати с разбором страниц по копиям;
- `ssAllowDuplex` — разрешает или запрещает настройку двусторонней печати;
- `ssAllowDestPreview` — разрешает или запрещает использование окна предварительного просмотра;

- `ssAllowDestPrinter` — разрешает или запрещает использование принтера;
- `ssAllowDestFile` — разрешает или запрещает использование файла для вывода отчета;
- `ssAllowPrinterSetup` — разрешает или запрещает использование диалога настройки параметров принтера.

Во время выполнения любой из перечисленных операций вывода отчета открывается окно состояния процесса (рис. 23.5). Его заголовок определяется свойством

```
property TitleStatus: TFormatString;
```

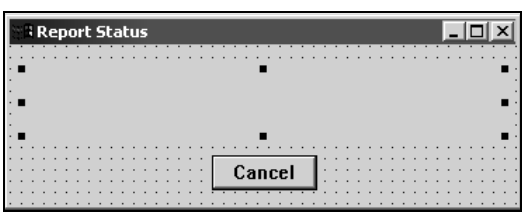


Рис. 23.5. Форма состояния процесса вывода отчета компонента `TRvSystem`

В нем отображается информационная строка состояния, которая может быть настроена при помощи свойств объекта `SystemFiler`, представленного в компоненте `TRvSystem` одноименным свойством.

Вложенное свойство

```
property StatusFormat: string;
```

определяет строку форматирования для текста о состоянии процесса. Для нее предусмотрены следующие управляющие символы:

- `%c` — текущее состояние процесса вывода;
- `%p` — номер текущей страницы;
- `%f` — номер первой страницы;
- `%l` — номер последней страницы;
- `%d` — название устройства вывода (название принтера, имя файла, предварительный просмотр);
- `%r` — имя драйвера устройства вывода;
- `%s` — общее число страниц;
- `%t` — порт печати;
- `%0-%9` — номера строк для свойства `StatusText` *(см. ниже)*.

Вложенное свойство

```
property StatusText: TStrings;
```

позволяет задать до десяти строк (можно задать и больше, но они не будут восприняты строкой статуса) с какой-либо дополнительной информацией, описывающей процесс вывода. Первая строка списка будет выведена при наличии в свойстве `StatusFormat` управляющего символа `%0`, вторая — при наличии символа `%1` и т. д.

При помощи перечисленных свойств вы сможете детально описать процесс вывода отчета. В этом вам помогут методы-обработчики событий компонента `TRvSystem`.

До начала печати отчета и по его окончании (даже если печать была прервана) соответственно вызывается пара методов-обработчиков:

```
property OnBeforePrint: TNotifyEvent;
property OnAfterPrint: TNotifyEvent;
```

В начале печати непосредственно отчета (не заголовка) вызывается метод-обработчик

```
property OnPrint: TNotifyEvent;
```

Если вы печатаете одну страницу, будет вызван метод-обработчик

```
type
  TPrintPageEvent = function(Sender: TObject; var PageNum: Integer):
Boolean;
property OnPrintPage: TPrintPageEvent;
```

Но до начала печати вызывается метод-обработчик

```
property OnNewPage: TNotifyEvent;
```

который обозначает генерацию страницы.

При печати колонтитулов в верхней и нижней частях страницы вызываются методы-обработчики

```
property OnPrintHeader: TNotifyEvent;
property OnPrintFooter: TNotifyEvent;
```

Разработчик может задать несколько опций для всего компонента `TRvSystem`, управляя тем самым процессом вывода отчета. Для это используется свойство

```
type
  TSystemOption = (soUseFiler, soWaitForOK, soShowStatus,
soAllowPrintFromPreview, soPreviewModal);
  TSystemOptions = set of TSystemOption;
property SystemOptions: TSystemOptions;
```

Элементы типа `TSystemOptions` обозначают следующее:

- `soUseFiler` — при установке этой опции в значение `True` вывод будет направляться в файл, заданный свойством `SystemFiler`, независимо от других настроек компонента;
- `SoWaitForOK` — если включить эту опцию, генерация отчета будет задержана до момента, когда пользователь нажмет кнопку **OK** в диалоге настройки печати компонента (см. рис. 23.4);
- `soShowStatus` — эта опция управляет видимостью окна состояния процесса вывода отчета в компоненте;
- `soAllowPrintFromPreview` — будучи включенной, эта опция позволяет печатать отчет из окна предварительного просмотра;
- `soPreviewModal` — при значении `True` делает окно предварительного просмотра модальным.

Резюме

В качестве основного средства создания отчетов и их использования в приложениях в состав Delphi 7 включен генератор отчетов Rave Reports 5.0. В его состав входят ядро генератора отчетов, визуальная среда создания отчетов и набор компонентов.

Ядро генератора отчетов обеспечивает предварительный просмотр или печать отчета. Оно включается в исполняемый файл приложения. Поэтому разработчики избавлены от необходимости распространять совместно с приложением какие-либо дополнительные файлы.

Визуальная среда создания отчетов позволяет разрабатывать самые разнообразные отчеты, в том числе использующие наборы данных из источников различных типов.

Набор компонентов предоставляет разработчику инструментарий для управления отчетом в приложении.

ГЛАВА 24

Визуальная среда создания отчетов

Визуальная среда создания отчетов входит в состав генератора отчетов Rave Reports 5.0. В отличие от генератора отчетов Quick Report, который поставлялся с Delphi 6 и более ранними версиями, визуальная среда в Rave Reports значительно облегчает самый трудоемкий этап в процессе создания отчета и его включения в состав приложения — постраничную разработку шаблона отчета.

Под *шаблоном отчета* мы подразумеваем совокупность страниц отчета с расположенными на них графическими и текстовыми элементами оформления, а также свойствами и правилами создания отчета, сохраненными в файле с расширением rav.

Каждый файл RAV может включать несколько независимых шаблонов отчетов. Затем эти шаблоны используются в компонентах `TRvProject` и служат основой для создания и печати отчетов в приложении.

Кроме этого, средствами визуальной среды шаблон отчета может быть подключен к различным источникам данных. При этом могут использоваться следующие технологии доступа к данным:

- ADO;
- dbExpress;
- BDE.

Визуальная среда Rave Reports открывается из меню **Tools | Rave Designer** главного окна Delphi или при двойном щелчке на компоненте `TRvProject`. Исполняемый файл Rave.exe расположен в папке Delphi7\Rave5.

Подводя итог сказанному, мы можем выделить несколько основных задач, которые можно решать в визуальной среде создания отчетов:

- загрузка, редактирование и сохранение шаблонов отчетов в файлах RAV;
- создание структуры отчета и определение его основных свойств;

- разработка страниц отчета;
- подключение к отчету источников данных и использование этих данных при оформлении страниц отчетов;
- генерация отчета на основе созданного шаблона, его предварительный просмотр или печать.

В этой главе рассматриваются следующие вопросы:

- составные части визуальной среды и инструментарий создания отчетов;
- структура шаблона отчета;
- текстовые и графические элементы оформления страниц отчетов;
- как использовать методы-обработчики событий;
- какие источники данных доступны в отчетах, как их подключить и использовать при оформлении страниц.

Инструментарий визуальной среды создания отчетов

Пользовательский интерфейс визуальной среды создания отчетов Rave Reports во многом напоминает среду разработки Delphi (рис. 24.1). В верхней части окна располагается панель инструментов, состоящая из набора кнопок слева и Палитры инструментов справа. В Палитре инструментов располагаются не только элементы оформления отчетов, но и инструменты для их настройки и управления.

Давайте посмотрим, для чего предназначены закладки Палитры инструментов. Первые четыре содержат элементы оформления отчетов:

- **Drawing** — графические элементы оформления;
- **Bar Code** — различные типы штрихкодов;
- **Standard** — элементы оформления, позволяющие размещать на страницах отчета текст и изображения;
- **Report** — элементы оформления, предназначенные для отображения данных из внешних источников данных, подключенных к отчету.

Остальные закладки содержат инструменты управления и настройки страниц и элементов оформления:

- **Zoom** — управляет увеличением текущей страницы;
- **Colors** — позволяет установить цвета элементов оформления и страниц;
- **Lines** — задает стиль и толщину линий элементов оформления;
- **Fills** — задает стиль заполнения элементов оформления;
- **Fonts** — позволяет задать параметры шрифта для текста;

- **Alignment** — управляет выравниванием элементов оформления на странице.

Центральную часть окна занимает блокнот с двумя закладками.

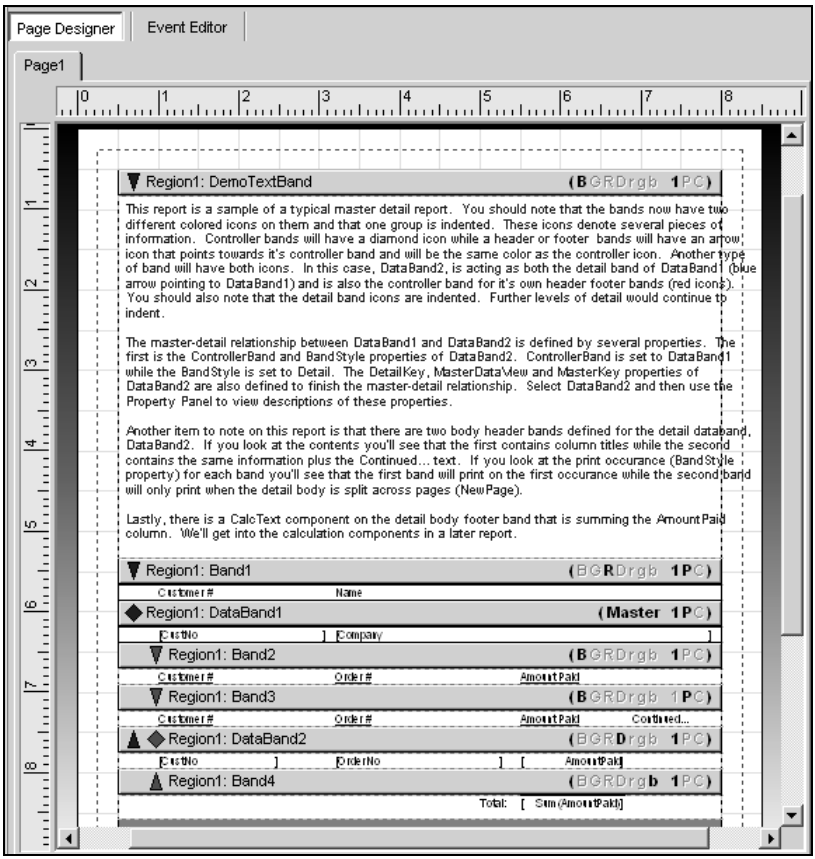


Рис. 24.1. Закладка **Page Designer** визуальной среды создания отчетов Rave Reports

Закладка **Page Designer** содержит еще один блокнот, каждая из страниц которого соответствует одной странице отчета. Когда вы добавляете к отчету новую страницу, здесь появляется еще одна закладка с именем новой страницы. На страницы можно переносить элементы оформления, изменять их размеры и местоположение. На страницу также можно спроектировать измерительную сетку, которая поможет размещать и выравнивать элементы оформления. Обрамляют страницу вертикальная и горизонтальная линейки.

На страницу можно переносить элементы оформления из Палитры инструментов, и затем элементы оформления можно выделять, настраивать их свойства, перемещать и удалять.

Закладка **Event Editor** обеспечивает создание методов-обработчиков событий для отчетов, страниц, элементов оформления и т. д.

Правую часть окна среды разработки занимает панель проекта отчета. Дерево проекта содержит все его составные части. При двойном щелчке на элементе дерева он отображается на странице в центральной части.

В левой части окна среды разработки располагается аналог Инспектора объектов Delphi, в котором доступны свойства текущего элемента. В нижней части этой панели отображается подсказка для текущего свойства.

Проект отчета

Визуальная среда работает с проектом отчета, который создается или загружается из файла с расширением rav. Состав проекта отчета отображается в дереве проекта отчета в панели в правой части окна визуальной среды (рис. 24.2).

Корневой элемент RaveProject содержит три дочерние ветви:

- **Report Library** — библиотека отчетов включает все шаблоны отчетов, содержащиеся в этом проекте;

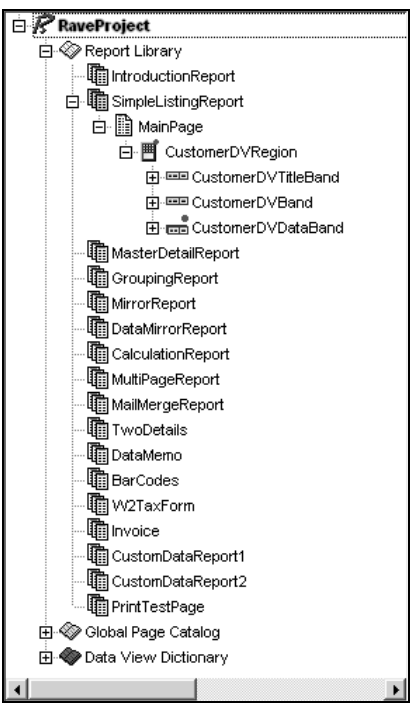


Рис. 24.2. Дерево проекта отчета

- **Global Page Catalog** — каталог глобальных страниц содержит перечень страниц, не принадлежащих какому-либо из отчетов проекта;
- **Data View Dictionary** — словарь просмотров данных содержит созданные соединения с внешними источниками данных.

Теперь давайте посмотрим, из каких составных частей может состоять проект отчета.

Библиотека отчетов

В первую очередь это отчеты, входящие в состав библиотеки отчетов. Каждый из этих отчетов описывает отдельный, самостоятельный отчет. Любой из них может быть загружен в компонент `TRvProject` для использования в приложениях Delphi. Первый отчет в списке по умолчанию становится текущим. Для смены текущего отчета достаточно дважды щелкнуть на нем в дереве проекта и это состояние будет сохранено при закрытии проекта.

Для того чтобы добавить к проекту новый отчет, можно использовать кнопку на главной панели окна визуальной среды или команду **File | New Report** главного меню. Для удаления отчета достаточно сделать его текущим и нажать клавишу <Delete>.

Для каждого отчета необходимо заполнить свойства `Name` и `FullName`, которые используются для идентификации отчета при работе с ним в Delphi *(см. гл. 23)*. Кроме этого, в свойстве `Description` полезно заполнить описание отчета и задать единицы измерения, т. к. по умолчанию установлено использование дюймов.

Отчет может содержать произвольное число страниц. Напечатать можно как все страницы, так и их произвольное подмножество. Для разработчика список страниц отчета, которые предлагаются к печати по умолчанию, доступен в свойстве `PageList`. С этим свойством связан редактор страниц **Page List Editor** (рис. 24.3), который позволяет выбирать страницы отчета и формировать из них список для печати. При этом одна страница может быть включена в список несколько раз.

Для добавления к текущему отчету новой страницы используйте кнопку на главной панели окна визуальной среды или команду **File | New Report Page** главного меню. Для удаления выберите страницу и нажмите клавишу <Delete>.

Страница имеет имя, задаваемое свойством `Name`, а также несколько свойств, задающих ее важнейшие параметры: `Orientation`, `PageSize`, `PageHeight`, `PageWidth`.

Свойство `GotoPage` позволяет задать страницу, которая будет напечатана после этой. Порядок печати страниц по умолчанию соответствует их порядку в дереве отчета.

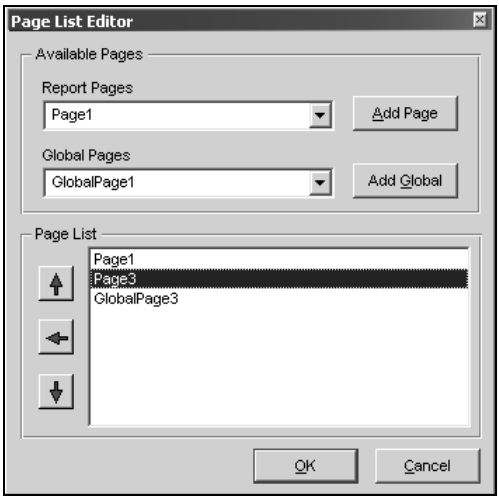


Рис. 24.3. Редактор страниц отчета **Page List Editor**

Свойство `GridLines` позволяет задать плотность измерительной сетки, накладываемой на страницу в визуальной среде для удобства размещения элементов оформления.

Каталог глобальных страниц

Каталог глобальных страниц объединяет страницы, доступные из любого отчета библиотеки отчетов. Таким образом вы можете оформить все отчеты проекта одинаково. Например, для всех отчетов можно создать глобальные страницы титульного листа, общего заголовка и т. д.

Добавить новую страницу можно при помощи команды главного меню **File | New Global Page**. После этого страница появляется в списке каталога и доступна для редактирования.

Глобальная страница добавляется в отчет при помощи редактора страниц (см. рис. 24.3). Для этого необходимо выбрать нужную страницу из списка **Global Pages** и нажать на кнопку **Add Global**.

Словарь просмотров данных

Словарь просмотра данных объединяет разнообразные объекты доступа к данным. Для создания нового объекта необходимо воспользоваться командой главного меню **File | New Data Object**. После этого открывается диалог выбора типа объекта **Data Connections** (рис. 24.4).

Здесь доступны следующие типы объектов:

- **Data Lookup Security Controller** — организует аутентификацию пользователя по имени и паролю при использовании одного из просмотра данных;

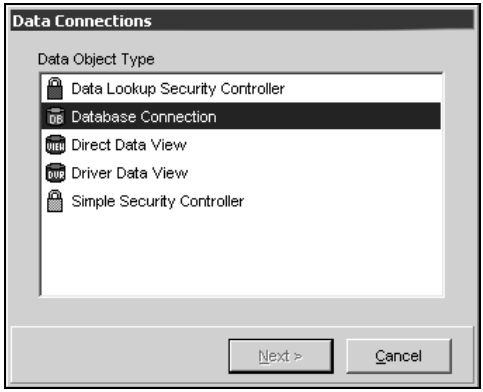


Рис. 24.4. Диалог выбора типа объекта доступа к данным **Data Connections**

- **Database Connection** — создает соединение с внешним источником данных на основе одной из трех технологий доступа к данным: ADO, BDE, dbExpress;
- **Direct Data View** — создает просмотр данных на основе активного соединения с источником данных;
- **Driver Data View** — создает просмотр данных на основе ранее созданного в словаре соединения;
- **Simple Security Controller** — представляет собой список пользователей, который может быть использован для организации доступа в отдельных отчетах.

Созданные в словаре объекты являются глобальными для всего проекта и доступны на любой странице любого отчета. Мы познакомимся с ними более подробно дальше в этой главе.

Стандартные элементы оформления и их свойства

Теперь остановимся подробнее на элементах оформления отчетов. Они используются так же, как и компоненты в Delphi. Выбранный элемент переносится из Палитры инструментов на страницу отчета. Здесь его можно разместить в нужном месте, изменить его размеры и настроить свойства. Свойства элемента оформления доступны на панели слева. Кроме этого, наиболее важные визуальные свойства (цвет, стиль линий и заполнения и т. д.) вынесены на Палитру инструментов.

После переноса на страницу имя элемента оформления также появляется в дереве проекта.

Элементы для представления текста и изображений

На странице **Standard** Палитры инструментов расположены элементы оформления, предназначенные для отображения текста и изображений. Рассмотрим их.

Для представления однострочного текста имеется простой элемент оформления `Text`. Текст задается свойством `Text`. Другие стандартные свойства позволяют настраивать шрифт, цвет и т. д. Кроме этого, свойство `Rotation` позволяет повернуть текст на любой угол в диапазоне от 0 до 360 градусов.

Для представления многострочного текста используется элемент оформления `Memo`. Текст задается свойством `Memo`.

Если вам необходимо объединить несколько элементов оформления в группу (например несколько строк текста и изображений в заголовке страницы) и использовать их на странице совместно, применяется невизуальный элемент оформления `Section`. Размещенные на нем другие элементы как бы оказываются на самостоятельной странице, ограниченной элементом `Section`. Вы можете выделять и перемещать отдельные элементы, но делать это только внутри секции. А при перемещении секции по странице все расположенные на ней элементы также перемещаются вместе с ней. Выравнивание элементов тоже работает по границам секции.

Для представления изображений используется элемент оформления `Bitmap`. Он позволяет оформлять отчеты изображениями, сохраненными в файлах в формате BMP. Для загрузки изображения используется диалог, открывающийся при щелчке на кнопке свойства `Image`. Также при помощи свойства `FileLink` можно связать элемент с файлом изображения и при печати отчета оно будет загружено. Кроме этого, изображение можно загрузить из базы данных. Для этого используются свойства `DataView` (определяет объект просмотра данных) и `DataField` (задает поле изображения).

Изображение можно масштабировать. Для этого используется свойство `MatchSide`. При его значениях `msHeight`, `msWidth`, `msBoth` изображение соответственно масштабируется по горизонтали, вертикали или в двух измерениях. Естественно, при этом может произойти искажение изображения. А вот при значении `msInside` изображение будет масштабировано пропорционально.

Аналогичными свойствами обладает элемент оформления `MetaFile`. Но изображение в формате WMF загружается в него при помощи свойства `FileLink` или `DataField`.

Невизуальный элемент оформления `FontMaster` позволяет задать единые свойства шрифта для группы элементов оформления. Его свойство `Font`

определяет шрифт. И этот шрифт будет использоваться всеми элементами, в чьих свойствах FontMirror будет указан данный элемент FontMaster.

Невизуальный элемент оформления PageNumInit обеспечивает нумерацию страниц, начиная с той, на которой он расположен. Свойство InitValue задает номер, с которого начинается отсчет нумерации. А при помощи свойств InitDataView и InitDataField можно загрузить это значение из базы данных.

Графические элементы управления

Графические элементы оформления расположены на странице **Drawing** Палитры инструментов. Конечно, с их помощью вам не удастся изобразить картину Сальвадора Дали, но для оформления отчетов и рисования таблиц они вполне подойдут.

Это основные три элемента оформления для рисования линий:

- HLine — горизонтальная линия;
- VLine — вертикальная линия;
- Line — универсальная линия.

Все эти элементы имеют идентичный набор свойств, позволяющих задавать параметры и размер линий. Кроме этого, элемент Line позволяет развернуть линию на любой угол. Это можно сделать при помощи мыши.

Элементы оформления Square и Rectangle изображают квадрат и прямоугольник соответственно.

Элементы Ellipse и Circle изображают эллипс и круг.

Штрихкоды

На странице **Bar Code** разработчику доступны шесть элементов оформления, позволяющие включать в отчеты штрихкоды. Все они реализуют различные стандарты, но значение для кодирования у всех задается одним свойством Text. Элементы PostNetBarCode, I2of5BarCode, UPCBarCode и EANBarCode позволяют вводить только числа, а элементы Code39BarCode и Code128BarCode могут работать и с буквенно-цифровыми последовательностями.

- Код PostNet используется почтовой службой США, содержит код адреса.
- Перемежающийся код I2of5 служит для представления числовых последовательностей. Перемежающимся назван потому, что цифры в последовательности попеременно кодируются штрихами и пробелами.
- Код Code39 предназначен для кодирования цифр, заглавных букв латинского алфавита и некоторых других символов. Для представления символа используются пять штрихов и четыре пробела.

- Код Code128 позволяет хранить первые 128 символов ASCII.
- Код UPC (Universal Product Code) может содержать только цифры. Разработан для маркировки продуктов. Код может включать 12 цифр.
- Код EAN (European Article Numbering system) подобен UPC. Код может включать 13 цифр. Первые две отводятся под код страны-производителя.

Для всех элементов значение для кодирования можно загрузить из базы данных при помощи свойства `FileLink` или `DataField`.

При необходимости можно рассчитать и напечатать контрольную сумму. Свойство `UseCheckSum` при значении `True` рассчитывает ее, а свойство `PrintCheckSum`, будучи установленным в значение `True`, печатает.

Штрихкод можно развернуть, но только с дискретностью 90°. Для этого используется свойство `BarCodeRotation`.

Обработка событий

Каждому отчету, странице или элементу оформления можно назначить один или несколько методов-обработчиков событий. Для этого используется Редактор событий **Event Editor**, доступный через одноименную закладку в центральной части окна визуальной среды Rave Reports.

Для текущего элемента здесь можно выбрать из списка **Avaiable Events** одно из доступных для обработки событий. При этом будет создан обработчик события, программный код которого вводится в окне редактора внизу. В списке **Defined Events** отображаются события, обрабатываемые текущим элементом.

Синтаксис кода, используемый для обработки событий, аналогичен синтаксису Object Pascal. При этом можно применять некоторые процедуры и функции Delphi.

Например, в метод-обработчик события `BeforePrint` элемента оформления `Text1` можно поместить следующий код:

```
Text1.Text := IntToStr(StrToInt(Text1.Text) + 1);
```

который обеспечит нумерацию страниц отчета.

Проверка созданного кода выполняется при нажатии кнопки **Compile**.

Внешние источники данных в отчете

Все объекты, обеспечивающие доступ к внешним источникам данных из отчетов проекта, собраны в словаре просмотра данных **Data View Dictionary**. Новый объект создается командой **File | New Data Object** главного меню.

Соединение с источником данных и просмотры

В первую очередь необходимо создать соединение с источником данных. Для этого в диалоге (см. рис. 24.4) необходимо выбрать объект соединения **Database Connection** и, после нажатия кнопки **Next**, в следующем окне выбрать одну из трех возможных технологий доступа к данным: ADO, dbExpress, BDE. В зависимости от сделанного выбора настраиваются параметры соединения. Подробнее о технологиях доступа к данным рассказывается в главах *части IV*.

После завершения настройки готовое соединение появляется в списке **Data View Dictionary**.

На втором этапе к работающему соединению можно подключать объекты просмотра данных. В диалоге **Data Connections** (см. рис. 24.4) надо выбрать тип объекта доступа к данным **Driver Data View** и нажать на кнопку **Next**. Затем в следующем окне нужно выбрать одно из существующих соединений *(см. выше)*. После этого появляется диалог **Query Advanced Designer** (рис. 24.5).

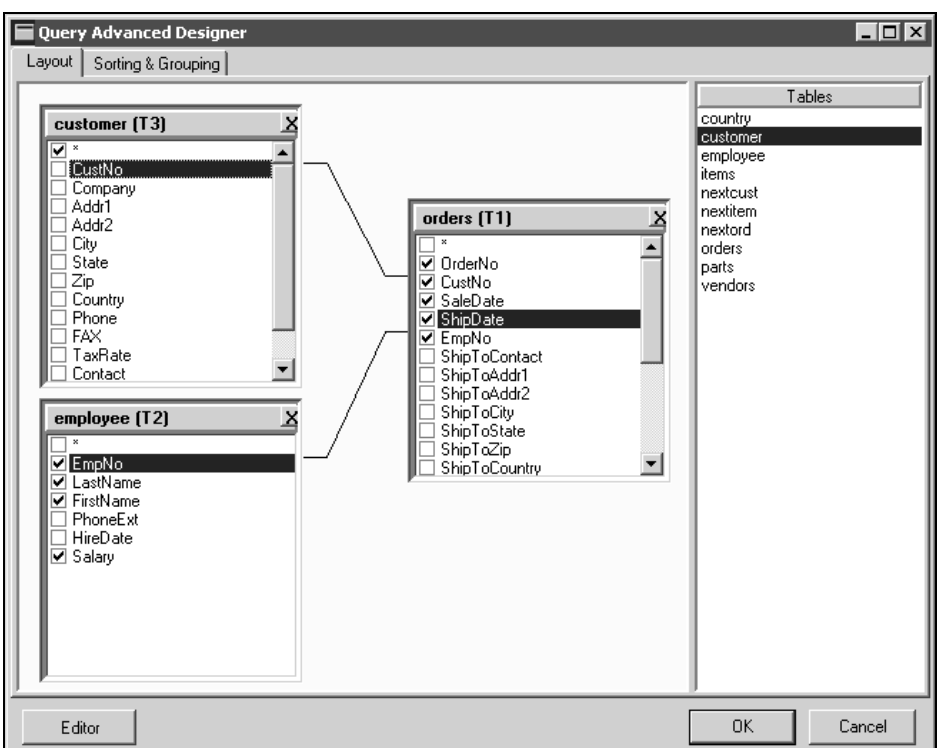


Рис. 24.5. Диалог **Query Advanced Designer**, позволяющий настроить объект просмотра данных

В этом диалоговом окне нужно выбрать из списка доступных таблиц источника данных необходимые и задать требуемые соотношения между полями. Страница **Layout** двухстраничного диалога позволяет выбирать таблицы. Нужная таблица выбирается из списка **Tables** в правой части и перетаскивается в левую часть, где каждая таблица представляется в виде списка полей. Необходимые поля также можно выбирать для будущего использования. Перетаскиванием полей между таблицами можно задавать внешние ключи.

Страница **Sorting & Gouping** позволяет выбрать поля для сортировки и группировки. Здесь в списке слева представлены выбранные ранее таблицы и поля. Поля можно переносить в список **Sort Fields** справа. Поля в этом списке будут использованы для сортировки наборов данных. А из полей для сортировки можно выбрать поле для группировки. Для этого используется список **Group By**.

При нажатии на кнопку **Editor** появляется редактор с текстом запроса SQL, реализующий все сделанные ранее настройки. При необходимости его можно исправить вручную.

После завершения настройки просмотра новый объект просмотра появляется в словаре просмотра данных **Data View Dictionary** и может использоваться в отчетах.

Созданный запрос просмотра доступен через его свойство `Query`.

Безопасность доступа к данным

На третьем этапе созданное соединение и просмотр можно "защитить". Для этого из списка в диалоговом окне **Data Connections** (см. рис. 24.4) выбирается объект аутентификации **Data Lookup Security Controller**. Он сразу же появляется в словаре просмотра данных **Data View Dictionary**. Остается только подключить его к нужному просмотру. Для этого используется его свойство `DataView`, в котором необходимо задать требуемый объект просмотра. Списки имен пользователей и их пароли задаются свойствами `UserField` и `PasswordField` соответственно.

Объект аутентификации можно подключить и для использования в отдельном отчете. Для этого применяется свойство `SecurityControl` отчета, в котором указывается нужный объект.

Еще один объект — простой объект аутентификации (объект **Simple Security Controller** в диалоговом окне **Data Connections**) — в свойстве `UserList` содержит список имен и паролей в формате `UserName=Password`. Он может использоваться только в отчетах.

Отображение данных в отчетах

Для представления данных в отчетах предназначены специализированные элементы оформления, представленные на странице **Report** Палитры инструментов.

Они делятся на две функциональные группы.

В первую группу выделены элементы, обеспечивающие размножение строк в отчете, их заполнение данными, а также группировку при создании сложных отчетов. Эти элементы мы будем называть *структурными*.

Во второй группе объединены элементы оформления, созданные на основе стандартных и обеспечивающие отображения текущего значения конкретного поля таблицы просмотра.

Структурные элементы отчета

Рассмотрим структурные элементы.

Основой отчета, использующего просмотры баз данных, является элемент `Region`. Он создает в отчете область, предназначенную для размещения любых других элементов и определяющую часть страницы отчета, отведенную под отображение данных. Он обладает одним интересным свойством `Columns`, которое задает число колонок, в которых будет печататься отчет.

При создании отчета, использующего базу данных, этот элемент переносится на страницу в первую очередь. Затем приходит очередь элементов `Band` и `DataBand`.

Элемент `Band` создает полосу, на которой можно располагать стандартные элементы оформления. Он служит для оформления заголовков, сносок, врезок и других статичных фрагментов оформления отчетов, которые не изменяются при печати просмотра данных.

Элемент `DataBand` создает полосу, моделирующую строку просмотра данных. На ней располагаются элементы отображения данных, которые будут рассмотрены ниже. При печати отчета для каждой строки печатается новый экземпляр полосы элемента `DataBand` со всеми расположенными на ней элементами оформления. Таким образом и получается отчет, отображающий строка за строкой весь просмотр данных.

Важнейшее свойство `BandStyle` определяет роль и поведение полосы в отчете. С ним связано диалоговое окно **Band Style Editor** (рис. 24.6), которое отображает взаимосвязь полос в области `Region` отчета и позволяет задать поведение текущей полосы.

В левой части диалога отображается список всех полос отчета с их взаимосвязями (отношениями "один-ко-многим", группировкой, вложенностью

и т. д.), текущая полоса выделяется жирным шрифтом с подчеркиванием. Имя каждой полосы отображается трижды. И это не ошибка разработчиков, а желание показать, что каждая полоса размножается для печати записей просмотра данных.

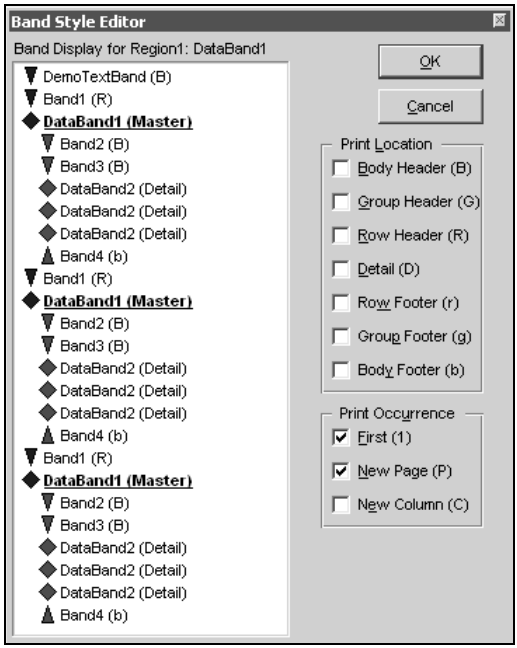


Рис. 24.6. Редактор полос отчета **Band Style Editor**

Группа флажков **Print Location** в правой части диалогового окна определяет назначение полосы. А группа **Print Occurrence** задает, в каком месте отчета появляется полоса:

- **Body Header (B)** — заголовок отчета, печатается в начале отчета;
- **Group Header (G)** — заголовок группы, печатается в начале группы записей, объединенных в просмотре данных выражением `GROUP BY`;
- **Row Header (R)** — заголовок записи, печатается в начале каждой записи просмотра данных;
- **Detail (D)** — печатается в начале подчиненного набора записей, входящего в отношение "один-ко-многим";
- **Row Footer (r)** — окончание строки, печатается в конце каждой записи просмотра данных;
- **Group Footer (g)** — окончание группы, печатается в конце группы записей, объединенных в просмотре данных выражением `GROUP BY`;

- **Body Footer (r)** — окончание отчета, печатается в конце отчета;
- **First (1)** — печатается один раз в начале отчета (титул отчета);
- **New Page (P)** — печатается в начале каждой страницы отчета;
- **New Column (C)** — печатается в начале каждой колонки отчета.

Примечание

Для каждого из перечисленных выше типов в скобках указан символ, который используется для обозначения типа полосы на странице отчета в визуальной среде Rave Reports (рис. 24.7). Таким образом, по совокупности символов разработчик может оценить роль той или иной полосы в отчете, не обращаясь к редактору.

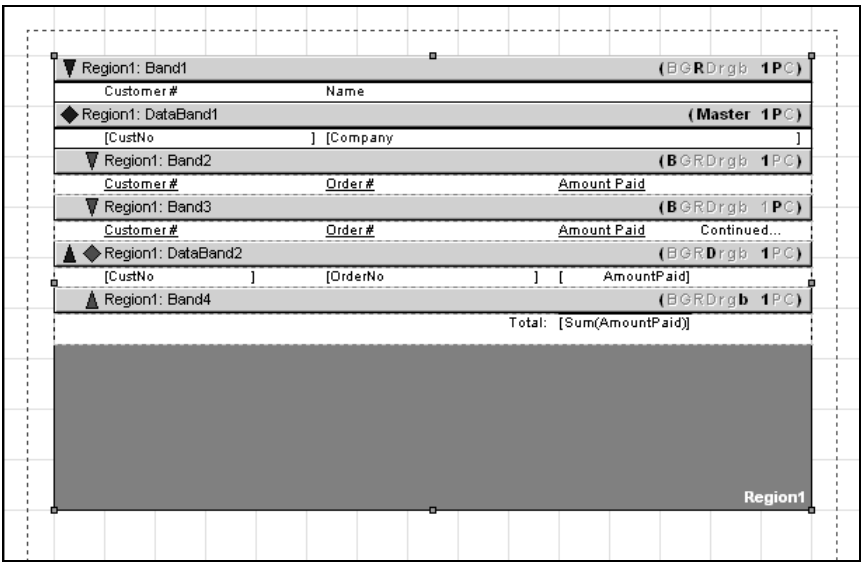


Рис. 24.7. Элементы `Band`, `DataBand` и `Region` на странице отчета

Другие свойства полос и способы создания простых и сложных отчетов рассматриваются в *гл. 26*.

Элементы отображения данных

Элементы отображения данных представляют собой модифицированные стандартные элементы, размещаются на структурных элементах отчета и отображают данные из связанных с ними полей просмотра данных. Они расположены на странице **Report** Палитры инструментов.

- Элемент `DataText` предназначен для представления строковых или числовых значений полей связанного просмотра данных.

- Элемент `DataMemo` используется при необходимости показать данные в формате Memo или BLOB.
- Элемент `CalcText` обеспечивает выполнение одной из агрегатных функций над значениями связанного поля и представление результата. Тип операции выбирается в свойстве `CalcType`.
- Невизуальный элемент `DataMirrorSection`, так же как и его предок `Section`, объединяет группу других элементов для совместного использования.

Кроме перечисленных элементов, еще один элемент способен отображать данные из поля просмотра. Это стандартный элемент оформления `Bitmap` со страницы **Standard** Палитры инструментов

Все перечисленные элементы (в том числе и элемент `Bitmap`) связываются с просмотром данных и полем одинаково.

- Свойство `DataView` определяет, какой просмотр данных используется элементом.
- Свойство `DataField` задает поле просмотра, значения которого будут отображаться элементом.

Детально использование компонентов отображения данных рассматривается в *гл. 26*.

Резюме

Визуальная среда Rave Reports позволяет создавать проекты отчетов. Каждый такой проект может содержать несколько отчетов.

Инструментарий визуальной среды позволяет конструировать страницы отчетов из элементов оформления, настраивать их свойства и создавать обработчики событий.

При помощи набора объектов доступа к данным к отчету можно подключить внешний источник данных. При этом можно использовать одну из трех технологий доступа к данным: ADO, dbExpress, BDE. Специализированный редактор предназначен для визуального построения просмотра данных, а набор элементов отображения данных, используя поля созданного просмотра, обеспечивает создание отчетов.

ГЛАВА 25

Разработка, просмотр и печать отчетов

Мы уже обсуждали в предыдущих главах визуальную среду создания отчетов Rave Reports и набор компонентов Delphi на странице **Rave** Палитры компонентов, предназначенный для интегрирования отчета в приложение Delphi. В этой главе мы обратимся к практике разработки и использования отчетов Rave Reports в приложениях Delphi.

Рассматриваемые в данной главе примеры не отличаются изощренностью оформления и сложностью, но позволяют поэтапно проследить всю методику создания отчетов и использования их в приложениях и обладают основными атрибутами стандартных отчетов. В них имеются заголовки, нумерация страниц, выводится текстовая и графическая информация.

Но сам по себе отчет — это всего лишь шаблон, который необходимо включить в приложение. Для этого применяются компоненты Rave Reports, которые помимо этой важной функции реализуют еще несколько очень полезных операций. Это, например, возможность сохранения и загрузки отчета в файлах и преобразование типов данных отчета из базового формата RAV в наиболее популярные форматы.

Вопросы создания отчетов для баз данных здесь не затрагиваются и обсуждаются в следующей главе.

В этой главе рассматриваются следующие вопросы:

- разработка проекта отчета в визуальной среде Rave Reports;
- использование компонентов `TRvProject` и `TrvSystem`;
- просмотр и печать отчета;
- использование в отчете внешних файлов;
- преобразование форматов данных при помощи компонентов Rave Reports.

Этапы создания отчета и включение его в приложение

Процесс создания отчета с использованием генератора отчетов Rave Reports состоит из трех этапов. Первый выполняется в визуальной среде Rave Reports, второй и третий — в среде разработки Delphi.

1. На первом этапе в визуальной среде Rave Reports создается проект отчета и в нем необходимые страницы, объекты доступа к данным *(см. гл. 24)*. На страницах располагаются элементы оформления, при необходимости к ним подключаются просмотры данных (если отчет отображает информацию из таблиц базы данных) и объекты аутентификации пользователей отчетов. Создаются обработчики событий. И в завершение этапа готовый проект отчета сохраняется в файле с расширением rav.
2. На втором этапе в проект приложения в Delphi переносятся компоненты `TRvProject` и `TRvSystem` *(см. гл. 23)* со страницы **Rave** Палитры компонентов. При этом в состав приложения автоматически включается ядро генератора отчета. Первый компонент связывается с файлом проекта отчета и представляет в приложении отчет со всеми его свойствами, страницами, элементами оформления и т. д. Второй компонент связывается с первым, взаимодействует с ядром генератора отчетов и обеспечивает печать отчета из приложения.
3. На третьем этапе создается программный код, обеспечивающий выполнение функций приложения, связанных с отчетом. Наряду со стандартными операциями предварительного просмотра и печати отчета это могут быть загрузка из файла и сохранение в файле, преобразование формата данных отчета, изменение содержания отчета в зависимости от выполненных пользователем действий и т. д. На этом этапе используются свойства, методы и методы-обработчики событий компонентов `TRvProject` и `TRvSystem` и других компонентов со страницы **Rave** Палитры компонентов.

Далее в этой главе мы подробно рассмотрим перечисленные этапы и различные аспекты программирования, связанные с ними.

Простой отчет в визуальной среде Rave Reports

При описании первого этапа наша задача — описать возможности проекта RAV, исследовать его структуру и составные части, которые можно использовать в приложении Delphi. Собственно элементы оформления просты в использовании и мы акцентируем внимание лишь на нескольких элементах, требующих небольших пояснений.

Для начала создадим в визуальной среде Rave Reports новый проект (команда **File | New** главного меню). Обратите внимание, что по умолчанию вместе с проектом создается первый отчет `Report1` с одной страницей `Page1`. Его мы и используем, переименовав в `rptXP`. Проекту присвоим имя `SimpleDemo`.

Нумерация страниц отчета

В первую очередь создадим заголовок страницы и включим механизм нумерации страниц. Для этого перенесем на страницу элемент `Section` со страницы **Standard** Палитры инструментов и поместим его в верхней части страницы отчета. Эта секция будет объединять элементы оформления заголовка.

Поместим в секции элемент `DataText` — нам необходимо его свойство `DataField`. Редактор свойства **Data Text Editor** (рис. 25.1) позволяет настраивать свойство так, чтобы элемент мог отображать разнообразные данные. Сейчас нас интересуют номера страниц отчета.

Из списка **Report Variables**, который содержит глобальные переменные отчета, необходимо выбрать переменную `RelativePage`. Затем нужно щелкнуть на кнопке **Insert Report Var** и переменная `Report.RelativePage` появится в поле **Data Text** в нижней части диалога. Эта переменная при печати отчета будет содержать порядковый номер текущей страницы.

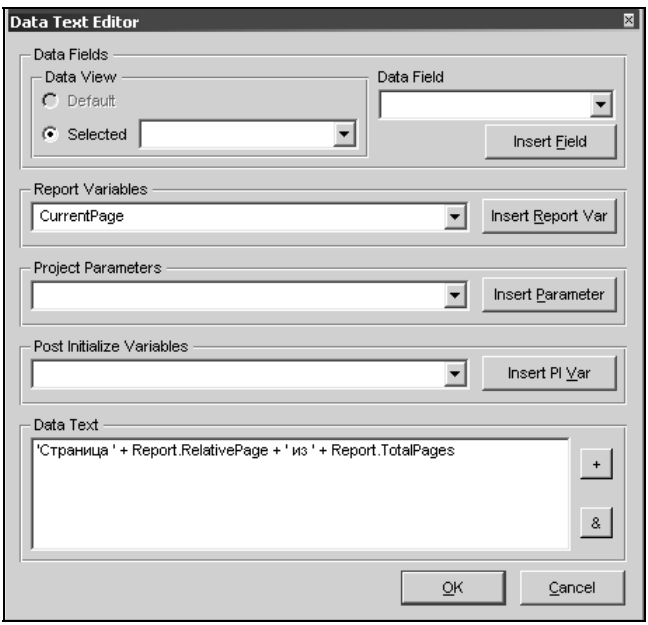


Рис. 25.1. Редактор свойства `DataField` элемента `DataText`

Аналогичным образом добавим переменную `TotalPages`, которая возвращает общее число страниц отчета. Затем вручную отредактируем текст в поле **Data Text**:

```
'Страница ' + Report.RelativePage + ' из ' + Report.TotalPages
```

Шаблон номера страницы готов. Но для того, чтобы механизм нумерации заработал, необходимо перенести на страницу невизуальный элемент `PageNumInit`. Он будет работать автоматически. Единственное, что нужно сделать, — это настроить свойство `InitValue`, в котором задается номер начальной страницы.

Примечание

Обратите внимание, что элемент `PageNumInit` должен быть только один и располагаться на первой странице, которая должна содержать свой номер. Иначе, на каждой странице, на которой есть такие элементы, нумерация начнется сначала.

Использование элемента *FontMaster*

Невизуальный элемент `FontMaster` позволяет использовать одинаковые шрифты в элементах оформления, например, в пределах одной секции. Для этого необходимо перенести в секцию элемент `FontMaster` и задать в его свойстве `Font` нужные характеристики шрифта. После этого во всех элементах, которые будут использовать этот шрифт, в списке свойства `FontMirror` надо выбрать этот элемент `FontMaster`.

В результате, один раз настроив шрифт, можно применить его для любого числа элементов оформления.

Добавление страниц к отчету

После создания первой страницы к отчету необходимо добавить еще две страницы и оформить их по образцу первой. Здесь все операции рутинны и не требуют дополнительных пояснений. В результате дерево проекта для отчета `rptXP` выглядит так, как показано на рис. 25.2.

Кроме обычных страниц, принадлежащих отчету, в Rave Reports можно создавать глобальные страницы, которые можно связать с любым отчетом проекта (например, титульные страницы). Для создания новой глобальной страницы используется команда **File | New Global Page** главного меню. Затем процесс разработки не отличается от обычных страниц.

Теперь для того, чтобы при просмотре или печати отчета отображались все страницы, а не только первая, необходимо дополнительно настроить свойство `PageList` отчета. В редакторе свойства необходимо перенести в список **Page List** все нужные страницы. Для этого страница выбирается из выпа-

дающего списка **Report Pages**. Затем нужно щелкнуть на кнопке **Add Page**. Аналогичная операция выполняется и для глобальных страниц проекта, доступных в списке **Global Pages**.

Результат для отчета `rptXP` представлен на рис. 25.3.

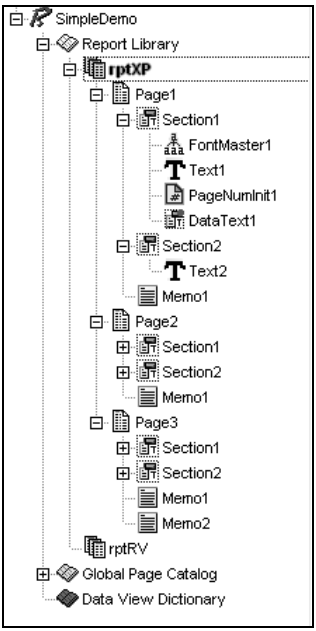


Рис. 25.2. Дерево проекта `SimpleDemo` для отчета `rptXP`

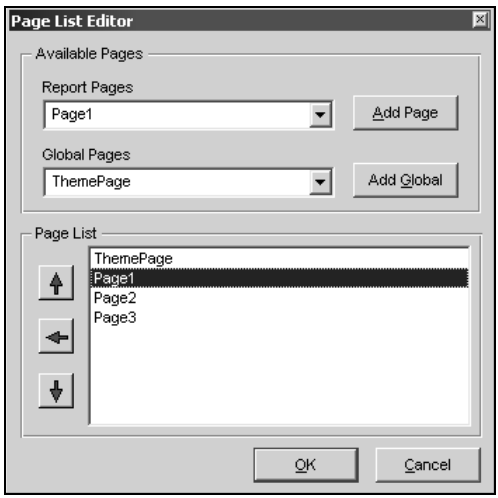


Рис. 25.3. Редактор свойства `PageList` отчета `rptXP`

Используя кнопки слева в группе **Page List**, можно изменять порядок следования страниц при печати отчета или удалять страницы из списка.

Теперь отчет `rptXP` готов. Дополнительно к нему, с использованием тех же элементов и действий, в рассматриваемом нами примере создан еще один отчет `rptRV`. При этом по умолчанию текущим считается отчет, который при последнем сохранении проекта был текущим.

Проект отчета сохранен в файле SimpleDemo.RAV.

Отчет в приложении

Теперь, когда проект SimpleDemo.RAV с двумя отчетами готов, перейдем к разработке приложения в Delphi.

Любое приложение, использующее генератор отчетов Rave Reports, должно иметь как минимум пару компонентов — `TRvProject` и `TRvSystem`. Первый из

них есть проект отчета в приложении. С его помощью разработчик получает доступ к отчетам проекта и их свойствам. Второй компонент обеспечивает использование ядра генератора отчетов Rave Reports при печати или предварительном просмотре отчета (см. рис. 23.1). Подробнее о свойствах и методах этих компонентов *см. гл. 23*.

При использовании этих компонентов в состав исполняемого кода приложения автоматически включается ядро генератора отчетов Rave Reports. Соответственно при распространении приложения не требуются дополнительные файлы — даже файл проекта отчета можно включить в приложение *(см. ниже)*.

Компонент `TRvProject` необходимо связать с файлом проекта SimpleDemo.RAV. Для этого используется его свойство

```
property ProjectFile: string;
```

Файл RAV можно распространять вместе с приложением или включить его в состав исполняемого файла. Для этого используется свойство

```
property StoreRAV: boolean;
```

компонента (см. рис. 23.2).

Перед использованием отчетов из компонента `TRvProject` его необходимо открыть. В нашем примере при открытии формы приложения компонент открывается и в список считываются полные имена отчетов проекта и отображается описание текущего отчета:

```
procedure TfmMain.FormShow(Sender: TObject);
begin
 rpProject.Open;
 rpProject.GetReportList(lbxRptList.Items, True);
 rpProject.ReportDescToMemo(meDesc);
end;
```

Но только один из отчетов доступен для использования одновременно. Для смены текущего отчета можно воспользоваться методом

```
function SelectReport(ReportName: string; FullName: boolean): boolean;
```

Для идентификации текущего отчета компонент `TRvProject` имеет два свойства, которые возвращают его имя и полное имя. Это соответственно свойства `ReportName` и `ReportFullName`. При необходимости использовать имя отчета для одного из методов (например метод `SelectReport`) можно использовать оба имени. Во всех методах, где в качестве параметра применяется имя отчета, имеется дополнительный параметр `FullName` типа `Boolean`. При его значении `True` используется полное имя отчета.

Обратите внимание, что перед использованием любых свойств и методов, относящихся к отчету в компоненте `TRvProject`, отчет необходимо открыть.

Для этого используется метод `Open`:

```
RvProject1.Open;
```

или свойство `Active`:

```
RvProject1.Active := True;
```

Так же, если в процессе работы приложения в компонент `TRvProject` нужно загрузить новый проект отчета, процедуру открытия нужно повторить:

```
...
RvProject1.Close;
RvProject1.SetProjectFile(dlgOpenProject.FileName);
RvProject1.Open;
...
```

Компонент отчета необходимо связать с компонентом `TRvSystem`. Для этого в свойстве `Engine` компонента `TRvProject` необходимо задать ссылку на компонент `TRvSystem`.

Просмотр и печать отчета

Если в предыдущем пункте, обсуждая отчет в приложении Delphi, мы говорили о компоненте `TRvProject`, то за выполнение любых операций с ним отвечает компонент `TRvSystem`.

При стандартной настройке этого компонента при печати или предварительном просмотре отчета всегда отображается диалог настройки печати (см. рис. 23.4). Если отображение этого диалога необходимо, печать текущего отчета компонента `TRvProject`, с которым связан данный компонент `TrvSystem`, осуществляется методом

```
procedure Execute;
```

любого из этих компонентов.

Если диалог настройки печати не нужен, компонент `TRvSystem` позволяет выполнить операцию напрямую. Для этого необходимо выполнить несколько действий.

Сначала нужно настроить свойство

```
type
  TReportDest = (rdPreview, rdPrinter, rdFile);
property DefaultDest: TReportDest;
```

которое определяет, куда будет направлен отчет — в окно просмотра, на принтер или в файл.

Затем необходимо изменить свойство

```
type
  SystemSetup = (ssAllowSetup, ssAllowCopies, ssAllowCollate,
  ssAllowDuplex, ssAllowDestPreview, ssAllowDestPrinter, ssAllowDestFile,
  ssAllowPrinterSetup);
  TSystemSetups = Set of TSystemSetup;
property SystemSetups: TSystemSetups;
```

убрав из него опцию `ssAllowSetup`, которая включена по умолчанию:

```
RvSystem1.SystemSetups := RvSystem1.SystemSetups - [ssAllowSetup];
```

И, наконец, свойству

```
property DoNativeOutput: Boolean;
```

необходимо присвоить значение `False`, т. к. по умолчанию оно имеет значение `True`, которое и заставляет компонент показывать диалог настройки печати перед выполнением операции.

Обратите внимание на очень важную деталь — чтобы все сделанные настройки действительно сработали, печать отчета необходимо выполнять только методом `Execute` компонента `TRvProject`.

Сохранение отчета во внешнем файле

При помощи методов компонента `TRvSystem` можно сохранить отчет для последующей печати в формате PRN или сохранить проект Rave Reports в формате RAV.

Для реализации первого варианта необходимо в качестве источника печати указать файл:

```
...
if dlgSavePRN.Execute then
 begin
  rsSystem.DoNativeOutput := False;
  rsSystem.DefaultDest := rdFile;
  rsSystem.SystemSetups := rsSystem.SystemSetups - [ssAllowSetup];
  rsSystem.OutputFileName := dlgSavePRN.FileName;
  rpProject.Execute;
 end;
...
```

Как видите, здесь мы воспользовались методикой прямой печати, описанной в предыдущем разделе, указав в качестве приемника файл с расширением prn, выбранный в стандартном диалоге выбора файла.

Если же нужно сохранить проект отчета в файле с расширением rav, можно использовать метод `SaveToFile` компонента `TRvProject`:

```
if dlgSaveProject.Execute
  then RvProject1.SaveToFile(dlgSaveProject.FileName);
```

Также просто выполнить и обратную операцию — загрузить в компонент TRvProject проект отчета из файла, но при этом не забудьте закрыть текущий отчет:

```
...
RvProject1.Close;
RvProject1.LoadFromFile(dlgOpenProject.FileName);
RvProject1.Open;
...
```

Аналогичную функцию выполняет метод

```
procedure SetProjectFile(Value: String);
```

компонента `TRvProject`.

Компонент *TRvNDRWriter*

Компонент `TRvNDRWriter` предназначен для сохранения отчетов в файлах. При этом используется двоичный формат NDR.

Приемник данных определяется свойством

```
type
  TStreamMode = (smMemory, smTempFile, smFile, smUser);
property StreamMode: TStreamMode;
```

`smMemory` — для вывода данных используется поток в памяти (объект типа `TMemoryStream`);

`smTempFile` — данные сохраняются во временном файле, в папке, определенной в операционной системе для хранения временных файлов;

`smFile` — данные сохраняются в файле;

`smUser` — данные передаются в поток, заданный разработчиком.

Имя файла, в котором будет сохранен отчет, определяется свойством

```
property FileName: String;
```

А для определения потока используется свойство

```
property Stream: TStream;
```

Таким образом, если вы хотите использовать для сохранения отчета файл, перед использованием компонент настраивается, например, так:

```
...
RvNDRWriter1.StreamMode := smFile;
RvNDRWriter1.FileName := ReportFilePath;
...
```

Если вы планируете использовать поток, сделайте следующим образом:

```
var ReportStream: TMemoryStream;
...
 ReportStream := TMemoryStream.Create;
 try
  RvNDRWriter1.StreamMode := smUser;
  RvNDRWriter1.Stream := ReportStream;
...
 finally
  ReportStream.Free;
 end;
...
```

Но сначала этот отчет необходимо создать. Для этого используется обширный набор методов, позволяющих отображать текст и графику, создавать таблицы и заполнять их данными. Перед началом работы следует вызвать метод

```
procedure Start;
```

а по окончании создания отчета использовать метод

```
procedure Finish;
```

Например, следующий фрагмент кода создает в отчете текст с заданным положением:

```
with RvNDRWriter1 do
begin
 Start;
 Units := unMM;
 SetFont('Times New Roman', 14);
 Bold := True;
 OriginX := 0.0;
 OriginY := 0.0;
 GotoXY(1.0, 12.0);
 Print('Заголовок #1');
 GotoXY(6.0, 18.0);
 Println('Заголовок #2');
 GotoXY(6.0, 24.0);
 Println('Заголовок #3');
 GotoXY(6.0, 30.0);
```

```
 Println('Заголовок #4');
 Finish;
end;
```

А вот так можно нарисовать прямоугольник и разместить в нем изображение:

```
with RvNDRWriter1 do
begin
 Start;
 Units := unMM;
 SetBrush(clBlue, bsSolid, nil);
 Rectangle(5.0, 35.0, 65.0, 95.0);
 Bitmap := TBitmap.Create;
 Bitmap.LoadFromFile('factory.BMP');
 PrintBitmapRect(10.0, 40.0, 60.0, 90.0, Bitmap);
 Bitmap.Free;
 Finish;
end;
```

В данном фрагменте кода метод

```
procedure PrintBitmapRect(X1,Y1,X2,Y2: double; Bitmap: TBitmap);
```

отображает растровое изображение `Bitmap` в прямоугольнике, обеспечивая его масштабирование в соответствии с размерами прямоугольника.

Внимание!

Компонент `TRvNDRWriter` имеет свойство `Canvas` *(см. гл. 10)*, но использовать его нельзя — любые операции с канвой не возымеют действия и ваши труды не будут сохранены.

При использовании пары методов `Start` и `Finish` не нужно предпринимать никаких дополнительных усилий для сохранения отчета — это будет сделано методом `Finish`.

Множество других свойств и методов компонента (мы не будем останавливаться на них специально, т. к. их использование достаточно прозрачно) обеспечивают оформление отчета, управление страницами, настройку принтера и т. д.

Преобразование форматов данных

С компонентом `TRvNDRWriter` (вернее с файлами в формате NDR, которые он создает) взаимодействует ряд компонентов Rave Reports, которые обеспечивают преобразование данных из этого специфического формата в более распространенные форматы.

- Компонент `TRvRenderPDF` обеспечивает преобразование отчета в формат PDF для дальнейшего использования в Adobe Acrobat Reader.
- Компонент `TRvRenderHTML` обеспечивает преобразование отчета в формат HTML.
- Компонент `TRvRenderRTF` обеспечивает преобразование отчета в формат RTF.
- Компонент `TRvRenderText` обеспечивает преобразование отчета в текстовый формат.

Все они используются по одной схеме.

Сначала необходимо загрузить отчет из файла NDR в поток.

После этого вызывается метод

```
procedure PrintRender(NDRStream: TStream; OutputFileName: TFileName);
```

который и выполняет преобразование:

```
var ReportStream: TMemoryStream;
...
 ReportStream := TMemoryStream.Create;
 try
  ReportStream.LoadFromFile(NDRFilePath);
  RvRenderHTML1.NDRStream := ReportStream;
  RvRenderHTML1.PrintRender(ReportStream, 'sdf');
 finally
  ReportStream.Free;
 end;
```

Резюме

Приложение Delphi, которое реализует печать отчетов, должно иметь в своем составе компоненты `TRvProject` и `TRvSystem`. Первый обеспечивает представление проекта отчета из файла RAV в приложении. Второй взаимодействует с ядром генератора отчетов и управляет печатью и предварительным просмотром отчетов.

Набор компонентов Rave Reports позволяет преобразовать формат данных отчета в наиболее распространенные форматы данных: HTML, RTF, PDF, TXT.

ГЛАВА 26

Отчеты для приложений баз данных

Генератор отчетов Rave Reports позволяет создавать отчеты, отображающие данные из различных источников данных. Для этого используются средства визуальной среды и компоненты со страницы **Rave** Палитры компонентов Delphi.

Приложение может воспользоваться соединением, предоставленным генератором отчетов, созданным и настроенным в проекте отчета, а также может использовать компоненты Rave Reports, которые обеспечивают передачу данных в отчет.

При помощи компонентов и визуальной среды генератора отчетов Rave Reports приложение баз данных может использовать базы данных, обращаясь к ним при помощи трех технологий доступа к данным *(подробнее о технологиях доступа к данным см. часть IV)*, доступных по умолчанию.

- ADO;
- dbExpress;
- BDE.

Кроме этого, компонент `TRvDataSetConnection` (без участия визуальной среды) позволяет использовать любые наборы данных, открываемые через компоненты доступа к данным Delphi. Любой потомок компонента `TDataSet` может быть использован в отчете Rave Reports. Это означает, что дополнительно к перечисленным технологиям отчеты Rave Reports могут использовать компоненты InterBase Express *(см. гл. 18)*, а также подключать клиентские наборы данных в распределенных приложениях *(см. часть IV)*.

Еще один интересный компонент `TRvCustomConnection` обеспечивает доступ к любым данным не из баз данных. Для него источником данных могут быть текстовые файлы, электронные таблицы, электронная почта и т. д.

Что касается элементов оформления отчетов средствами визуальной среды Rave Reports, можно создавать отчеты самых различных типов. Естественно,

это простые табличные отчеты, а также отчеты, представляющие данные в отношении "один-ко-многим", отчеты с группировкой данных, отчеты с вычисляемыми значениями.

В этой главе рассматриваются следующие вопросы:

- как подключить источник данных к отчету;
- типы соединений с источниками данных, используемые в проекте отчета и приложении Delphi;
- компоненты соединений с базами данных;
- использование компонента `TrvCustomConnection`;
- типы отчетов;
- использование вычисляемых значений.

Соединения с источниками данных в Rave Reports

Если отчет Rave Reports должен отображать данные из какого-либо источника данных, на этапе разработки в визуальной среде в проект отчета должны быть добавлены специальные объекты, обеспечивающие соединение с источником данных и формирование набора данных, который затем отображается в отчете.

В Rave Reports существуют два типа соединений с источниками данных (рис. 26.1):

- соединение через драйвер Rave Reports;
- соединение через компонент Rave Reports и компонент набора данных в приложении Delphi.

В обоих случаях соединение инкапсулировано в объекте визуальной среды, а различаются только способы доступа к данным и набор компонентов, необходимый для этого.

При соединении через драйвер Rave Reports проект отчета на этапе разработки и ядро генератора отчетов на этапе выполнения используют драйверы, которые реализованы в виде файлов с расширением rvd. Именно наличие этих файлов предопределяет выбор технологий доступа к данным при создании объекта соединения в среде разработки (рис. 26.2). В стандартную поставку Rave Reports 5.0 входят драйверы для следующих технологий доступа к данным:

- ADO;
- dbExpress;
- BDE.

Рис. 26.1. Соединения с источниками данных в отчете Rave Reports

При соединении через компоненты в Delphi сначала необходимо создать объект просмотра **Direct Data View**, который реализует прямой доступ к набору данных на основе активного соединения в приложении Delphi. При этом соединение может быть создано на основе любой доступной в Delphi технологии доступа к данным. Это позволяет сделать набор компонентов Rave Reports на странице **Rave** Палитры компонентов Delphi. Это следующие технологии доступа к данным:

- ADO;
- DbExpress;
- BDE;
- InterBase Express;
- клиентские наборы данных распределенных приложений;
- источники данных, не использующие базы данных (текстовые файлы, электронная почта, электронные таблицы и т. д.).

За визуализацию данных в отчете отвечают специализированные элементы оформления, расположенные на странице **Report** Палитры инструментов визуальной среды Rave Reports. Они связываются с полями объекта просмотра данных, созданного разработчиком. Это может быть просмотр на основе запроса SQL, созданного разработчиком, или прямой просмотр набора данных Delphi. Просмотр объединяет нужные поля из таблиц, которые

доступны через соединение с источником данных *(см. гл. 24)*. Для этого используется свойство `DataView` элементов оформления.

Соединения с источниками данных в визуальной среде Rave Reports

Любой отчет, работающий с базами данных, должен быть настроен соответствующим образом в визуальной среде создания отчетов Rave Reports. Независимо от типа соединения здесь должен быть создан хотя бы один объект доступа к данным.

Если вы хотите создать соединение через драйверы Rave Reports, вам потребуется объект соединения и связанный с ним объект просмотра на основе запроса SQL.

При необходимости использовать соединение на основе компонентов Delphi в визуальной среде вам потребуется создать объект прямого просмотра.

Рассмотрим подробнее действия, которые необходимо выполнить в визуальной среде Rave Reports для создания и настройки соединения.

Соединение через драйвер Rave Reports

Проект, отчеты которого используют соединение через драйверы Rave Reports, должен содержать два объекта доступа к данным (в дереве проекта они отображаются в ветви **Data View Dictionary**).

В первую очередь нужно создать объект соединения (компонент `Database`). Для этого используется команда **File | New Data Object** главного меню. В появившемся диалоге **Data Connections** (см. рис. 24.4) необходимо выбрать **Database Connection** и нажать кнопку **Next** — появится список **Database Connection Type** (рис. 26.2). В нем представлены все доступные в визуальной среде типы соединений. Элементы списка соответствуют файлам RDV драйверов соединений. В стандартную поставку входят драйверы для соединений ADO, BDE, dbExpress. Эти файлы по умолчанию устанавливаются в папке \Delphi7\Rave5\DataLinks.

После нажатия кнопки **Finish** появляется специализированный диалог настройки параметров соединения, типовой для выбранной технологии доступа к данным *(см. часть IV)*, и создается новый объект соединения. В его свойствах `AuthDesign` и `AuthRun` содержатся настройки соединения, которые используются при его открытии на этапах разработки и выполнения отчета соответственно.

Теперь, когда соединение готово, необходимо создать просмотр данных. Он формирует набор данных, который будет передан в отчет и там связан с элементами оформления (см. рис. 26.1).

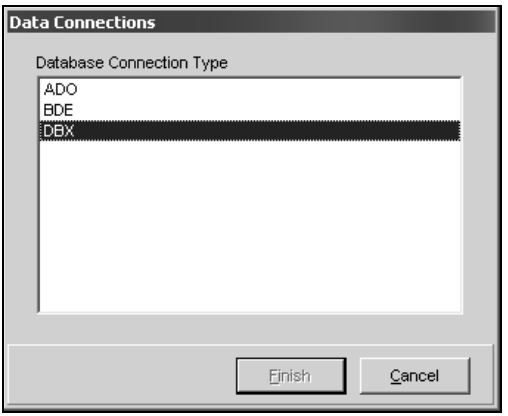


Рис. 26.2. Диалог выбора драйвера соединения

В диалоге выбора объекта доступа к данным (см. рис. 24.4) используются два объекта просмотра. Но с соединением на основе драйвера Rave Reports умеет взаимодействовать только один из них — **Driver Data View**. Для него необходимо выбрать соединение из созданных ранее в этом проекте (рис. 26.3).

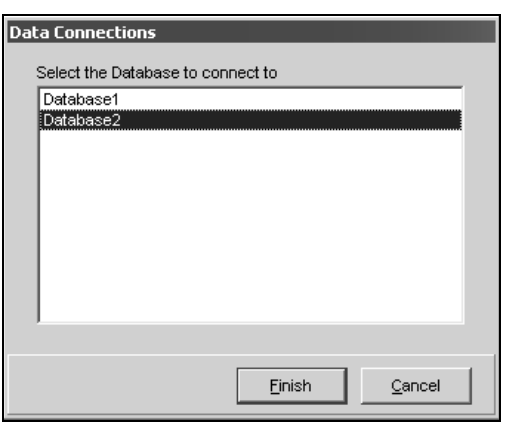


Рис. 26.3. Список выбора соединения для просмотра **Driver Data View**

После этого в редакторе **Query Advanced Designer** (см. рис. 24.5) создается запрос SQL, используемый для создания набора данных просмотра. Подробнее о пользовательском интерфейсе этого редактора рассказывается в *гл. 24*.

После создания объекта просмотра (компонент `DriverDataView`) в его свойстве `Database` появляется ссылка на объект выбранного соединения, а свойство `Query` позволяет редактировать запрос SQL просмотра. Для этого используется все тот же редактор **Query Advanced Designer**. Соединение про-

смотра можно изменить, выбрав новое в списке свойства `Database`, однако при этом может потребоваться частично или полностью переписать запрос SQL.

В состав объекта просмотра входят объекты всех полей, определенных на этапе разработки для запроса SQL.

Теперь подготовленный в просмотре набор данных можно подключить к элементам оформления нужного отчета *(см. разд. "Типы отчетов" ниже в данной главе)*.

Соединение через компонент приложения Delphi

Второй тип соединения базируется на компоненте набора данных, который уже существует и соединен с источником данных в приложении Delphi. Основой такого соединения в приложении является один из специализированных компонентов со страницы **Rave** Палитры компонентов Delphi — например, компонент `TRvDataSetConnection`. В проекте отчета в визуальной среде Rave Reports соединение обеспечивает объект просмотра **Direct Data View**. При создании объекта прямого просмотра для него необходимо выбрать из списка одно из соединений из приложения Delphi, в котором будет использоваться проект отчета. В этом списке можно выбрать все соединения, доступные как во время разработки, так и во время выполнения приложения. Для этого используются флажки в нижней части окна (рис. 26.4).

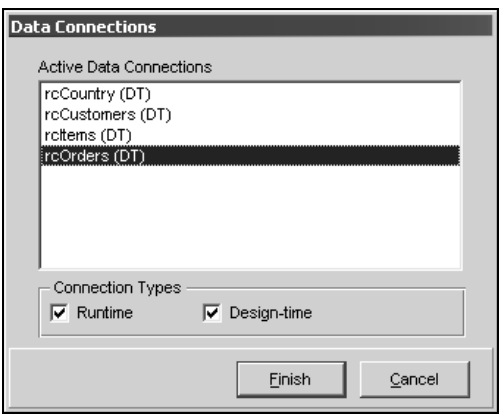


Рис. 26.4. Список доступных соединений из приложения Delphi

После создания объект прямого просмотра (компонент `DataView`) доступен в дереве проекта и может быть использован. В его состав входят все поля набора данных, связанного в приложении Delphi с компонентом соединения. И их можно подключить к элементам оформления нужного отчета *(см. разд. "Типы отчетов" ниже в данной главе)*.

Соединения с источниками данных в приложении

Теперь давайте посмотрим, как нужно использовать специализированные компоненты Rave Reports в приложениях Delphi для того, чтобы создать соединение отчета с источником данных. Их основная задача — передать в отчет связанный набор данных.

Для создания соединения можно использовать следующие компоненты:

- `TRvCustomConnection` — обеспечивает доступ к источникам данных, не основанным на базах данных;
- `TRvDataSetConnection` — создает соединение с любыми компонентами наборов данных, предком которых является класс `TDataSet`;
- `TRvTableConnection` — создает соединение с компонентом `TTable`;
- `TRvQueryConnection` — создает соединение с компонентом `TQuery`.

После переноса на форму проекта Delphi эти компоненты становятся доступны для выбора при создании прямых просмотров **Direct Data View** в визуальной среде создания отчетов Rave Reports. Однако перед этим компонент соединения необходимо связать с набором данных.

Компонент *TRvDataSetConnection*

Компонент `TRvDataSetConnection` позволяет отчету получить доступ к наборам данных, инкапсулированных в любых компонентах, произошедших от класса `TDataSet`. Это открывает перед разработчиком самые широкие возможности по созданию отчетов для любых приложений баз данных и распределенных приложений.

Сразу после переноса на форму компонент становится доступным в визуальной среде Rave Reports при создании объекта прямого просмотра. Однако толк от ненастроенного соединения пока небольшой. Сначала его нужно связать с компонентом набора данных. Для этого предназначено свойство

```
property DataSet: TDataSet;
```

И это все. Теперь созданный в визуальной среде объект прямого просмотра автоматически получит объекты полей, соответствующие полям в наборе данных компонента `DataSet`.

Впрочем, еще несколько вспомогательных свойств могут дать разработчику дополнительные удобства.

Примечание

Здесь мы рассмотрим только часть свойств и методов. Компонент `TRvDataSetConnection` обладает большой группой свойств и методов, которые,

будучи использованы в методах-обработчиках событий, позволяют дополнительно оформлять отчет. Более детально эти свойства и методы рассматриваются *ниже в разд. "Компонент TRvCustomConnection" данной главы*.

Свойство

```
property FieldAliasList: TStrings;
```

пригодится, если нужно изменить имена полей в прямом просмотре проекта отчета. Для этого в списке свойства в формате `Name = Alias` задаются имена полей связанного набора данных и их псевдонимы, которые будут использованы в объекте прямого просмотра.

Методы-обработчики событий компонента отслеживают процесс навигации по набору данных при печати отчета.

При открытии соединения для создания отчета генератором отчетов вызывается метод-обработчик

```
type
  TRPConnectorEvent = procedure(Connection: TRvCustomConnection);
property OnOpen: TRPConnectorEvent;
```

При открытии соединения отчет требует передать ему информацию о структуре набора данных (метаданные). Компонент соединения делает это и вызывает метод-обработчик

```
property OnGetCols: TRPConnectorEvent;
```

Когда курсор устанавливается на первую строку набора данных, вызывается метод-обработчик

```
property OnFirst: TRPConnectorEvent;
```

а при перемещении на следующую запись можно использовать метод

```
property OnNext: TRPConnectorEvent;
```

Если генератор отчетов нашел нужную запись и считал ее для представления в отчете, для отслеживания этого события разработчик может использовать метод

```
property OnGetRow: TRPConnectorEvent;
```

При достижении последней записи набора вызывается метод-обработчик

```
type
  TRPEOFEvent = procedure(Connection: TRvCustomConnection; var Eof:
Boolean);
property OnEOF: TRPEOFEvent;
```

При повторном использовании набора данных, например при печати отчетов "один-ко-многим" или при группировке записей, соединение обновля-

ется. При этом все параметры соединения и набора данных приводятся к исходному состоянию и вызывается метод-обработчик

```
property OnRestore: TRPConnectorEvent;
```

Кроме этого, при печати сложных отчетов генератору отчетов может понадобиться некоторое подмножество отсортированных записей набора данных.

Перед началом сортировки и фильтрации и после их завершения вызываются пары методов-обработчиков

```
property OnGetSort: TRPConnectorEvent;
property OnSetSort: TRPConnectorEvent;
```

и

```
property OnGetFilter: TRPConnectorEvent;
property OnSetFilter: TRPConnectorEvent;
```

Однако вы можете обеспечить дополнительную фильтрацию записей, передаваемых из набора данных в отчет. Для этого используется метод-обработчик

```
type
  TRPValidateRowEvent = procedure(Connection: TRvCustomConnection;
  var ValidRow: Boolean);
property OnValidateRow: TRPValidateRowEvent;
```

Параметр `ValidRow` управляет отправкой отдельной записи отчету: при значении `True` запись пропускается в отчет.

Рассмотрим простой пример. Для отчета, печатающего всем нам хорошо известную таблицу COUNTRY из демонстрационной базы Delphi, можно ограничить список стран, а также исключить страны с площадью территории, меньше заданной:

```
procedure TForm1.RvSomeConValidateRow(Connection: TRvCustomConnection;
var ValidRow: Boolean);
begin
 with TRvDataSetConnection(Connection) do
  ValidRow :=DataSet.FieldByName('Area').AsInteger > 1000000;
end;
```

Компоненты, использующие BDE

В состав набора компонентов соединений Rave Reports включены два компонента, которые обеспечивают связь прямого просмотра в проекте отчета с набором данных BDE.

Компонент `TRvTableConnection` работает с компонентом `TTable`. Для связывания с таблицей BDE используется свойство

```
property Table: TTable;
```

Еще одно свойство

```
property UseSetRange: Boolean;
```

при значении `True` определяет, что при создании отчета будут использованы механизмы фильтрации и сортировки компонента `TTable`.

Компонент `TRvQueryConnection` работает с компонентом `TQuery`.

Свойство

```
property Query: TQuery;
```

задает компонент `TQuery`.

Остальные свойства и методы этих компонентов соответствуют компоненту `TRvDataSetConnection`.

Компонент *TRvCustomConnection*

Компонент `TRvCustomConnection` обеспечивает доступ к самым разнообразным источникам данных. Фактически через этот компонент разработчик может передать в отчет все данные, какие только сможет загрузить в приложение. Причина столь удивительной универсальности кроется в том, что:

- во-первых, компонент изначально не ориентирован ни на один конкретный вид данных;
- во-вторых, работа по созданию строк отчета возлагается на разработчика, который должен использовать для этого методы-обработчики событий.

Повторим банальную истину, что чудес не бывает, и, как видите, за гибкость приходится расплачиваться дополнительным объемом работы.

Для того чтобы настроить соединение через компонент `TRvCustomConnection`, необходимо выполнить следующие действия.

1. Определить число строк отчета и установить его в компоненте.
2. Создать структуру данных отчета (метаданные). Здесь нужно решить, какие именно поля будут присутствовать в отчете, в каком порядке, дать им названия и определить их тип данных.
3. Создать процедуру, обеспечивающую передачу данных из источника данных в текущую строку отчета.
4. Связать компонент соединения с объектом прямого просмотра.

Обсудим эту последовательность действий более детально на простом примере. Создадим небольшое приложение, которое позволяет загружать текстовые файлы в два компонента TMemo. Перенесем на форму и настроим все необходимые компоненты Rave Reports.

Затем разработаем отчет, который печатает данные из этих двух компонентов в двух колонках. Отчет тоже несложен и состоит из полос заголовка и окончания, а также полосы данных с расположенными на ней двумя элементами оформления DataText.

Наша задача сейчас — настроить компонент TRvCustomConnection так, чтобы он мог отображать данные из двух компонентов TMemo.

Листинг 26.1. Методы-обработчики событий компонента TRvCustomConnection, обеспечивающего соединение отчета с массивами Memo

```
procedure TfmMain.rcCustomOpen(Connection: TRvCustomConnection);
begin
 Connection.DataRows := Max(meLeft.Lines.Count, meRight.Lines.Count);
 i := 0;
end;

procedure TfmMain.rcCustomGetCols(Connection: TRvCustomConnection);
begin
 Connection.WriteField('LeftColumn', dtString, 40, 'LeftColumn', '');
 Connection.WriteField('RightColumn', dtString, 40, 'RightColumn', '');
end;

procedure TfmMain.rcCustomGetRow(Connection: TRvCustomConnection);
begin
 if meLeft.Lines.Count >= i
  then Connection.WriteStrData('',  meLeft.Lines[i])
  else Connection. WriteNullData;
 if meRight.Lines.Count >= i
  then Connection.WriteStrData('',  meRight.Lines[i])
  else Connection. WriteNullData;
 Inc(i);
end;
```

При открытии соединения в методе-обработчике OnOpen рассчитывается число записей, необходимое для отображения наиболее длинного из двух файлов.

Метод-обработчик OnGetCols вызывается, когда отчету необходимы метаданные о наборе данных соединения. Здесь создаются два поля.

Для этого используется метод

```
procedure WriteField(Name: String; DataType; TRPDataType; Width: Integer;
FullName: String; Description: String);
```

который создает поле в соответствии с переданными в нем параметрами.

И при печати отчета для каждой строки вызывается метод-обработчик `OnGetRow`, в котором задаются значения полей. Для каждого типа данных используется свой метод:

```
function WriteBCDData(FormatData: String; NativeData: Currency): String;
function WriteBlobData(var: Buffer; Len: Longint): String;
function WriteBoolData(FormatData: String; NativeData: Boolean): String;
function WriteCurrData(FormatData: String; NativeData: Currency): String;
function WriteDateTime(FormatData: String; NativeData: TDateTime);
function WriteFloatData(FormatData: String; NativeData: Extended): String;
function WriteIntData(FormatData: String; NativeData: Integer): String;
function WriteNullData;
function WriteStrData(FormatData: String; NativeData: String): String;
```

Обратите внимание, что все эти методы не определяют, какому именно полю будет присвоено значение. Поэтому присваивание осуществляется в порядке следования полей: первый по порядку метод отправляет в отчет значение для первого поля, второй для второго и т. д.

Примечание

Методы-обработчики компонентов `TRvCustomConnection` и `TRvDataSetConnection` совпадают *(см. выше разд. "Компонент TRvDataSetConnection" данной главы)*.

Теперь осталось связать соединение с проектом отчетов. Это делается стандартным образом — при создании объекта прямого просмотра. Но здесь есть одна особенность. Как уже говорилось выше, при создании прямого просмотра в нем автоматически создаются объекты полей, соответствующие полям набора данных. И теперь мы знаем, что у компонента соединения имеется специальный метод-обработчик `OnGetCols`, который вызывается при создании полей.

Однако, если вы создадите объект прямого просмотра обычным способом, визуальная среда Rave Reports создаст один-единственный объект поля, не имеющего ничего общего с реальными метаданными. Для того чтобы поля импортировались в проект отчета правильно, необходимо, чтобы при создании объекта просмотра приложение, содержащее компонент `TRvDataSetConnection`, было запущено. Тогда в диалоге выбора соединений (см. рис. 26.4) необходимо включить флажок **Runtime**, и вы увидите компонент нужного соединения. В этом случае объект прямого просмотра получит все необходимые поля, которые затем следует связать с элементами оформления отчета.

Аутентификация пользователя в отчете

Два объекта Rave Reports позволяют включить в проекте отчета механизм проверки имени пользователя и пароля. Это объекты **Simple Security Controller** (элемент `SimpleSecurity`) и **Data Lookup Security Controller** (элемент `LookupSecurity`), которые доступны для выбора в диалоге создания объектов доступа к данным визуальной среды Rave Reports (см. рис. 24.4). Создается новый объект командой **File | New Data Object** главного меню Rave Reports. Созданный объект появляется в ветви **Data View Dictionary** дерева проекта.

Элемент `SimpleSecurity` предназначен для хранения списка пользователей и их паролей. Он имеет свойство `UserList`, в котором в формате `UserName = Password` заносятся имена и пароли пользователей.

Элемент `LookupSecurity` обеспечивает загрузку имен пользователей и паролей из таблицы базы данных. Для этого к нему через свойство `DataView` должен быть подключен соответствующий просмотр данных. В свойстве `UserField` необходимо указать поле, которое содержит имена пользователей, а в свойстве `PasswordField` задать поле с паролями.

Теперь несколько слов о том, как подключить созданные объекты.

Вы можете организовать аутентификацию на двух уровнях — уровне проекта и уровне отчета. В обоих случаях используется свойство `SecurityControl` объекта проекта или отчета. В нем необходимо выбрать нужный объект аутентификации.

Однако это действие не сделает ваши отчеты сколько-нибудь защищеннее — все необходимые проверочные операции придется написать самому в исходном коде приложения Delphi. Единственное отличие в аутентификации по уровням в том, где именно вы сможете получить доступ к объекту аутентификации — из компонента проекта или отчета.

Для организации простейшей проверки имени пользователя и пароля на уровне проекта в приложении нужно написать примерно такой код:

```
...
rpReport.Open;
 if rpReport.ProjMan.SecurityControl.IsValidUser(
                                   edUserName.Text,
                                   edPassword.Text)
  then rpReport.Execute
  else ShowMessage('Доступ запрещен');
 rpReport.Close;
...
```

В данном случае доступ к объекту аутентификации `SecurityControl` (класс `TRaveBaseSecurity`) осуществляется через объект менеджера проекта `ProjMan` (класс `TRaveProjectManager`).

Метод

```
function IsValidUser(AUserName: string; APassword: string): Boolean;
```

этого объекта возвращает значение `True`, если переданные в параметрах имя и пароль не совпадают со значениями из списка или базы данных.

Для уровня отчета код выглядит так:

```
...
rpReport.Open;
 if rpReport.ProjMan.ActiveReport.SecurityControl.IsValidUser(
                                   edUserName.Text,
                                   edPassword.Text)
  then rpReport.Execute
  else ShowMessage('Доступ запрещен');
 rpReport.Close;
...
```

Здесь объект `ActiveReport` (класс `TRaveReport`) представляет текущий отчет.

Типы отчетов

Сейчас мы займемся вопросами разработки собственно отчетов. Схема использования элементов оформления, работающих с объектами доступа к данным, стандартна для любых типов отчетов. Поэтому сначала мы рассмотрим общую методику на примере простого отчета, а затем перейдем к более сложным отчетам.

Для всех рассматриваемых типов отчетов создано демонстрационное приложение DemoReports.

Простой табличный отчет

Для создания отчетов, использующих данные из источников, предоставленных объектами соединений и просмотров, используются элементы оформления со страницы **Report** Палитры инструментов Rave Reports.

Основой, без которой нельзя использовать полосы (элементы `Band` и `DataBand`), является элемент `Region`. Он ограничивает часть страницы, на которой будут печататься данные.

Главную роль в отчетах для приложений баз данных играют полосы. Это невизуальные элементы оформления, моделирующие горизонтальную область или строку отчета. На странице **Report** доступны два таких элемента. Обычная полоса `Band` создает горизонтальную область, которая не изменяет свое абсолютное или относительное положение на странице. Например,

созданная на основе элемента `Band` полоса `TitleBand` всегда располагается в начале первой страницы отчета и оформляет заголовок таблицы (рис. 26.5). Полоса `FooterBand` будет напечатана сразу после основной таблицы — но ее конкретное положение на странице зависит от размера набора данных.

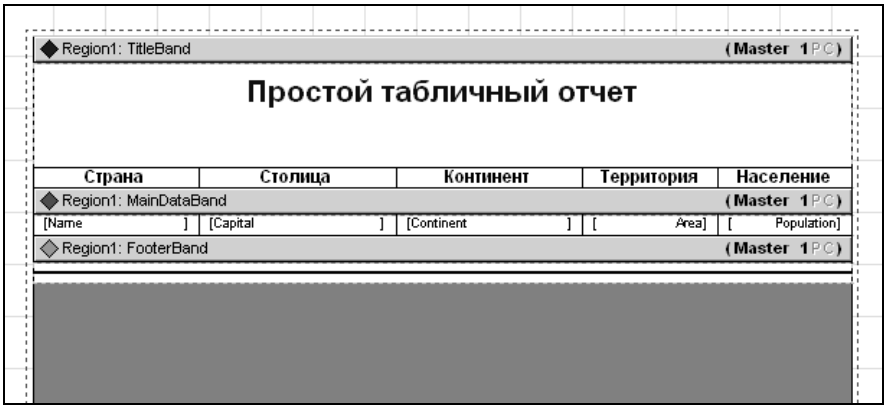


Рис. 26.5. Страница простого табличного отчета в визуальной среде Rave Reports

Элемент оформления `DataBand` обеспечивает размножение строк отчета в соответствии с числом строк набора данных. Для этого полосу данных `MainDataView` необходимо связать с просмотром при помощи свойства `DataView`. В нашем примере это прямой просмотр `CountryView`, связанный с компонентом соединения `TRvDataSetConnection` в приложении.

На полосе данных `MainDataView` необходимо разместить элементы оформления `DataText`. Каждый из этих элементов связывается с объектом просмотра и полем данных такого просмотра. Для этого используются свойства `DataView` и `DataField` соответственно. Таким образом, каждый элемент оформления `DataText` размножается вместе с полосой данных и формирует в отчете колонку значений поля набора данных.

Расположив на полосах горизонтальные и вертикальные линии, можно легко оформить данные в табличном виде.

Отчет "один-ко-многим"

При помощи средств Rave Reports можно создавать и более сложные отчеты. В приложениях баз данных очень часто используются отношения "один-ко-многим" между наборами данных.

Давайте посмотрим, как создать отчет "один-ко-многим". Само собой, он должен быть связан как минимум с двумя просмотрами, которые находятся в отношении "один-ко-многим".

Внимание!

Для компонентов наборов данных в Delphi не нужно создавать отношение "один-ко-многим" — речь идет о том, что их поля позволяют такое отношение создать теоретически.

Подобно рассмотренному выше простому отчету, отчет "один-ко-многим" может содержать полосы `Band` и `DataBand`. Причем дополнительные настройки необходимы для обоих типов полос. Число полос `DataBand` должно соответствовать числу используемых в отчете наборов данных. Полосы `Band` несут в основном оформительскую нагрузку, и их число зависит от эскиза отчета и вашей фантазии.

В качестве примера создадим отчет для двух таблиц из демонстрационной базы данных Delphi. Таблицы CUSTOMER и ORDERS находятся в отношении "один-ко-многим". Для них в тестовом приложении создано соединение с использованием ADO, и два табличных компонента ADO подключены к компонентам соединения `TRvDataSetConnection`.

Соответственно полоса данных `CustBand` будет отображать записи из набора данных `tCustomer`, а полоса `OrdBand` — из набора данных `tOrders` (рис. 26.6). Их необходимо связать с объектами прямых просмотров, как уже описывалось выше для примера простого отчета.

Рис. 26.6. Страница отчета MasterDetailReport в визуальной среде Rave Reports

А теперь займемся созданием отношения "один-ко-многим".

В подчиненной полосе данных `OrdBand` необходимо задать значения для четырех свойств.

- Свойство `ControllerBand` должно содержать ссылку на главную полосу `CustBand`.

- Свойство `MasterDataView` должно содержать ссылку на главный объект просмотра `CustomersView`.
- В свойстве `MasterKey` необходимо задать ключевое поле `CustNo` главного просмотра `CustomersView`, по которому будет установлено отношение.
- В свойстве `DetailKey` необходимо задать ключевое поле `CustNo` подчиненного просмотра `OrdersView`, по которому будет установлено отношение.

Кроме этого, необходимо настроить атрибуты местоположения полос на странице отчета. Для этого используется Редактор полос отчета **Band Style Editor** (рис. 26.7), который открывается при щелчке на кнопке свойства `BandStyle` Инспектора объектов визуальной среды Rave Reports. В нем в группе **Print Location** для подчиненной полосы `OrdBand` необходимо установить флажок **Detail**.

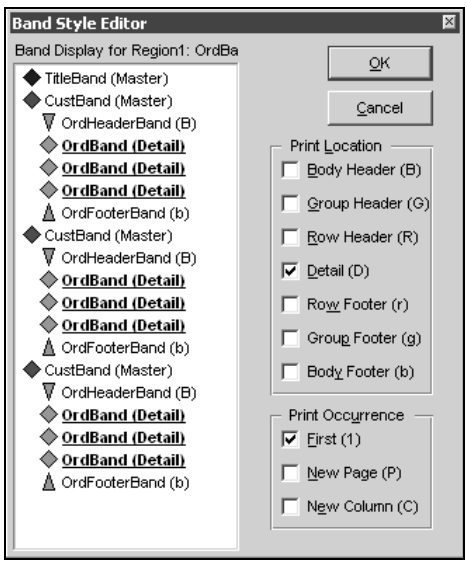


Рис. 26.7. Редактор полос отчета **Band Style Editor** для отчета MasterDetailReport

На этом настройка отношения "один-ко-многим" завершена. Однако скажем еще несколько слов об использовании обычных полос при оформлении такого рода отчетов. В нашем примере две дополнительные полосы `OrdHeaderBand` и `OrdFooterBand` помогают визуально выделить группы записей подчиненной полосы. Для этого необходимо в их свойстве `ControllerBand` выбрать полосу данных `OrdBand`. Затем в редакторе полос отчета в группе **Print Location** для полосы `OrdHeaderBand` необходимо выбрать флажок **Body Header (B)**, а для полосы `OrdFooterBand` — флажок **Body Footer (b)**.

Обратите внимание (см. рис. 26.6 и 26.7), что значки маркировки на полосах страницы и в редакторе полос наглядно демонстрируют текущий статус

полосы. Цветом выделены уровни вложения данных и подчиненность полос. Полосы с маркировкой одного цвета печатаются в одном блоке. Квадраты обозначают полосы данных, а треугольники — обычные полосы, при этом направление вершины треугольника обозначает полосу заголовка или окончания. Левая панель редактора полос отчета **Band Style Editor** (рис. 26.7) наглядно демонстрирует модель отчета, как если бы он был напечатан для трех записей набора данных.

На основе отчета "один-ко-многим" можно легко разработать и более сложные отчеты. Для этого необходимо детальную полосу данных связать с новыми детальными полосами и настроить по описанной методике отношение "один-ко-многим".

Группирующий отчет

Отчеты, работающие с базами данных, часто должны отображать данные с различными уровнями группировки. Обычно группировка осуществляется в наборе данных, если он создается на основе запроса SQL с применением оператора `GROUP BY`. Но в самом наборе данных невозможно предусмотреть оформление групп записей, однако это можно сделать в отчете.

Для полосы данных, которая отображает данные из просмотра с группировкой, можно создать полосы группового заголовка и группового окончания. Для этого используются свойства `GroupDataView` и `GroupKey`. Первое должно указывать на объект группирующего просмотра, а второе задает поле или несколько полей, по которым осуществляется группировка. Применительно к оформлению отчета это означает, что при изменении значения группового ключа будут напечатаны полосы группового заголовка и окончания.

В качестве таких полос могут использоваться обычные полосы и полосы данных. Обычные полосы применяются, если группировка имеет один уровень вложенности (для каждого значения группового ключа существует одна или несколько сгруппированных записей). Полосы данных используются, если группировка имеет несколько уровней (внутри группы выделяется еще один групповой ключ и каждая запись в группе имеет еще несколько сгруппированных записей второго уровня).

Кроме этого, для полос группового заголовка необходимо в свойстве `ControllerBand` задать основную полосу данных и настроить свойство `BandStyle`. Для группового заголовка в редакторе **Band Style Editor** в группе **Print Location** устанавливается флажок **Group Header (G)**, а для полосы группового окончания — флажок **Group Footer (g)**.

Обычно группирующие запросы SQL используют агрегатные функции для вычисления одного или нескольких величин по всей группе. Чаще всего это общая денежная сумма или общее количество. Такие величины удобно размещать в полосах группового окончания.

Использование вычисляемых значений

На странице **Reports** Палитры инструментов визуальной среды Rave Reports доступны несколько компонентов, которые позволяют применять агрегатные функции к значениям полей набора данных, переданного через соединение в отчет.

К агрегатным относятся следующие функции:

- `AVG` — вычисление среднего;
- `COUNT` — подсчет числа элементов множества;
- `MAX` — нахождение максимального значения;
- `MIN` — нахождение минимального значения;
- `SUM` — суммирование.

Простые элементы оформления позволяют получать вычисляемые значения на базе полей одного источника данных, более сложные "умеют" объединять несколько источников данных.

Рассмотрим все эти элементы оформления.

Вычисляемые значения по одному источнику

Для вычисления агрегатного значения одного или нескольких полей одного источника данных используются два элемента оформления.

Элемент `CalcText` позволяет отобразить результат вычисления на полосе отчета. Так же, как и обычный элемент `DataText`, его необходимо связать с просмотром и полем. Для этого используются свойства `DataView` и `DataField` соответственно.

Кроме этого, элемент `CalcText` должен быть связан со специализированным элементом `CalcController` *(см. ниже)*, который будет управлять процессом вычисления. Связывание осуществляется при помощи свойства `Controller`.

В свойстве `CalcType` задается одна из пяти перечисленных выше агрегатных функций.

Дополнительно, для функции `COUNT` можно настроить еще три свойства. При необходимости включить или отключить подсчет нулевых значений или пробелов используются булевские свойства `CountNulls` и `CountBlanks` соответственно. А свойство `CountValue` позволяет задать значение поля, которое будет учитываться при расчете функции, все остальные значения будут игнорироваться.

Вычисленное значение при сохранении может быть отформатировано в соответствии с шаблоном, заданным свойством `DisplayFormat`.

Для всех агрегатных функций можно задать момент начала вычислений. Для этого в свойстве `Initializer` необходимо указать элемент оформления отчета, и при его печати начнется вычисление. Это может быть любой элемент, расположенный с элементом `CalcText` на одной полосе. Но желательно использовать для этого специализированный элемент `CalcController` *(см. ниже)*.

Примечание

Пример использования элемента оформления `CalcText` имеется в отчетах, рассмотренных нами выше.

Элемент `CalcTotal` является невизуальным аналогом элемента `CalcText`. Поэтому он обладает всеми свойствами, о которых рассказывается выше для элемента `CalcText`. Вычисленное при его печати значение разработчик может использовать по своему усмотрению после того, как оно сохранено. Приемником вычисленного значения может быть один из параметров объекта отчета.

Свойство `DestParam` позволяет выбрать один из предопределенных или созданных разработчиком параметров отчета (свойство `Parameters` объекта отчета).

Свойство `DestPiVar` задает переменную отчета, в которую будет передано вычисленное значение (свойство `PIVars` объекта отчета).

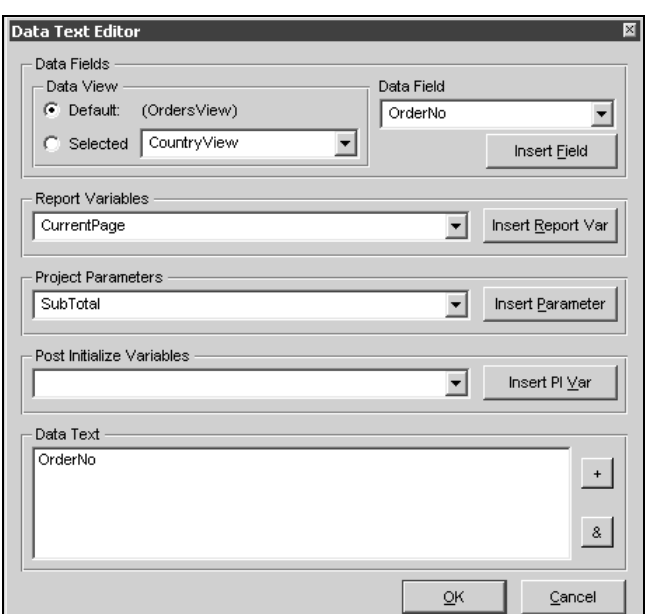


Рис. 26.8. Редактор свойства `DataField`

Затем параметр или переменная может быть использована для дальнейших вычислений или напечатана при помощи элемента `DataText`. В редакторе свойства `DataField` этого элемента (рис. 26.8) параметры и переменные отчета можно выбрать из списков **Project Parameters** и **Post Initialize Variables**.

Вычисляемые значения по нескольким источникам

Вычислительный элемент `CalcOp` позволяет проводить вычислительные операции над значениями из двух различных источников.

Разработчик должен задать исходные значения и источники данных, используя два набора свойств (табл. 26.1). Назначение части этих свойств вам уже знакомо *(см. разд. "Вычисляемые значения по одному источнику" выше)*.

Таблица 26.1. *Свойства элемента `CalcOp` для определения двух источников данных*

Первый источник	Второй источник	Назначение свойства
`Src1CalcVar`	`Src2CalcVar`	Определяет вычисляемый элемент, результат которого берется в качестве исходного
`Src1DataField`	`Src2DataField`	Задает поле просмотра, над значениями которого проводятся вычисления. Игнорируется при заданном свойстве `SrcXCalcVar`
`Src1DataView`	`Src2DataView`	Задает поле просмотра, над значениями которого проводятся вычисления. Игнорируется при заданном свойстве `SrcXCalcVar`
`Src1Function`	`Src2Function`	Позволяет выбрать математическую функцию (их список гораздо шире, чем просто агрегатные функции), которая будет выполнена над исходным значением перед вычислением основной операции элемента
`Src1Value`	`Src2Value`	Задает фиксированное значение, над которым производится вычислительная операция

Собственно функция, которая должна обработать значения из двух заданных источников, задается свойством `Operator`.

После выполнения основной операции результат может быть обработан еще один раз, если вы зададите математическую функцию в свойстве `ResultFunction`.

Таким образом, при помощи элемента `CalcOp` разработчик может реализовывать довольно сложные вычисления.

Если задать в качестве двух источников данных:

- два фиксированных значения (свойства `Src1Value` и `Src2Value`);
- два поля из одного или двух просмотров данных (свойства `Src1DataField` и `Src2DataField`);
- комбинацию первых двух вариантов

то их значения будут последовательно обработаны вычислительной операцией, которую вы зададите свойствами:

- `Src1Function`;
- `Src1Function`;
- `Operator`;
- `ResultFunction`.

Кроме этого, элемент `CalcOp` позволяет создавать вычислительные цепочки, если использовать в качестве одного или двух источников другие вычислительные элементы (рис. 26.9).

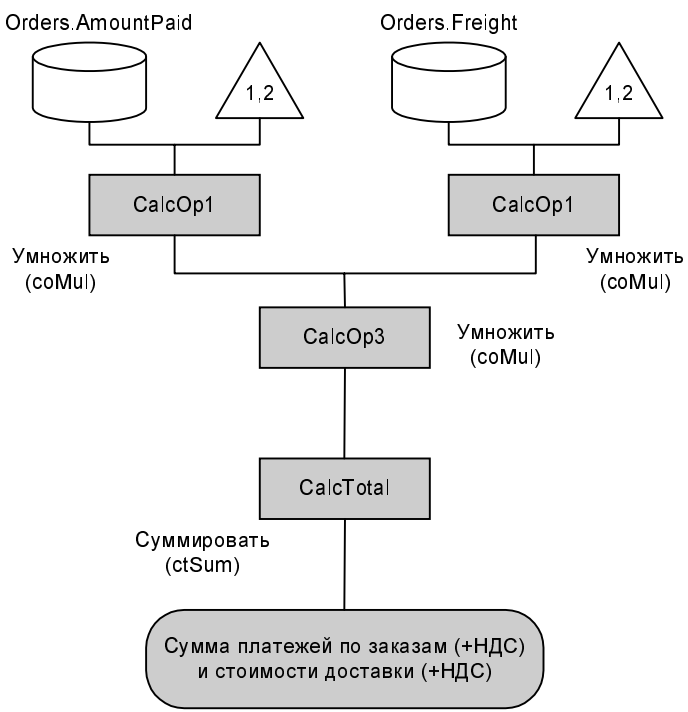


Рис. 26.9. Пример вычислительной цепочки на основе элементов `CalcOp`

Это могут быть как простые элементы CalcText и CalcTotal, так и другие элементы CalcOp, которые, в свою очередь, могут содержать сколь угодно сложные вычислительные цепи.

Пример использования элемента CalcOp имеется в демонстрационном приложении DemoReports на дискете, прилагаемой к этой книге.

Управляющие вычислительные элементы

Выше мы упоминали о свойстве Controller элементов CalcText и CalcTotal, которое позволяет определить момент начала вычислений. Для этого используется специальный невизуальный элемент CalcController. Обычно он располагается на той же полосе, что и вычислительные элементы и инициализирует процесс вычисления в момент своей печати. Хотя на самом деле невизуальный элемент CalcController не печатается, тем не менее событие OnBeforePrint он получает исправно вместе со всеми элементами, расположенными на данной полосе. А значит и с инициализацией вычислений он справится вполне.

Обладая несколькими специфическими свойствами, он позволяет определить момент начала вычислений более точно. И так же, как элемент FontMaster используется для централизованного управления шрифтами, этот элемент может быть центром управления вычислениями.

Свойство InitCalcVar должно ссылаться на другой вычислительный элемент. И вычисленное им значение будет использовано в качестве начального.

Свойство InitDataField задает поле данных, значение которого используется в качестве начального. Работает, если свойство InitCalcVar не задано.

Свойство InitValue задает начальное значение, если предыдущие два свойства не заданы.

Для того чтобы эти свойства работали и задавали начальное значение, ссылка на элемент CalcController должна присутствовать в свойстве Initializer элементов оформления CalcText или CalcTotal.

Элемент DataCycle используется для дополнительной фильтрации, сортировки и просмотра данных, поля которого используются для вычислений. С его помощью можно получить нужное для вычислений подмножество записей набора данных, не изменяя просмотра данных.

Свойство DataView задает просмотр данных, с которым будет работать элемент DataCycle.

При помощи свойств MasterDataView, MasterKey и DetailKey можно получить подмножество записей для отношения "один-ко-многим".

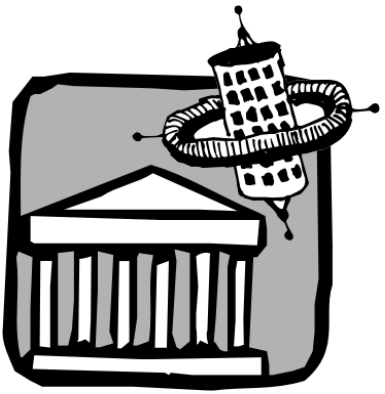
Свойство SortKey позволяет отсортировать записи по заданному полю.

Резюме

Генератор отчетов Rave Reports позволяет создавать разнообразные отчеты для приложений баз данных.

Совокупность компонентов Delphi и средств визуальной среды Rave Reports обеспечивают создание соединений с источниками данных любых видов, в том числе и не основанных на СУБД. Соединения могут использовать технологии доступа к данным, реализованные соответствующими компонентами Delphi *(см. часть IV)* или драйверами Rave Reports.

Основой отчетов являются элементы `Region`, `Band` и `DataBand`, которые обеспечивают размножение строк отчета в соответствии с записями источников данных. Набор специализированных элементов оформления позволяет отображать в отчетах все основные типы данных.



ЧАСТЬ VII

Технологии программирования

Глава 27. Стандартные технологии программирования

Глава 28. Динамические библиотеки

Глава 29. Потоки и процессы

Глава 30. Многомерное представление данных

Глава 31. Использование возможностей Shell API

ГЛАВА 27

Стандартные технологии программирования

В этой главе обсуждаются вопросы использования стандартных для приложений Windows технологий программирования. С их помощью ваше приложение обретет законченный вид и будет соответствовать необходимым канонам и правилам пользовательского интерфейса.

В настоящей главе рассматриваются следующие вопросы:

- интерфейс переноса Drag-and-Drop;
- механизм Drag-and-Dock;
- усовершенствованное масштабирование;
- как правильно передавать фокус между элементами управления;
- управление мышью;
- ярлыки.

Интерфейс переноса Drag-and-Drop

Интерфейс переноса и приема компонентов появился достаточно давно. Он обеспечивает взаимодействие двух элементов управления во время выполнения приложения. При этом могут выполняться любые необходимые операции. Несмотря на простоту реализации и давность разработки, многие программисты (особенно новички) считают этот механизм малопонятным и экзотическим. Тем не менее использование Drag-and-Drop может оказаться очень полезным и простым в реализации. Сейчас мы в этом убедимся.

Для того чтобы механизм заработал, требуется настроить соответствующим образом два элемента управления. Один должен быть источником (Source), второй — приемником (Target). При этом источник никуда не перемещается, а только регистрируется в качестве такового в механизме.

Примечание

Один элемент управления может быть одновременно источником и приемником.

Пользователь помещает указатель мыши на нужный элемент управления, нажимает левую кнопку мыши и, не отпуская ее, начинает перемещать курсор ко второму элементу. При достижении этого элемента пользователь отпускает кнопку мыши. В этот момент выполняются предусмотренные разработчиком действия. При этом первый элемент управления является источником, а второй — приемником.

После выполнения настройки механизм включается и реагирует на перетаскивание мышью компонента-источника в приемник. Группа методов-обработчиков обеспечивает контроль всего процесса и служит для хранения исходного кода, который разработчик сочтет нужным связать с перетаскиванием. Это может быть передача текста, значений свойств (из одного редактора в другой можно передать настройки интерфейса, шрифта и сам текст); перенос файлов и изображений; простое перемещение элемента управления с места на место и т. д. Пример реализации Drag-and-Drop в Windows — возможность переноса файлов и папок между дисками и папками.

Как видите, можно придумать множество областей применения механизма Drag-and-Drop. Его универсальность объясняется тем, что это всего лишь средство связывания двух компонентов при помощи указателя мыши. А конкретное наполнение зависит только от фантазии программиста и поставленных задач.

Весь механизм Drag-and-Drop реализован в базовом классе `TControl`, который является предком всех элементов управления. Рассмотрим суть механизма.

Любой элемент управления из Палитры компонентов Delphi является источником в механизме Drag-and-Drop. Его поведение на начальном этапе переноса зависит от значения свойства

```
type TDragMode = (dmManual, dmAutomatic);
property DragMode: TDragMode;
```

Значение `dmAutomatic` обеспечивает автоматическую реакцию компонента на нажатие левой кнопки мыши и начало перетаскивания — при этом механизм включается самостоятельно.

Значение `dmManual` (установлено по умолчанию) требует от разработчика обеспечить включение механизма вручную. Этот режим используется в том случае, если компонент должен реагировать на нажатие левой кнопки мыши как-то иначе. Для инициализации переноса используется метод

```
procedure BeginDrag(Immediate: Boolean; Threshold: Integer = -1);
```

Параметр `Immediate = True` обеспечивает немедленный старт механизма. При значении `False` механизм включается только при перемещении курсора на расстояние, определенное параметром `Threshold`.

О включении механизма сигнализирует указатель мыши — он изменяется на курсор, определенный в свойстве

```
property DragCursor: TCursor;
```

Еще раз напомним, что источник при перемещении курсора не изменяет собственного положения, и только в случае успешного завершения переноса сможет взаимодействовать с приемником.

Приемником может стать любой компонент, в котором создан метод-обработчик

```
procedure DragOver(Source: TObject; X, Y: Integer; State: TDragState;
var Accept: Boolean);
```

Он вызывается при перемещении курсора в режиме Drag-and-Drop над этим компонентом. В методе-обработчике можно предусмотреть селекцию источников переноса по нужным атрибутам.

Если параметр `Accept` получает значение `True`, то данный компонент становится приемником. Источник переноса определяется параметром `Source`. Через этот параметр разработчик получает доступ к свойствам и методам источника. Текущее положение курсора задают параметры `X` и `Y`. Параметр `State` возвращает информацию о характере движения мыши:

```
type TDragState = (dsDragEnter, dsDragLeave, dsDragMove);
```

`dsDragEnter` — указатель появился над компонентом;

`dsDragLeave` — указатель покинул компонент;

`dsDragMove` — указатель перемещается по компоненту.

Приемник должен предусматривать выполнение некоторых действий в случае, если источник завершит перенос именно на нем. Для этого используется метод-обработчик

```
type TDragDropEvent = procedure(Sender, Source: TObject; X, Y: Integer)
of object;
property OnDragDrop: TDragDropEvent;
```

который вызывается при отпускании левой кнопки мыши на компоненте-приемнике. Доступ к источнику и приемнику обеспечивают параметры `Source` и `Sender` соответственно. Координаты мыши возвращают параметры `X` и `Y`.

При завершении переноса элемент управления — источник — получает соответствующее сообщение, которое обрабатывается методом

```
type TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer)
of object;
property OnEndDrag: TEndDragEvent;
```

Источник и приемник определяются параметрами `Sender` и `Target` соответственно. Координаты мыши определяются параметрами `X` и `Y`.

Для программной остановки переноса можно использовать метод `EndDrag` источника (при обычном завершении операции пользователем он не используется):

```
procedure EndDrag(Drop: Boolean);
```

Параметр `Drop = True` завершает перенос. Значение `False` прерывает перенос.

Теперь настало время закрепить полученные знания на практике. Рассмотрим небольшой пример. В проекте DemoDragDrop на основе механизма Drag-and-Drop реализована передача текста между текстовыми редакторами и перемещение панелей по форме (рис. 27.1).

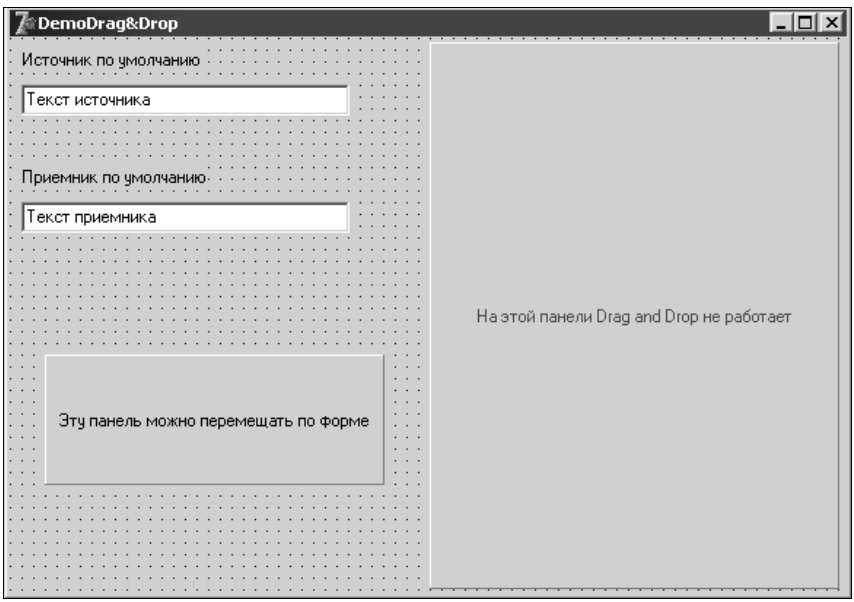


Рис. 27.1. Главная форма проекта DemoDragDrop

Листинг 27.1. Секция `implementation` модуля главной формы проекта DemoDragDrop

```
implementation

{$R *.DFM}

procedure TMainForm.Edit1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
 if Button = mbLeft
  then TEdit(Sender).BeginDrag(True);
end;

procedure TMainForm.Edit2DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
 if Source is TEdit
  then Accept := True
  else Accept := False;
end;

procedure TMainForm.Edit2DragDrop(Sender, Source: TObject; X, Y:
Integer);
begin
 TEdit(Sender).Text := TEdit(Source).Text;
 TEdit(Sender).SetFocus;
 TEdit(Sender).SelectAll;
end;

procedure TMainForm.Edit1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
 if Assigned(Target)
  then TEdit(Sender).Text := 'Текст перенесен в ' + TEdit(Target).Name;
end;

procedure TMainForm.FormDragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
 if Source.ClassName = 'TPanel'
  then Accept := True
  else Accept := False;
end;

procedure TMainForm.FormDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
 TPanel(Source).Left := X;
 TPanel(Source).Top := Y;
end;

end.
```

Для однострочного редактора `Edit1` определены методы-обработчики источника. В методе `Edit1MouseDown` обрабатывается нажатие левой кнопки мыши

и включается механизм переноса. Так как свойство DragMode для Edit1 имеет значение dmManual, то компонент без проблем обеспечивает получение фокуса и редактирование текста.

Метод Edit1EndDrag обеспечивает отображение информации о выполнении переноса в источнике.

Для компонента Edit2 определены методы-обработчики приемника. Метод Edit2DragOver проверяет класс источника и разрешает или запрещает прием.

Метод Edit2DragDrop осуществляет перенос текста из источника в приемник.

Примечание

Обратите внимание, что оба компонента TEdit одновременно являются источниками и приемниками. Для этого каждый из них использует методы-обработчики другого. А исходный код методов настроен на обработку владельца как экземпляра класса TEdit.

Форма, как приемник Drag-and-Drop, обеспечивает перемещение панели Panel2, которая выступает в роли источника. Метод FormDragOver запрещает прием любых компонентов, кроме панелей. Метод FormDragDrop осуществляет перемещение компонента.

Панель не имеет своих методов-обработчиков, т. к. работает в режиме dmAutomatic и не нуждается в дополнительной обработке завершения переноса.

Интерфейс присоединения Drag-and-Dock

Эта возможность появилась в Delphi 4. Она "подсмотрена" опять-таки у разработчиков из Microsoft, внедривших плавающие панели инструментов в MS Office, Internet Explorer и другие продукты (рис. 27.2).

Речь идет о том, что ряд элементов управления (а конкретно — потомки класса TWinControl) могут служить носителями (доками) для других элементов управления с возможностью их динамического перемещения из одного дока в другой при помощи мыши. Перетаскивать можно практически все — от статического текста до форм включительно. Пример использования техники Drag-and-Dock дает сама среда разработки Delphi — с ее помощью можно объединять на экране различные инструменты, такие как Инспектор объектов и Менеджер проекта.

Как и в случае с технологией перетаскивания Drag-and-Drop, возможны два варианта реализации техники Drag-and-Dock: автоматический и ручной. В первом случае дело сводится к установке нужных значений для нескольких свойств, а остальную часть работы берет на себя код VCL; во втором, как следует из названия, вся работа возлагается на программиста.

Итак, что же нужно сделать для внедрения Drag-and-Dock? В Инспекторе объектов необходимо изменить значение свойства `DragKind` на `dkDock`, а свойства `DragMode` — на `dmAutomatic`. Теперь этот элемент управления можно перетаскивать с одного носителя-дока на другой.

Носителем других компонентов (доком) может служить потомок `TWinControl`. У него есть свойство `DockSite`, установка которого в `True` разрешает перенос на него других компонентов. Если при этом еще и установить свойство `AutoSize` в `True`, док будет автоматически масштабироваться в зависимости от того, что на нем находится. В принципе, этими тремя операциями исчерпывается минимальный обязательный набор.

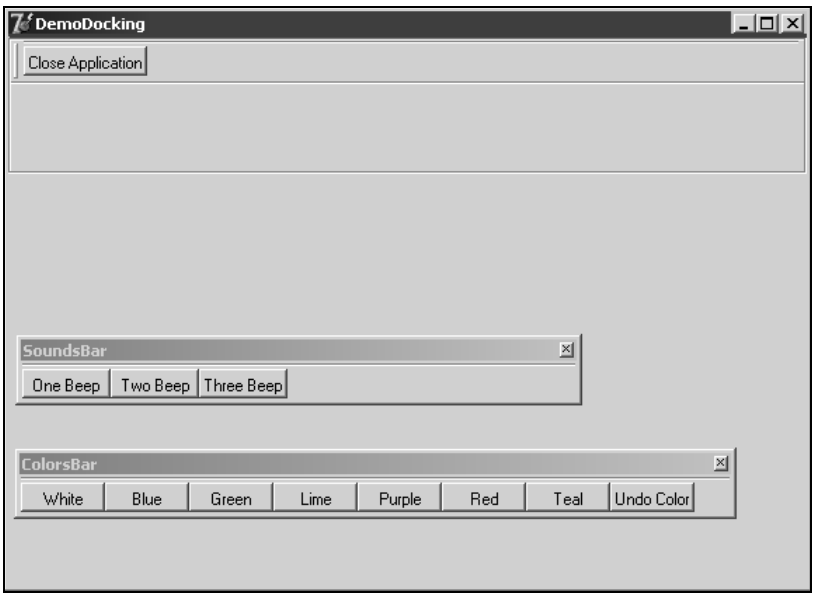


Рис. 27.2. Плавающие панели инструментов

Естественно, для программиста предусмотрены возможности контроля за этим процессом. Каждый переносимый элемент управления имеет два события, возникающие в моменты начала и конца переноса:

```
type  TStartDockEvent = procedure(Sender: TObject; var DragObject:
TDragDockObject) of object;
TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer)
of object;
```

В первом из методов `Sender` — это переносимый объект, а `DragObject` — специальный объект, создаваемый на время процесса переноса и содержащий его свойства. Во втором `Sender` — это также переносимый объект, а `Target` — объект-док.

Док тоже извещается о событиях во время переноса:

```
type TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl;
var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean)
of object;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject;
X, Y: Integer; State: TDragState; var Accept: Boolean) of object;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject;
X, Y: Integer) of object;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; NewTarget:
TWinControl; var Allow: Boolean) of object;
```

Как только пользователь нажал кнопку мыши над переносимым компонентом и начал сдвигать его с места, всем потенциальным докам (компонентам, свойство которых `DockSite` установлено в `True`) рассылается событие `OnGetSiteInfo`. С ним передаются параметры: кто хочет "приземлиться" (параметр `DockClient`) и где (`MousePos`). В ответ док должен сообщить решение, принимает он компонент (параметр `CanDock`) и предоставляемый прямоугольник (`InfluenceRect`) или нет. При помощи этого события можно принимать только определенные элементы управления, как показано в примере:

```
procedure TForm1.Panel1GetSiteInfo(Sender: TObject; DockClient: TControl;
var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean);
begin
 if DockClient is TBitBtn then CanDock := False;
end;
```

Два последующих события в точности соответствуют своим аналогам из механизма переноса Drag-and-Drop). Событие `OnDockOver` происходит при перемещении перетаскиваемого компонента над доком, `OnDockDrop` — в момент его отпускания. Наконец, `OnUnDock` сигнализирует об уходе компонента с дока и происходит в момент его "приземления" в другом месте.

Между доком и содержащимися на нем элементами управления есть двусторонняя связь. Все "припаркованные" элементы управления содержатся в векторном свойстве `DockClients`, а их количество можно узнать из свойства `DockClientCount`:

```
s := '';
 for i := 0 to Panel1.DockClientCount-1 do
  AppendStr(s,Panel1.DockClients[i].Name+#$D#$A);
 ShowMessage(s);
```

С другой стороны, если элемент управления находится на доке, то ссылка на док располагается в свойстве `HostDockSite`. С ее помощью можно установить, где находится элемент, и даже поменять свойства дока:

```
procedure TMyForm.Button1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
 (Sender as TControl).HostDockSite.SetTextBuf(pChar((Sender
  as TControl).Name));
end;
```

Компоненты можно не только переносить с одного дока на другой, но и отпускать в любом месте. Хотя сам по себе компонент `TControl` и его потомки не являются окнами Windows, но специально для этого случая создается окно-носитель. Свойство `FloatingDockSiteClass` как раз и определяет класс создаваемого окна. По умолчанию для большинства компонентов значение этого свойства равно `TCustomDockForm`. Это — форма, которая обладает свойствами дока и создается в момент отпускания элемента управления вне других доков. Внешне она ничем не отличается от обычной стандартной формы. Если вы хотите, чтобы ваша плавающая панель инструментов выглядела по-особенному, нужно породить потомка от класса `TCustomDockForm` и связать свойство `FloatingDockSiteClass` с этим порожденным классом:

```
TMyCustomFloatingForm = class(TCustomDockForm)
public
  constructor Create(AOwner: TComponent); override;
end;
constructor TMyCustomFloatingForm.Create(AOwner: TComponent);
begin
 inherited Create(AOwner);
 BorderStyle := bsNone;
end;

...
procedure TForm1.FormCreate(Sender: TObject);
begin
...
 ToolBar1.FloatingDockSiteClass := TMyCustomFloatingForm;
...
end;
```

В этом примере решена типовая задача — сделать так, чтобы несущее окно плавающей панели инструментов не содержало заголовка. Внешний вид таких панелей приведен на рис. 27.3.

Переносить компоненты можно не только с помощью мыши, но и программно. Для этого есть пара методов `ManualDock` и `ManualFloat`. В приводимом ниже примере нажатие кнопки с именем `BitBtn1` переносит форму `CustForm` на док `MainForm.Panel1` и размещает ее по всей доступной площади (параметр выравнивания `alClient`). Нажатие кнопки `BitBtn2` снимает эту

форму с дока и выравнивает ее по центру экрана. В свойствах `UndockHeight` и `UndockWidth` хранятся высота и ширина элемента управления на момент, предшествующий помещению на док:

```
procedure TMainForm.BitBtn1Click(Sender: TObject);
begin
 CustForm.ManualDock(MainForm.Panel1,nil,alClient);
end;

procedure TMainForm.BitBtn2Click(Sender: TObject);
begin
 with CustForm do
 begin
 ManualFloat(Rect((Screen.Width-UndockWidth) div 2,
                    (Screen.Height-UndockHeight) div 2,
                    (Screen.Width+UndockWidth) div 2,
                    (Screen.Height+UndockHeight) div 2)
           );
end;
```

Рис. 27.3. Плавающие панели инструментов без заголовка окна

Полное рассмотрение внутреннего устройства механизмов Drag-and-Dock потребовало бы расширения объема этой главы. Тем, кто хочет использовать их на все 100%, рекомендуем обратиться к свойствам `UseDockManager`

и DockManager. Последнее представляет собой COM-интерфейс, позволяющий расширить возможности дока, вплоть до записи его состояния в поток (класс TStream).

Усовершенствованное масштабирование

В класс TControl добавлены свойства, позволяющие упростить масштабирование форм и находящихся на них компонентов.

Свойство Anchors:

```
TAnchorKind = (akLeft, akTop, akRight, akBottom);
  TAnchors = set of TAnchorKind;
property Anchors: TAnchors;
```

отвечает за привязку компонентов к определенным краям формы при масштабировании. По умолчанию любой компонент привязан к верхней и левой сторонам ([akLeft, akTop]), т. е. не двигается при стандартном масштабировании. Но, изменив значение этого свойства, можно сделать так, чтобы компонент находился, к примеру, все время в нижнем правом углу.

С другой стороны, если прикрепить все четыре стороны, то получится интересный и нужный во многих случаях эффект. Такой компонент увеличивается и уменьшается вместе с формой; но в то же время сохраняется расстояние до всех четырех ее краев.

Свойство Constraints представляет собой набор ограничений на изменение размеров компонента. Оно содержит четыре свойства: MaxHeight, MaxWidth, MinHeight и MinWidth. Как легко догадаться из названий, размеры компонента могут меняться только в пределах значений этих четырех свойств.

Наконец, большинство элементов управления получили свойство AutoSize, позволяющее им автоматически масштабироваться при изменении содержимого (скажем, надписи на кнопке).

Управление фокусом

В процессе работы приложения тот или иной элемент управления получает фокус ввода в зависимости от действий пользователя. Очень часто передача фокуса между элементами управления должна быть упорядочена. Например, при вводе данных в приложениях баз данных пользователь должен иметь максимум удобств для обеспечения хорошей производительности труда. Для этого он должен работать только с клавиатурой, не отвлекаясь на лишние операции по передаче фокуса в нужный компонент при помощи мыши.

Для решения подобного рода проблем все оконные элементы управления имеют два свойства. Свойство TabOrder определяет порядок передачи фоку-

са между элементами управления одного владельца (формы, панели, группы) при нажатии клавиши <Tab>. Значение 0 имеет компонент, который будет получать фокус при открытии формы.

Для того чтобы свойство `TabOrder` работало, свойство `TabStop` должно иметь значение `True`.

Кроме этого, все кнопки (произошедшие от `TButtonControl`) имеют свойство `Default`, которое при значении `True` заставляет кнопку реагировать на нажатие клавиши <Enter> как на щелчок на кнопке, даже если она не имеет фокус. Только одна кнопка на форме может иметь это свойство установленным.

Для передачи фокуса любому оконному элементу управления программными средствами можно использовать метод

```
procedure SetFocus; virtual;
```

унаследованный от класса `TWinControl`.

При необходимости работы в форме применяется метод

```
function SetFocusedControl(Control: TWinControl): Boolean; virtual;
```

класса `TForm`, в параметре указывается указатель на компонент, принадлежащий форме.

Управление мышью

Каждый элемент управления обладает набором свойств и методов, обеспечивающих управление мышью. Понятно, что это важный и нужный механизм. Рассмотрим кратко его устройство.

Воздействие мышью на интерфейсные элементы приложения разработчик может отслеживать при помощи целой группы методов-обработчиков.

На нажатие кнопки мыши реагирует метод

```
type
  TMouseEvent = procedure (Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer) of object;
property OnMouseDown: TMouseEvent;
```

В параметре `Button` передается признак нажатой кнопки:

```
type TMouseButton = (mbLeft, mbRight, mbMiddle);
```

Параметр `Shift` определяет нажатие дополнительной клавиши на клавиатуре:

```
type TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
ssMiddle, ssDouble);
```

Параметры `X` и `Y` возвращают координаты курсора.

На отпускание кнопки мыши реагирует метод:

```
type
  TMouseEvent = procedure (Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer) of object;
property OnMouseUp: TMouseEvent;
```

Его параметры описаны выше.

При перемещении мыши можно вызывать метод-обработчик

```
TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState;
X, Y: Integer) of object;
property OnMouseMove: TMouseMoveEvent;
```

Если у разработчика нет необходимости так подробно отслеживать состояние мыши, можно воспользоваться двумя другими методами:

```
property OnClick: TNotifyEvent;
property OnDblClick: TNotifyEvent;
```

Первый реагирует на щелчок кнопкой, второй — на двойной щелчок.

Каждый элемент управления может изменять внешний вид указателя мыши, перемещающейся над ним. Для этого используется свойство

```
property Cursor: TCursor;
```

Для управления дополнительными возможностями мыши для работы в Internet (ScrollMouse) предназначены три метода обработчика, реагирующие на прокрутку:

- `property OnMouseWheel: TMouseWheelEvent;`

  вызывается при прокрутке;
- `property OnMouseWheelUp: TMouseWheelUpDownEvent;`

  вызывается при прокрутке вперед;
- `property OnMouseWheelDown: TMouseWheelUpDownEvent;`

  вызывается при прокрутке назад.

В VCL имеется класс `TMouse`, содержащий свойства мыши, установленной на компьютере. Обращаться к экземпляру класса, который создается автоматически, можно при помощи глобальной переменной `Mouse`. Свойства класса представлены в табл. 27.1.

В качестве примера обработки управляющих воздействий от мыши рассмотрим пример DemoMouse. Он очень прост. Перемещение мыши с нажатой левой кнопкой обеспечивает выделение прямоугольного фрагмента. Такую функцию вы можете наблюдать в любом графическом редакторе, а исходный код проекта использовать в собственных разработках (листинг 27.2).

Таблица 27.1. *Свойства и методы класса* `TMouse`

Объявление	Тип	Описание
`property Capture: HWND;`	Pu	Дескриптор элемента управления, над которым находится мышь
`property CursorPos: TPoint;`	Pu	Содержит координаты указателя мыши
`property DragImmediate: Boolean;`	Ro	При значении `True` реакция на нажатие выполняется немедленно
`property DragThreshold: Integer;`	Ro	Задержка реакции на нажатие
`property MousePresent: Boolean;`	Ro	Определяет наличие мыши
`type UINT = LongWord;` `property RegWheelMessage: UINT;`	Ro	Задает сообщение, посылаемое при прокрутке в ScrollMouse
`property WheelPresent: Boolean;`	Ro	Определяет наличие ScrollMouse
`property WheelScrollLines: Integer;`	Ro	Задает число прокручиваемых линий

Листинг 27.2. Модуль главной формы проекта DemoMouse

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
    ColorDlg: TColorDialog;
    StatusBar: TStatusBar;
    Timer: TTimer;
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
```

```
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure TimerTimer(Sender: TObject);
  private
    MouseRect: TRect;
    IsDown: Boolean;
    RectColor: TColor;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 if Button = mbLeft then
 with MouseRect do
 begin
  IsDown := True;
  Left := X;
  Top := Y;
  Right := X;
  Bottom := Y;
  Canvas.Pen.Color := RectColor;
 end;
 if (Button = mbRight) and ColorDlg.Execute
 then RectColor := ColorDlg.Color;
end;

procedure TMainForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 IsDown := False;
 Canvas.Pen.Color := Color;
 with MouseRect do
 Canvas.Polyline([Point(Left, Top), Point(Right, Top), Point(Right,
 Bottom), Point(Left, Bottom), Point(Left, Top)]);
 with StatusBar do
 begin
```

```
  Panels[4].Text := '';
  Panels[5].Text := '';
 end;
end;

procedure TMainForm.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
 with StatusBar do
 begin
  Panels[2].Text := 'X: ' + IntToStr(X);
  Panels[3].Text := 'Y: ' + IntToStr(Y);
 end;
 if Not IsDown then Exit;
 Canvas.Pen.Color := Color;
 with mouserect do
 begin
  Canvas.Polyline([Point(Left, Top), Point(Right, Top),
  Point(Right, Bottom), Point(Left, Bottom), Point(Left, Top)]);
  Right := X;
  Bottom := Y;
  Canvas.Pen.Color := RectColor;
  Canvas.Polyline([Point(Left, Top), Point(Right, Top),
  Point(Right, Bottom), Point(Left, Bottom), Point(Left, Top)]);
 end;
 with StatusBar do
 begin
  Panels[4].Text := 'Ширина: ' +
IntToStr(Abs(MouseRect.Right - MouseRect.Left));
  Panels[5].Text := 'Высота: ' +
IntToStr(Abs(MouseRect.Bottom - MouseRect.Top));
 end;
end;

procedure TMainForm.TimerTimer(Sender: TObject);
begin
 with StatusBar do
 begin
  Panels[0].Text := 'Дата: ' + DateToStr(Now);
  Panels[1].Text := 'Время: ' + TimeToStr(Now);
 end;
end;

end.
```

При нажатии левой кнопки мыши в методе-обработчике `FormMouseDown` включается режим рисования прямоугольника (`IsDown := True`) и задаются его начальные координаты.

При перемещении мыши по форме проекта вызывается метод-обработчик `FormMouseMove`, в котором координаты курсора и размеры прямоугольника передаются на панель состояния. Если левая кнопка мыши нажата (`IsDown = True`), то осуществляется перерисовка прямоугольника.

При отпускании кнопки мыши в методе `FormMouseUp` рисование прямоугольника прекращается (`IsDown := False`).

Если была нажата правая кнопка мыши, то метод-обработчик `FormMouseDown` обеспечивает отображение диалога выбора цвета, который позволяет сменить цвет линий прямоугольника.

Метод-обработчик `TimerTimer` обеспечивает отображение на панели состояния текущей даты и времени.

Примечание

Для рисования прямоугольника использовался метод `PolyLine`, который работает при перемещении курсора влево и вверх относительно начальной точки. Для стирания старого прямоугольника желательно использовать режимы XOR и NOTXOR, которые обеспечивают восстановление рисунка под линией. Подробно об этом *см. гл. 10*.

Ярлыки

Пользовательский интерфейс трудно представить без ярлычков с оперативной подсказкой (Hints). Если задержать курсор, например, над кнопкой или компонентом палитры самой среды Delphi, появляется маленький прямоугольник яркого цвета (окно подсказки), в котором одной строкой сказано о названии этого элемента или связанном с ним действии. Delphi поддерживает механизмы создания и отображения таких ярлычков в создаваемых программах.

Свойство, определяющее активность системы подсказки у элемента управления:

```
property ShowHint: Boolean;
```

Если свойство `ShowHint` установлено в `True`, и во время выполнения курсор задержался над компонентом на некоторое время, в окне подсказки высвечивается текстовая строка с подсказкой, которая задана свойством:

```
property Hint: string;
```

Подсказка компонента может быть пустой строкой — в этом случае система ищет в цепочке первый родительский компонент с непустой подсказкой.

Если в строке `Hint` встречается специальный символ-разделитель "|", то часть строки до него ("короткая") передается в окно подсказки, а после ("длинная") — присваивается свойству `Hint` объекта `Application`. Ее можно использовать, например, в строке состояния внизу главной формы приложения *(см. пример ниже)*.

Система оперативных подсказок имеет свойства и методы, общие для всех форм в приложении. Не удивительно, что они сосредоточены в Application — глобальном объекте, соответствующем работающему приложению. Все описанные ниже в этом разделе свойства относятся не к компоненту, показывающему подсказку, а именно к `Application`.

Фоновый цвет окна подсказки можно изменить посредством свойства

```
property HintColor: TColor;
```

У объекта `Application` значение свойства `ShowHint` нужно устанавливать во время выполнения, например, в обработчике `OnCreate` главной формы приложения. Оно является главенствующим для всей системы подсказок: если оно установлено в значение `False`, ярлычки не возникают.

Есть еще один способ получения подсказки. При смене текущего элемента управления (т. е. при смене текста в свойстве `Hint`) в объекте `Application` возникает событие

```
property OnHint: TNotifyEvent;
```

Пример:

```
procedure TForm1.AppHint(Sender: TObject);
begin
 Panel1.Caption:= Application.Hint;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
Application.OnHint := AppHint;
end;
```

В этом примере текст подсказки будет отображаться в строке состояния `Panel1` независимо от значения `ShowHint` у любого объекта — лишь бы этот текст был в наличии. Для этого разделяйте подсказку у элементов управления вашего приложения на две части при помощи символа "|" — краткая информация появится рядом с элементом, а более полная — в строке состояния.

```
function GetLongHint(const Hint: string): string;
function GetShortHint(const Hint: string): string;
```

У других компонентов свойство `ShowHint` интерпретируется системой так: когда курсор мыши останавливается над элементом управления или пунк-

том меню, и приложение не занято обработкой сообщения, происходит проверка, и если свойство `ShowHint` у элемента или у одного из его родительских элементов в иерархии равно `True`, то начинается ожидание.

Если в данный момент другие ярлычки не показываются, то интервал времени задается свойством `HintPause`:

```
property HintPause: Integer;
```

Интервал времени по умолчанию равен 500 мс. Если в данный момент уже виден ярлычок другого компонента, то интервал времени ожидания задается свойством:

```
property HintShortPause: Integer;
```

По истечении этого времени, если мышь осталась над тем же элементом управления, наступает момент инициализации окна подсказки. При этом программист может получить управление, предусмотрев обработчик события объекта `Application`:

```
property OnShowHint: TShowHintEvent;
TShowHintEvent = procedure (var HintStr: string; var CanShow: Boolean;
var HintInfo: THintInfo) of object;
```

Рассмотрим параметры обработчика события `OnShowHint`:

- `HintStr` — отображаемый текст;
- `CanShow` — необходимость (возможность) появления подсказки. Если в переменной `CanShow` обработчик вернет значение `False`, то окно подсказки высвечиваться не будет;
- `HintInfo` — структура, несущая всю информацию о том, какой элемент управления, где и как собирается показать подсказку. Ее описание:

```
THintInfo = record
  HintControl: TControl;
  HintPos: TPoint;
  HintMaxWidth: Integer;
  HintColor: TColor;
  CursorRect: TRect;
  CursorPos: TPoint;
end;
```

Для показа окна подсказки необходимо еще, чтобы у элемента управления или у его предков в цепочке строка `Hint` была непустой. Впрочем, это можно исправить в обработчике `OnShowHint`:

```
procedure TForm1.AppShowHint(var HintStr: string; var CanShow:
Boolean;var HintInfo: THintInfo);
begin
 if HintStr='' then
```

```
  begin
  HintStr := HintInfo.HintControl.Name;
  HintInfo.HintColor := clRed;
  CanShow := True;
  end;
end;
```

Присвоив этот метод обработчику `Application.OnShowHint`, установив `Form.ShowHint:=True` и очистив все строки `Hint`, получим в качестве подсказки имя каждого элемента.

Длительность показа ярлычка задается свойством

```
property HintHidePause: Integer;
```

По умолчанию его значение равно 2500 мс.

Свойство

```
property HintShortCuts: Boolean;
```

отвечает за показ вместе с текстом ярлычка описания "горячих" клавиш данного элемента управления.

Наконец, можно вручную "зажечь" и "потушить" ярлычок. При помощи метода

```
procedure ActivateHint(CursorPos: TPoint);
```

ярлычок показывается в точке `CursorPos` (система координат — экранная). "Спрятать" окно подсказки можно с помощью метода:

```
procedure CancelHint;
```

Без повторного перемещения мыши на текущий элемент оно более не возникнет.

Резюме

Delphi предоставляет разработчику набор стандартных программных механизмов, позволяющих добавлять к приложениям функции пользовательского интерфейса Windows. Кроме представленных здесь, разработчик может использовать расширенный набор функций, содержащийся в библиотеке Shell API, которой посвящена *гл. 31*.

ГЛАВА 28

Динамические библиотеки

Динамические библиотеки (DLL, Dynamic Link Library) играют важную роль в функционировании ОС Windows и прикладных программ. Они представляют собой файлы с откомпилированным исполняемым кодом, который используется приложениями и другими DLL. Реализация многих функций ОС вынесена в динамические библиотеки, которые используются по мере необходимости, обеспечивая тем самым экономию адресного пространства. DLL загружается в память только тогда, когда к ней обращается какой-либо процесс.

По существу динамические библиотеки отличаются от исполняемых файлов только одним, они не могут быть запущены самостоятельно. Для того чтобы динамическая библиотека начала работать, необходимо, чтобы ее вызвала уже запущенная программа или работающая DLL.

Обычно в динамические библиотеки выносятся группы функций, которые применяются для решения сходных задач. Кроме этого, в них можно хранить и использовать разнообразные ресурсы — от строк локализации до форм.

Динамическая библиотека может использоваться несколькими приложениями, при этом не обязательно, чтобы все они были созданы при помощи одного языка программирования.

Разновидностью динамических библиотек являются пакеты Delphi, предназначенные для хранения кода компонентов для среды разработки и приложений.

Применение динамических библиотек позволяет добиться ряда преимуществ:

- уменьшается размер исполняемого файла приложения и занимаемые им ресурсы;

- функции DLL могут использовать несколько процессов одновременно;
- управление динамическими библиотеками возлагается на операционную систему;
- внесение изменений в DLL не требует перекомпиляции всего проекта;
- одну DLL могут использовать программы, написанные на разных языках.

При разработке динамических библиотек в среде Delphi удобно использовать группу проектов, которая включает проект приложения и проекты динамических библиотек.

В этой главе рассматриваются следующие вопросы:

- структура файла DLL;
- инициализация DLL;
- явная и неявная загрузка;
- вызовы функций из динамической библиотеки;
- ресурсы в динамических библиотеках.

Проект DLL

Для создания динамической библиотеки в Репозитории Delphi имеется специальный шаблон. Его значок **DLL Wizard** расположен на странице **New** Репозитория. В отличие от проекта обычного приложения, проект DLL состоит всего из одного исходного файла. Впоследствии к нему можно добавлять отдельные модули и формы.

Листинг 28.1. Исходный файл проекта динамической библиотеки

```
library Project1;

{ Important note about DLL memory management: ShareMem must be the
  first unit in your library's USES clause AND your project's (select
  Project-View Source) USES clause if your DLL exports any procedures or
  functions that pass strings as parameters or function results. This
  applies to all strings passed to and from your DLL-even those that
  are nested in records and classes. ShareMem is the interface unit to
  the BORLNDMM.DLL shared memory manager, which must be deployed along
  with your DLL. To avoid using BORLNDMM.DLL, pass string information
  using PChar or ShortString parameters. }

uses
  SysUtils,
  Classes;
```

```
{$R *.res}

begin
end.
```

Примечание

Обширный комментарий в каждом проекте DLL касается использования модуля `ShareMem`. О нем рассказывается ниже.

Для определения типа проекта используется ключевое слово `library` (вместо `program` в обычном проекте). При компиляции такого проекта динамической библиотеки создается файл с расширением dll.

Как и в любом другом проекте, в проекте динамической библиотеки можно использовать иные модули. Это могут быть просто модули с исходным кодом и модули форм. При этом динамическая библиотека может экспортировать функции, описанные не только в главном файле, но и в присоединенных модулях.

Блок `begin..end` называется *блоком инициализации библиотеки* и предназначен для размещения кода, который автоматически выполняется при загрузке DLL.

Между секцией `uses` и блоком инициализации можно располагать исходный код функций динамической библиотеки и их объявления. При этом можно использовать любые конструкции языка Object Pascal, а также применять формы и компоненты.

Примечание

При создании динамических библиотек очень удобно использовать группы проектов. В группу помещается проект приложения и проект (проекты) необходимой для его работы динамической библиотеки (библиотек). Для переключения между проектами удобно использовать Диспетчер проектов (команда **Project Manager** из меню **View**). Его можно поместить в окно Редактора кода.

Еще один способ удобной работы с проектами динамических библиотек заключается в задании для DLL вызывающей программы. Это делается в диалоге команды **Parameters** из меню **Run** (рис. 28.1). Вызывающее приложение задается в группе **Host Application**. В результате после компиляции динамической библиотеки вызывается использующее ее приложение.

Для того чтобы приложения могли применять функции динамической библиотеки, необходимо, во-первых, экспортировать их из DLL; во-вторых, объявить функции в самом приложении как внешние. Ниже рассматриваются способы решения этих задач.

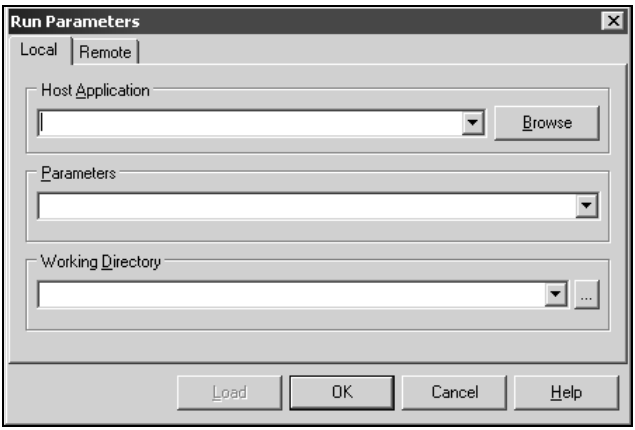


Рис. 28.1. Диалог команды **Parameters** меню **Run**

Экспорт из DLL

Для создания перечня экспортируемых из динамической библиотеки процедур и функций используется ключевое слово `exports`. При этом можно указывать как функции, описанные в главном файле DLL, так и функции из присоединенных модулей.

В качестве примера рассмотрим исходный код динамической библиотеки DataCheck, простейшие функции которой проверяют введенную строку перед конвертацией на соответствие одному из типов данных.

Листинг 28.2. Исходный код динамической библиотеки DataCheck

```
library DataCheck;

uses
  Windows, SysUtils, Classes, Messages, Forms,
  Dialogs, StdCtrls, ComCtrls;

  function ValidDate(AText: String): Integer;
  begin
   try
    Result := 0;
    StrToDate(AText);
   except
    on E:EConvertError do Result := -1;
   end;
  end;
```

```
  function ValidTime(AText: String): Integer;
  begin
   try
    Result := 0;
    StrToTime(AText);
   except
    on E:EConvertError do Result := -1;
   end;
  end;

  function ValidInt(AText: String): Integer;
  begin
   try
    Result := 0;
    StrToInt(AText);
   except
    on E:EConvertError do Result := -1;
   end;
  end;

  exports
   ValidInt,
   ValidDate index 1,
   ValidTime index 2 name 'IsValidTime';

begin
  if Length(DateToStr(Date)) < 10
  then ShowMessage('Год представлен двумя цифрами');
end.
```

Итак, три функции этой библиотеки обеспечивают проверку строки перед преобразованием ее в целое число, дату или время. Для обеспечения экспорта этих функций их необходимо объявить в секции `exports`.

При компиляции библиотеки адрес, имя и порядковый номер экспортируемой функции добавляется к специальной таблице экспорта в файле DLL.

Примечание

Компилятор Delphi без проблем добавит таблицу экспорта и к исполняемому файлу приложения. Правда, при этом получить доступ к такой функции невозможно — это системное ограничение Windows.

Попробуйте объявить пару функций после ключевого слова `exports` в обычном приложении — проект компилируется без ошибок. Но сами функции недоступны другим процессам.

Имена процедур и функций в секции экспорта разделяются запятыми. Внимательный взгляд на пример экспорта в листинге 28.2 обнаруживает три различных варианта объявления.

В первом варианте компилятор самостоятельно определяет положение функции в таблице экспорта.

При использовании ключевого слова `index` следующее за ним число задает положение функции в таблице экспорта относительно других таких же функций.

Ключевое слово `name` позволяет экспортировать функцию под другим именем.

Соглашения о вызовах

При объявлении процедур и функций в динамических библиотеках используются различные соглашения о вызовах. Дело в том, что различные языки программирования по-разному реализуют передачу параметров в процедуру (через стек или регистры). Порядок следования параметров в стеке как раз определяется соглашением о вызовах.

Стандартный вызов в языках C++ и Object Pascal различается, но набор директив смены типа вызова позволяет обеспечить любую реализацию.

Во всех соглашениях о вызовах вызывающая процедура помещает параметры в стек. В зависимости от типа соглашения, очистка стека осуществляется вызывающей или вызываемой процедурой.

Если очистка стека выполняется вызывающей процедурой, то она успевает забрать из него возвращаемые значения.

Если очистка стека осуществляется вызываемой процедурой, то перед этим она помещает возвращаемые значения во временную область памяти.

Примечание

Помимо рассмотренных ниже директив имеются еще три типа вызовов, которые не используются и сохранены для обеспечения обратной совместимости. Это директивы `near`, `far`, `export`.

Директива *register*

Эта директива используется по умолчанию. Поэтому нет необходимости добавлять ключевое слов `register` после объявления функции. Вызов такого типа называется *быстрым* (fast call). В нем используются три расширенных регистра процессора, в которые помещаются переменные длиной не более 32-х разрядов и указатели. Остальные параметры помещаются в стек слева направо. После использования стек очищается вызываемой процедурой.

Директива *pascal*

Реализует вызовы в стиле языка Pascal. За очистку стека отвечает вызываемая процедура. Параметры помещаются в стек слева направо. Этот способ вызова является очень быстрым, но не поддерживает переменное число параметров. Используется для обеспечения обратной совместимости.

Директива *stdcall*

Параметры помещаются в стек слева направо. Очистка стека осуществляется вызываемой процедурой. Этот вызов обеспечивает обработку фиксированного числа параметров.

Директива *cdecl*

Реализует вызовы в стиле языка C. Параметры в стек помещаются справа налево. Очистка стека осуществляется вызывающей процедурой. Такие вызовы обеспечивают обслуживание переменного числа параметров, но скорость обработки меньше, чем в вызовах при реализации директивы `pascal`.

Эта директива в основном применяется для обращения к динамическим библиотекам, использующим соглашения о вызовах в стиле языка C. Использование директивы `cdecl` для библиотек Delphi не вызовет ошибку компиляции, но переменное число параметров не обеспечит.

Директива *safecall*

Параметры помещаются в стек справа налево. Очистка стека осуществляется вызываемой процедурой. Используется в COM и основанных на ней технологиях.

Инициализация и завершение работы DLL

При загрузке динамической библиотеки выполняется код инициализации, который расположен в блоке `begin..end` (см. листинги 28.1 и 28.2). Обычно здесь выполняются операции по заданию начальных значений используемых в функциях библиотеки переменных, проверка условий функционирования DLL, создание необходимых структур и объектов и т. д.

При возникновении ошибок выполнения кода инициализации можно воспользоваться специальной глобальной переменной `ExitCode` из модуля `System`. Если при возникновении исключительной ситуации присвоить этой переменной любое ненулевое значение, загрузка библиотеки прерывается.

Примечание

Любые объявленные в DLL глобальные переменные недоступны за ее пределами.

Оказывается, что при загрузке динамической библиотеки в адресное пространство вызывавшего ее процесса, происходят важные события, знание которых позволит вам эффективно управлять инициализацией и выгрузкой DLL.

Итак, перед запуском кода инициализации автоматически вызывается встроенная ассемблерная процедура `_InitDLL` (она расположена в модуле `System`). Она сохраняет состояние регистров процессора; получает значение экземпляра модуля библиотеки и записывает его в глобальную переменную `hInstance`; устанавливает для глобальной переменой `IsLibrary` значение `True` (по этому значению вы всегда сможете распознать код DLL); получает из стека ряд параметров; проверяет переменную процедурного типа `DLLProc`:

```
var DLLProc: Pointer;
```

Эта переменная используется для проверки вызовов операционной системой точки входа DLL. С этой переменной можно связать процедуру с одним целочисленным параметром. Такая процедура называется *функцией обратного вызова системного уровня*.

Если при проверке переменной `DLLProc` процедура `_InitDLL` находит связанную функцию обратного вызова, то она вызывается. При этом ей передается параметр, полученный из стека.

В качестве параметра могут быть переданы четыре значения:

```
const
  DLL_PROCESS_DETACH = 0;
  DLL_PROCESS_ATTACH = 1;
  DLL_THREAD_ATTACH  = 2;
  DLL_THREAD_DETACH  = 3;
```

Рассмотрим их.

- Значение `DLL_PROCESS_DETACH` передается при выгрузке DLL из адресного пространства процесса. Это происходит при явном вызове системной функции `FreeLibrary` *(см. ниже)* или при завершении процесса.
- Значение `DLL_PROCESS_ATTACH` означает, что библиотека отображается в адресное пространство процесса, который загружает ее в первый раз.
- Значение `DLL_THREAD_ATTACH` посылается всем загруженным в процесс динамическим библиотекам при создании нового потока. Обратите внимание, что при создании процесса и первичного потока посылается только одно значение `DLL_PROCESS_ATTACH`.

- Значение DLL_THREAD_DETACH посылается всем загруженным в процесс динамическим библиотекам при уничтожении существующего потока.

Впоследствии, при работе процесса с загруженной DLL, в случае возникновения описанных событий, функция обратного вызова вызывается снова и снова. При этом ей передается одно из рассмотренных значений.

Это хороший способ организовать в динамической библиотеке необходимую в каждом случае обработку. Как это сделать?

Во-первых, необходимо создать процедуру, подходящую для процедурного типа DLLProc, и написать для нее исходный код, применяемый в зависимости от переданного параметра.

Во-вторых, в секции инициализации нужно связать переменную DLLProc и созданную процедуру.

Применительно к рассматриваемому нами примеру, модернизированный исходный код библиотеки DataCheck будет выглядеть так:

Листинг 28.3. Часть исходного кода динамической библиотеки DataCheck с функцией обратного вызова

```
...
{Часть исходного кода опущена (см. листинг 24.2)}

exports
  IsValidInt,
  IsValidDate index 1,
  IsValidTime index 2 name 'ValidTime',

  procedure DLLEntryPoint(Reason: Integer);
  begin
   case Reason of
    DLL_PROCESS_ATTACH: ShowMessage('Первая загрузка DLL');
    DLL_PROCESS_DETACH:;
     DLL_THREAD_ATTACH: ShowMessage('Создан новый поток');
     DLL_THREAD_DETACH:;
   end;
  end;

begin
DLLProc := @DLLEntryPoint;
 DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

Процедура DLLEntryPoint обеспечивает простой показ сообщения о полученном значении параметра. В коде инициализации глобальной переменной

DLLProc передается адрес процедуры DLLEntryPoint. Затем эта процедура вызывается явно с параметром DLL_PROCESS_ATTACH.

У недоверчивого читателя может возникнуть вопрос — а зачем городить такие сложности, если можно просто использовать код в секции инициализации? Дело в том, что этот код выполняется только при запуске DLL. Поэтому, как, например, вовремя уничтожить создаваемые в библиотеке объекты при завершении ее работы? Для этого можно использовать функцию обратного вызова:

Листинг 28.4. Создание и удаление объекта при загрузке и выгрузке динамической библиотеки DataCheck

```
...
{Часть исходного кода опущена (см. листинг 24.2)}

exports
  IsValidInt,
  IsValidDate index 1,
  IsValidTime index 2 name 'ValidTime',

  type TSomeObject = class(TObject)
                       Field1: String;
                     end;
  var FirstObj: TSomeObject;

  procedure DLLEntryPoint(Reason: Word);
  begin
   case Reason of
    DLL_PROCESS_ATTACH: begin
                         FirstObj := TSomeObject.Create;
                         FirstObj.Field1 := 'Объект создан';
                         ShowMessage(FirstObj.Field1);
                        end;
    DLL_PROCESS_DETACH: FirstObj.Free;
     DLL_THREAD_ATTACH: ShowMessage('Создан новый поток');
     DLL_THREAD_DETACH:;
   end;
  end;

begin
 DLLProc := @DLLEntryPoint;
 DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

При завершении работы динамической библиотеки вызывается процедура, на которую указывает адрес, содержащийся в переменной `ExitProc`:

```
var ExitProc: Pointer;
```

Вызов DLL

Теперь рассмотрим, как из динамических библиотек вызываются функции.

При запуске исполняемого файла приложения операционная система создает для его работы отдельный процесс. Также система создает первичный поток, владельцем которого является процесс. Процесс приложения получает 4 Гбайт адресного пространства, в которое отображается исполняемый код приложения.

После этого из исполняемого кода извлекается информация обо всех вызываемых приложением динамических библиотеках и их функциях. Эта информация основывается на анализе исходного кода компоновщиком Delphi, который включает в исполняемый файл имена функций и динамических библиотек. При этом используется неявный вызов, описываемый ниже.

В результате при обращении приложения к функции из DLL вся информация о ней уже имеется в процессе. Для выполнения функции (вызов осуществляется одним из потоков процесса приложения) в адресное пространство процесса приложения загружается соответствующая динамическая библиотека. После этого исполняемый код DLL становится полностью доступен внутри процесса, но не вне его. Другие процессы могут загрузить эту же библиотеку и использовать ее образ в собственном адресном пространстве. Именно поэтому несколько приложений могут применять одну динамическую библиотеку одновременно.

Каждый поток имеет собственный стек, в который загружаются параметры функций DLL и все необходимые локальные переменные. Дело в том, что динамические библиотеки не имеют собственной кучи и не могут владеть данными. Поэтому любые создаваемые функциями DLL данные или объекты принадлежат вызывавшему потоку.

Функции динамических библиотек могут вызываться двумя способами — явным и неявным. Рассмотрим их.

Неявный вызов

Механизм неявного вызова наиболее прост, т. к. выполняется автоматически и основан на имеющейся в приложении информации о вызываемых функциях и динамических библиотеках. Однако разработчик не имеет возможности влиять на ход загрузки DLL. Если операционная система не смогла загрузить библиотеку, просто выдается сообщение об ошибке. Един-

ственный способ повлиять на процесс загрузки — использовать секцию инициализации библиотеки *(см. выше)*.

В качестве примера неявного вызова рассмотрим простое приложение DemoDLL1, использующее функции библиотеки DataCheck *(см. выше)*. Для этого в нем имеются три компонента `TEdit`, в которых осуществляется проверка введенной строки на соответствие формату одного из типов данных.

Примечание

Проекты DemoDLL1 и DataCheck объединены в одну группу. Переключение между проектами легко выполняется утилитой Диспетчер проектов.

Листинг 28.5. Модуль главной формы проекта DemoDLL1

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, comctrls, Buttons;

type
  TMainForm = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure Edit1Exit(Sender: TObject);
    procedure Edit2Exit(Sender: TObject);
    procedure Edit3Exit(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

  function IsValidInt(AText: String): Boolean; external 'DataCheck.dll';
  function IsValidDate(AText: String): Boolean; external 'DataCheck.dll';
  function ValidTime(AText: String): Boolean; external 'DataCheck.dll';
```

```
implementation

{$R *.DFM}

procedure TMainForm.Edit1Exit(Sender: TObject);
begin
 if not IsValidInt(Edit1.Text)
  then Edit1.Clear;
end;

procedure TMainForm.Edit2Exit(Sender: TObject);
begin
 if not IsValidDate(Edit2.Text)
  then Edit2.Clear;
end;

procedure TMainForm.Edit3Exit(Sender: TObject);
begin
 if not ValidTime(Edit3.Text)
  then Edit3.Clear;
end;

end.
```

Для организации неявного вызова достаточно объявить нужную функцию с директивой `external` и указать имя содержащей ее динамической библиотеки. Обратите внимание, что третья функция объявлена под псевдонимом `IsValidTime`, который объявлен для этой функции при помощи ключевого слова `name` в исходном коде динамической библиотеки.

В дальнейшем импортированные функции используются обычным образом.

Явный вызов

Явный вызов динамической библиотеки подразумевает создание программистом соответствующего исходного кода. Ему необходимо предусмотреть загрузку DLL, получение адресов переменных процедурного типа для используемых функций и процедур, выгрузку DLL.

Пример явного вызова функций динамической библиотеки имеется в демонстрационном приложении DemoDLL2, которое по выполняемым функциям полностью совпадает с предыдущим примером.

Листинг 28.6. Модуль главной формы проекта DemoDLL2

```
unit Unit2;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type

  StandardProc = function(AText: String): Boolean;

  TMainForm = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure FormShow(Sender: TObject);
    procedure Edit1Exit(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Edit2Exit(Sender: TObject);
    procedure Edit3Exit(Sender: TObject);
  private
    DLLHandle: THandle;
    LoadError: Word;
    IsValidInt: StandardProc;
    IsValidDate: StandardProc;
    ValidTime: StandardProc;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormShow(Sender: TObject);
begin
 DLLHandle := LoadLibrary('DataCheck');
 if DLLHandle = 0 then
 begin
  if GetLastError = ERROR_DLL_NOT_FOUND
   then ShowMessage('Ошибка загрузки DLL');
  Close;
 end;
```

```
 @IsValidInt := GetProcAddress(DLLHandle, 'IsValidInt');
 @IsValidDate := GetProcAddress(DLLHandle, 'IsValidDate');
 @ValidTime := GetProcAddress(DLLHandle, 'ValidTime');
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 if DLLHandle <> 0
  then FreeLibrary(DLLHandle);
end;

procedure TMainForm.Edit1Exit(Sender: TObject);
begin
  if not IsValidInt(Edit1.Text)
  then Edit2.Clear;
end;

procedure TMainForm.Edit2Exit(Sender: TObject);
begin
 if not IsValidDate(Edit2.Text)
  then Edit1.Clear;
end;

procedure TMainForm.Edit3Exit(Sender: TObject);
begin
 if not ValidTime(Edit3.Text)
  then Edit3.Clear;
end;

end.
```

Загрузка динамической библиотеки DataCheck осуществляется в методе-обработчике `FormShow` при помощи функции `LoadLibrary`. Имя динамической библиотеки может не содержать маршрута, если файл DLL расположен в одном каталоге с программой. Если в этом каталоге файл DLL не найден, поиск последовательно проводится в текущем каталоге, \SYSTEM и каталогах из перечня Path.

Так как для этой системной функции не создается исключительная ситуация, то следом предусмотрен контроль возможных ошибок. Функция `GetLastError` возвращает код последней ошибки.

Примечание

Код ошибки `ERROR_DLL_NOT_FOUND`, наряду со многими другими кодами, содержится в файле Windows.PAS.

Если библиотека успешно загружена, в три процедурные переменные типа `StandardProc` передаются адреса соответствующих функций DLL. Процедурный тип `StandardProc` объявлен перед классом формы. Для этого используется системная функция `GetProcAddress`.

В дальнейшем созданные таким образом функции применяются для вводимых значений в компонентах `TEdit`.

При закрытии приложения необходимо выгрузить все используемые динамические библиотеки при помощи системной функции `FreeLibrary`.

Ресурсы в DLL

Динамические библиотеки могут содержать не только исполняемый код, проводящий некоторые вычисления, но и ресурсы. Чаще всего бывает необходимо распространять вместе с DLL формы, обеспечивающие работу процедур и функций. Приемы работы с формами в проектах динамических библиотек ничем не отличаются от тех же приемов в проектах обычных приложений.

Единственная особенность заключается в том, что любая форма в DLL должна рассматриваться как создаваемая вручную, а не автоматически. При этом в процедуру, создающую форму, должен быть передан указатель на владельца будущей формы.

Например, процедура `ShowDemoForm` из рассматриваемой нами библиотеки DataCheck, выглядит так:

```
procedure ShowDemoForm(AOwner: TComponent);
  begin
   DemoForm := TDemoForm.Create(AOwner);
   DemoForm.ShowModal;
   DemoForm.Free;
  end;
```

Уничтожение формы можно организовать не только в самой процедуре, но и (при неоднократном применении) в другой процедуре или при выгрузке динамической библиотеки.

При вызове этой процедуры из приложения в параметре необходимо указать экземпляр класса приложения:

```
...
procedure ShowDemoForm(AOwner: TComponent); external 'DataCtrl.dll';
...
procedure TMainForm.BitBtn1Click(Sender: TObject);
begin
 ShowDemoForm(Application);
end;
```

Обратите внимание, что в данном случае форма из динамической библиотеки рассматривается операционной системой как отдельная задача, о чем свидетельствует системная панель задач.

Для распространения с приложением можно создавать специальные динамические библиотеки ресурсов, которые используются для локализации приложений. Например, в библиотеку ресурсов можно вынести все строковые константы (сообщения, тексты и т. д.), а с приложением распространять динамическую библиотеку ресурсов, строки в которой соответствуют языковым запросам заказчика.

Создать такую библиотеку можно, использовав Репозиторий Delphi (страница **New**) для проекта приложения или динамической библиотеки. Мастер создания библиотеки ресурсов проводит разработчика через все этапы создания проекта библиотеки.

Примечание

Для каждого языка необходимо создавать свои варианты форм и новый проект библиотеки ресурсов.

Перед началом создания проекта библиотеки ресурсов необходимо сохранить и откомпилировать базовый проект (для него создается проект локализации), а затем начать новый проект библиотеки ресурсов.

Первый диалог мастера библиотеки ресурсов предоставляет справочную информацию.

Второй — позволяет создать список форм базового проекта, которые войдут в библиотеку (рис. 28.2). При этом можно удалить из списка ненужные формы и добавить необходимые из других проектов.

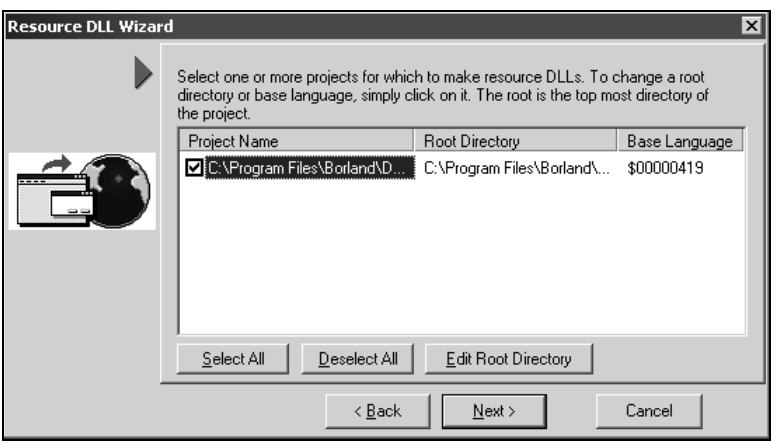


Рис. 28.2. Диалог мастера библиотеки ресурсов со списком форм, включаемых в проект

После этого в третьем диалоге мастера необходимо выбрать один или несколько языков локализации ресурсов (рис. 28.3). От этого выбора языка зависит расширение откомпилированного файла библиотеки и алгоритм поведения базового проекта при загрузке.

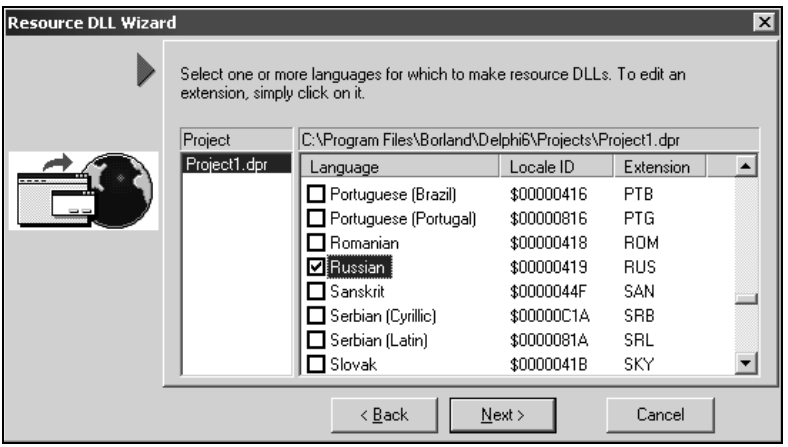


Рис. 28.3. Диалог мастера библиотеки ресурсов со списком доступных языков локализации проекта

Если запускаемое приложение или DLL находит в своей папке одноименный файл с расширением, которое соответствует одной из возможных локализаций и эта локализация применяется в системе, то приложение использует ресурсы из этой библиотеки вместо собственных. Поэтому при определении имени файла библиотеки ресурсов категорически не рекомендуется изменять предложенное мастером имя.

Кроме того, папку с файлом ресурсов можно задать дополнительно. Для этого на следующей странице мастера необходимо указать нужную папку для каждого языкового ресурса или принять предложенную по умолчанию (рис. 28.4).

Затем вы можете добавить к библиотеке ресурсов собственные файлы. Это могут быть ресурсы любого рода, используемые приложением. В окне мастера (рис. 28.5) необходимо выбрать эти файлы.

В последующих диалогах мастера задается способ создания или обновления для каждого языкового ресурса и запускается процесс создания ресурса. При первоначальном создании DLL ресурсы можно только создавать, впоследствии их можно полностью перезаписывать или изменять.

По завершении работы мастера для каждого выбранного языка создается новый проект библиотеки ресурсов. Результат работы мастера выводится в информационном окне (рис. 28.6).

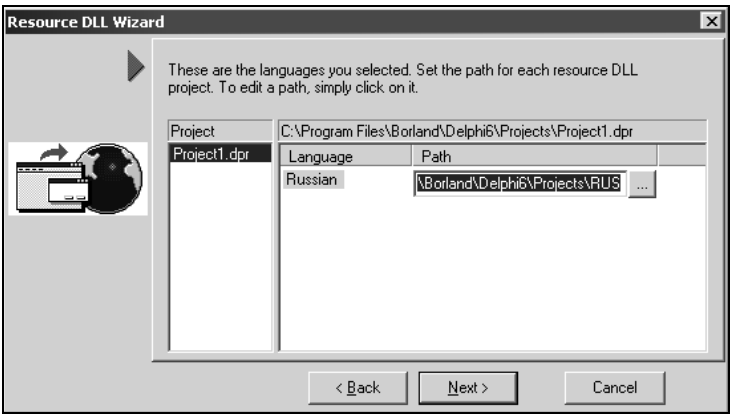


Рис. 28.4. Диалог мастера библиотеки ресурсов со списком папок для ресурсов локализации проекта

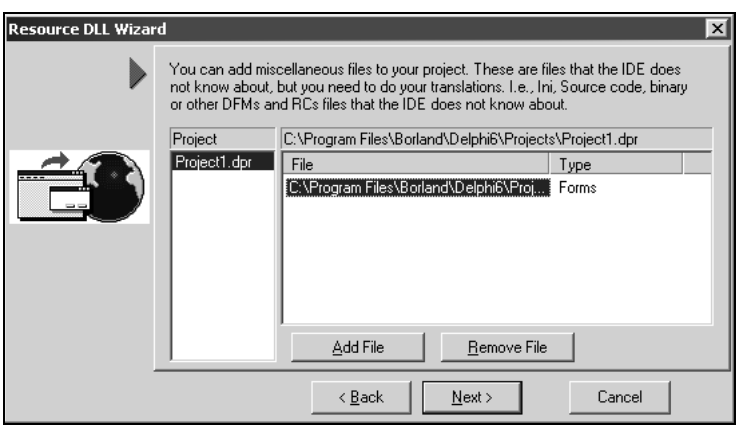


Рис. 28.5. Диалог мастера библиотеки ресурсов для включения в проект дополнительных файлов

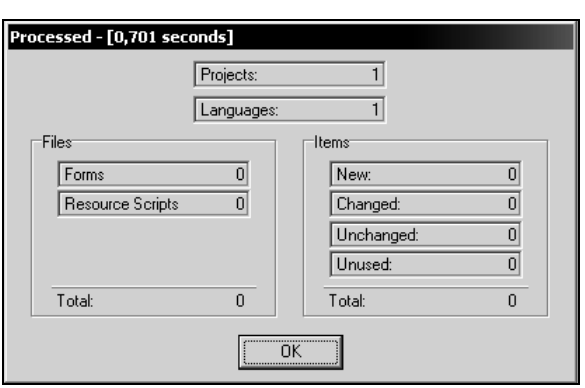


Рис. 28.6. Окно с информацией о результате создания ресурса

Каждый созданный проект необходимо откомпилировать и включить в состав дистрибутива базового приложения.

Использование модуля *ShareMem*

Если динамическая библиотека в процессе работы использует переменные или функции, осуществляющие динамическое выделение памяти под собственные нужды (длинные строки, динамические массивы, функции `New` и `GetMem`), а также, если такие переменные передаются в параметрах и возвращаются в результатах, то в таких библиотеках обязательно должен использоваться модуль `ShareMem`. При этом в секции `uses` модуль должен располагаться на первом месте. Об этом напоминает комментарий, автоматически добавляемый в файл динамической библиотеки при создании (см. листинг 28.1).

Управление этими операциями осуществляет специальный диспетчер печати BORLANDMM.DLL. Он должен распространяться вместе с динамическими библиотеками, использующими модуль `ShareMem`.

Резюме

Динамические библиотеки широко используются в ОС Windows. При их применении исполняемые файлы приложений становятся существенно меньше. К одной динамической библиотеке могут обращаться несколько программ одновременно. При этом динамические библиотеки могут использовать весь арсенал программных средств Delphi.

ГЛАВА 29

Потоки и процессы

Работая с Delphi, нужно иметь в виду: этот замечательный продукт не только упрощает разработку сложных приложений, он использует при этом все возможности операционной системы. Одна из возможностей, которую поддерживает Delphi, — это так называемые потоки (threads) или нити.

Потоки позволяют в рамках одной программы решать несколько задач одновременно. С недавних пор операционные системы для персональных компьютеров сделали это возможным.

Операционная система (ОС) предоставляет приложению некоторый интервал времени центрального процессора (ЦП) и в момент, когда приложение переходит к ожиданию сообщений или освобождает процессор, операционная система передает управление другой задаче. Теперь, когда компьютеры с более чем одним процессором резко упали в цене, а операционная система Windows NT может использовать наличие нескольких процессоров, пользователи действительно могут запускать одновременно более одной задачи. Планируя время центрального процессора, Windows 95 или Windows NT распределяют его между потоками, а не между приложениями. Чтобы использовать все преимущества, обеспечиваемые несколькими процессорами в современных операционных системах, программист должен знать, как создавать потоки.

В этой главе рассматриваются следующие вопросы:

- что такое потоки;
- разница между потоком и процессом;
- преимущества потоков;
- класс `TThread` в Delphi;
- реализация многопоточного приложения;
- синхронизация потоков.

Обзор потоков

Определение потока довольно простое: *потоки* — это объекты, получающие время процессора. Время процессора выделяется квантами (quantum, time slice). *Квант времени* — это интервал, имеющийся в распоряжении потока до тех пор, пока время не будет передано в распоряжение другого потока.

Обратите внимание, что кванты выделяются не программам или процессам, а порожденным ими потокам. Как минимум, каждый процесс имеет хотя бы один (главный) поток, но современные операционные системы, начиная с Windows 95 (для приверженцев Borland Kylix и Linux также), позволяют запустить в рамках процесса несколько потоков.

Если вы новичок в использовании потоков, самый простой пример их использования — приложения из состава Microsoft Office. К примеру, пакеты Excel и Word задействуют по несколько потоков. Word может одновременно корректировать грамматику и печатать, при этом осуществляя ввод данных с клавиатуры и мыши; программа Excel способна выполнять фоновые вычисления и печатать.

Примечание

Узнать число потоков, запущенных приложением, в Windows NT, 2000 и XP можно при помощи утилиты Task Manager (Диспетчер задач). Для этого среди показателей, отображаемых в окне **Processes**, нужно выбрать опцию **Thread Count**. Так, в момент написания этих строк MS Word использовал 5 потоков, среда Delphi — 3.

Вполне возможно, что эту главу сейчас вы читаете из чистого любопытства. Но, более вероятно, вы пришли в поиске ответов на конкретные проблемы. Какого же рода проблемы могут быть решены с применением потоков?

Если задачи приложения можно разделить на различные подмножества: обработка событий, ввод/вывод, связь и др., то потоки могут быть органично встроены в программное решение. Если разработчик может разделить большую задачу на несколько мелких, это только повысит переносимость кода и возможности его многократного использования.

Сделав приложение многопоточным, программист получает дополнительные возможности управления им. Например, через управление приоритетами потоков. Если один из них "притормаживает" приложение, занимая слишком много процессорного времени, его приоритет может быть понижен.

Другое важное преимущество внедрения потоков — при возрастании "нагрузки" на приложение можно увеличить количество потоков и тем самым снять проблему.

Потоки упрощают жизнь тем программистам, которые разрабатывают приложения в архитектуре клиент/сервер. Когда требуется обслуживание нового

клиента, сервер может запустить специально для этого отдельный поток. Такие потоки принято называть *симметричными потоками* (symmetric threads) — они имеют одинаковое предназначение, исполняют один и тот же код и могут разделять одни и те же ресурсы. Более того, приложения, рассчитанные на серьезную нагрузку, могут поддерживать *пул* (pool) однотипных потоков. Поскольку создание потока требует определенного времени, для ускорения работы желательно заранее иметь нужное число готовых потоков и активизировать их по мере подключения очередного клиента.

Примечание

Такой подход особенно характерен для Web-сервера Microsoft Internet Information Services и приложений, обрабатывающих запросы в его среде. Если вы создаете приложения ISAPI на Delphi, то можете использовать пулинг потоков, подключив к проекту модуль ISAPIThreadPool.pas. Если вы хотите позаимствовать идеи для других целей, ознакомьтесь с содержимым этого модуля.

Асимметричные потоки (asymmetric threads) — это потоки, решающие различные задачи и, как правило, не разделяющие совместные ресурсы. Необходимость в асимметричных потоках возникает:

- когда в программе необходимы длительные вычисления, при этом необходимо сохранить нормальную реакцию на ввод;
- когда нужно обрабатывать асинхронный ввод/вывод с использованием различных устройств (COM-порта, звуковой карты, принтера и т. п.);
- когда вы хотите создать несколько окон и одновременно обрабатывать ввод в них.

Потоки и процессы

Когда мы говорим "программа" (application), то обычно имеем в виду понятие, в терминологии операционной системы обозначаемое как "процесс". Процесс состоит из виртуальной памяти, исполняемого кода, потоков и данных. Процесс может содержать много потоков, но обязательно содержит, по крайней мере, один. Поток, как правило, имеет "в собственности" минимум ресурсов; он зависит от процесса, который и распоряжается виртуальной памятью, кодом, данными, файлами и другими ресурсами ОС.

Почему мы используем потоки вместо процессов, хотя, при необходимости, приложение может состоять и из нескольких процессов? Дело в том, что переключение между процессами — значительно более трудоемкая операция, чем переключение между потоками. Другой довод в пользу использования потоков — то, что они специально задуманы для разделения ресурсов; разделить ресурсы между процессами (имеющими раздельное адресное пространство) не так-то просто.

Фоновые процедуры, или способ обойтись без потоков

Здесь мы рассмотрим возможность для организации фоновых действий (job) внутри однопоточной программы с сохранением реакции этого потока на события от мыши и клавиатуры.

Еще не столь давно программисты пытались эмулировать потоки, запуская процедуры внутри цикла обработки сообщений Windows. *Цикл обработки сообщений* (или *цикл ожидания*) — это особый фрагмент кода в программе, управляемой событиями. Он исполняется тогда, когда программа находит в очереди события, которые нужно обработать; если таковых нет, программа может выполнить в это время "фоновую процедуру". Такой способ имитации потоков весьма сложен, т. к. вынуждает программиста, во-первых, сохранять состояние фоновой процедуры между ее вызовами, а во-вторых, определять момент, когда она вернет управление обработчику событий. Если такая процедура выполняется долго, то у пользователя может сложиться впечатление, что приложение перестало реагировать на внешние события. Использование потоков снимает проблему переключения контекста, теперь контекст (стек и регистры) сохраняет операционная система.

В Delphi возможность создать фоновую процедуру реализована через событие `OnIdle` объекта `Application`:

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean)
of object;
property OnIdle: TIdleEvent;
```

Обработчик этого события вы можете написать, поместив на форму компонент `TApplicationEvents` со страницы `Additional` Палитры компонентов.

Чтобы сделать в фоновом режиме какую-то работу, следует разбить ее на кванты и выполнять по одному кванту каждый вызов `OnIdle` — иначе приложение будет плохо реагировать на внешние воздействия.

Приоритеты потоков

Интерфейс Win32 API позволяет программисту управлять распределением времени между потоками; это распространяется и на приложения, написанные на Delphi. Операционная система планирует время процессора в соответствии с приоритетами потоков.

Приоритет потока — величина, складывающаяся из двух составных частей: приоритета породившего поток процесса и собственно приоритета потока. Когда поток создается, ему назначается приоритет, соответствующий приоритету породившего его процесса.

В свою очередь, процессы могут иметь следующие классы приоритетов.

- `Real time`;
- `High`;
- `Above normal`;
- `Normal`;
- `Below normal`;
- `Idle`.

Примечание

Классы `Above normal` и `Below normal` появились впервые в Windows 2000.

Класс реального времени задает приоритет даже больший, чем у многих процессов операционной системы. Такой приоритет нужен для процессов, обрабатывающих высокоскоростные потоки данных. Если такой процесс не завершится за короткое время, пользователь почувствует, что система перестала откликаться, т. к. даже обработка событий мыши не получит времени процессора.

Использование класса `High` ограничено процессами, которые должны завершаться за короткое время, чтобы не вызвать сбойной ситуации. Пример — процесс, который посылает сигналы внешнему устройству; причем устройство отключается, если не получит своевременный сигнал. Если у вас возникли проблемы с производительностью вашего приложения, было бы неправильно решать их просто за счет повышения его приоритета до high — такой процесс также влияет на всю ОС. Возможно, в этом случае следует модернизировать компьютер.

Большинство процессов запускается в рамках класса с нормальным приоритетом. Нормальный приоритет означает, что процесс не требует какого-либо специального внимания со стороны операционной системы.

И наконец, процессы с фоновым приоритетом запускаются лишь в том случае, если в очереди Диспетчера задач нет других процессов. Обычные виды приложений, использующие такой приоритет, — это программы сохранения экрана и системные агенты (system agents). Программисты могут использовать фоновые процессы для организации завершающих операций и реорганизации данных. Примерами могут служить сохранение документа или резервное копирование базы данных.

Приоритеты имеют значения от 0 до 31. Процесс, породивший поток, может впоследствии изменить его приоритет; в этой ситуации программист имеет возможность управлять скоростью отклика каждого потока.

Базовый приоритет нити складывается из двух составляющих, однако это не означает, что он просто равен их сумме. Взгляните на соответствующие величины, которые показаны в табл. 29.1. Для потока, имеющего собственный приоритет `THREAD_PRIORITY_IDLE`, базовый приоритет будет равен 1, невзирая на приоритет породившего его процесса.

И еще для класса `Normal` приведены по два приоритета, снабженные буквами B (Background) и F (Foreground). Объяснение этому дается ниже.

Таблица 29.1. *Классы процессов и приоритеты их потоков (для Windows 2000 и XP)*

	IDLE_PRIORITY_CLASS	BELOW_NORMAL_PRIORITY_CLASS	NORMAL_PRIORITY_CLASS	ABOVE_NORMAL_PRIORITY_CLASS	HIGH_PRIORITY_CLASS	REALTIME_PRIORITY_CLASS
THREAD_PRIORITY_IDLE	1	1	1	1	1	16
THREAD_PRIORITY_LOWEST	2	4	5 (B) 7 (F)	8	11	22
THREAD_PRIORITY_BELOW_NORMAL	3	5	6 (B) 8 (F)	9	12	23
THREAD_PRIORITY_NORMAL	4	6	7 (B) 9 (F)	10	13	24
THREAD_PRIORITY_ABOVE_NORMAL	5	7	8 (B) 10 (F)	11	14	25
THREAD_PRIORITY_HIGHEST	6	8	9 (B) 11(F)	12	15	26
THREAD_PRIORITY_TIME_CRITICAL	15	15	15	15	15	31

Помимо базового приоритета, описываемого в этой таблице, планировщик заданий (scheduler) может назначать так называемые динамические приоритеты. Для процессов класса `NORMAL_PRIORITY_CLASS` при переключении из фонового режима в режим переднего плана и в ряде других случаев приоритет потока, с которым создано окно переднего плана, повышается. Так работают все клиентские операционные системы от Microsoft. Серверные операционные системы оптимизированы для выполнения фоновых приложений. Впрочем, Windows NT и более поздние ОС на этом ядре позволяют переключать режим оптимизации, используя переключатель **Application response** апплета System панели управления Windows (рис. 29.1).

К тому же Windows 2000 Professional и Windows 2000 Server имеют разные алгоритмы выделения квантов времени. Первая — клиентская — операци-

онная система выделяет время короткими квантами переменной длины для ускорения реакции на приложения переднего плана (foreground). Для сервера же более важна стабильная работа системных служб, поэтому во второй ОС система распределяет длинные кванты постоянной длины.

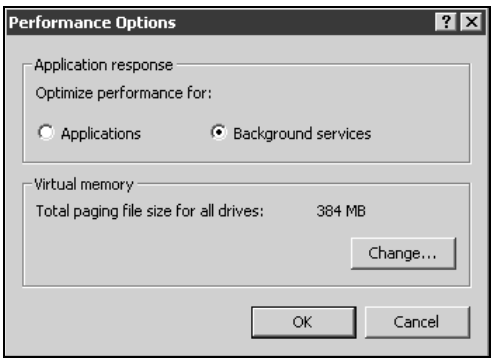


Рис. 29.1. С помощью диалога **Performance Options** можно управлять алгоритмом назначения приоритетов

Теперь, разобравшись в приоритетах потоков, нужно обязательно сказать о том, как же их использует планировщик заданий для распределения процессорного времени.

Операционная система имеет различные очереди готовых к выполнению потоков — для каждого уровня приоритета свой. В момент распределения нового кванта времени она просматривает очереди — от высшего приоритета к низшему. Готовый к выполнению поток, стоящий первым в очереди, получает этот квант и перемещается в хвост очереди. Поток будет исполняться всю продолжительность кванта, если не произойдет одно из двух событий:

- выполняющийся поток остановился для ожидания;
- появился готовый к выполнению поток с более высоким приоритетом.

Теперь, наверное, вам более ясна опасность, исходящая от неоправданного завышения приоритетов. Ведь, если есть активные потоки с высоким приоритетом, ни один поток с более низким приоритетом ни разу не получит времени процессора. Эта проблема может подстерегать вас даже на уровне вашего приложения. Предположим, вы назначили вычислительному потоку приоритет `THREAD_PRIORITY_ABOVE_NORMAL`, а потоку, где обрабатывается ввод пользователя, — `THREAD_PRIORITY_BELOW_NORMAL`. Тогда вместо запланированного результата — совместить вычисления с нормальной реакцией приложения — вы получите строго обратный. Приложение вообще перестанет откликаться на ввод, и снять его будет возможно только с помощью средств ОС.

Так что нормальная практика для асимметричных потоков — это назначение потоку, обрабатывающему ввод, более высокого приоритета, а всем остальным — более низкого или даже приоритета `Idle`, если этот поток должен выполняться только во время простоя системы.

Класс *TThread*

Delphi представляет программисту полный доступ к возможностям программирования интерфейса Win32. Для чего же тогда фирма Borland представила специальный класс для организации потоков? Вообще говоря, программист не обязан разбираться во всех тонкостях механизмов, предлагаемых операционной системой. Класс должен инкапсулировать и упрощать программный интерфейс; класс `TThread` — прекрасный пример предоставления разработчику простого доступа к программированию потоков. Сам API потоков, вообще говоря, не очень сложен, но предоставленные классом `TThread` возможности вообще замечательно просты. В двух словах, все, что вам необходимо сделать, — это перекрыть виртуальный метод `Execute`.

Другая отличительная черта класса `TThread` — это гарантия безопасной работы с библиотекой визуальных компонентов VCL. Без использования класса `TThread` во время вызовов VCL могут возникнуть ситуации, требующие специальной синхронизации *(см. разд. "Проблемы при синхронизации потоков" далее в этой главе)*.

Нужно отдавать себе отчет, что с точки зрения операционной системы поток — это ее объект. При создании он получает дескриптор и отслеживается ОС. Объект класса `TThread` — это конструкция Delphi, соответствующая потоку ОС. Этот объект VCL создается до реального возникновения потока в системе и уничтожается после его исчезновения.

Изучение класса `TThread` начнем с метода `Execute`:

```
procedure Execute; virtual; abstract;
```

Это и есть код, исполняемый в создаваемом вами потоке `TThread`.

Примечание

Хотя формальное описание `Execute` — метод `abstract`, но мастер создания нового объекта `TThread` создает для вас пустой шаблон этого метода.

Переопределяя метод `Execute`, мы можем тем самым закладывать в новый потоковый класс то, что будет выполняться при его запуске. Если поток был создан с аргументом `CreateSuspended`, равным `False`, то метод `Execute` выполняется немедленно, в противном случае `Execute` выполняется после вызова метода `Resume` *(см. описание конструктора ниже)*.

Если поток рассчитан на однократное выполнение каких-либо действий, то никакого специального кода завершения внутри `Execute` писать не надо.

Если же в потоке будет выполняться какой-то цикл, и поток должен завершиться вместе с приложением, то условия окончания цикла должны быть примерно такими:

```
procedure TMyThread.Execute;
begin
repeat
  DoSomething;
Until CancelCondition or Terminated;
end;
```

Здесь `CancelCondition` — ваше личное условие завершения потока (исчерпание данных, окончание вычислений, поступление на вход того или иного символа и т. п.), а свойство `Terminated` сообщает о завершении потока (это свойство может быть установлено как изнутри потока, так и извне; скорее всего, завершается породивший его процесс).

Конструктор объекта:

```
constructor Create(CreateSuspended: Boolean);
```

получает параметр `CreateSuspended`. Если его значение равно `True`, вновь созданный поток не начинает выполняться до тех пор, пока не будет сделан вызов метода `Resume`. В случае, если параметр `CreateSuspended` имеет значение `False`, конструктор завершается и только затем поток начинает исполнение.

```
destructor Destroy; override;
```

Деструктор `Destroy` вызывается, когда необходимость в созданном потоке отпадает. Деструктор завершает его и высвобождает все ресурсы, связанные с объектом `TThread`.

```
function Terminate: Integer;
```

Для окончательного завершения потока (без последующего запуска) существует метод `Terminate`. Но если вы думаете, что этот метод делает какие-то принудительные действия по остановке потока, вы ошибаетесь. Все, что происходит, — это установка свойства

```
property Terminated: Boolean;
```

в значение `True`. Таким образом, `Terminate` — это указание потоку завершиться, выраженное "в мягкой форме", с возможностью корректно освободить ресурсы. Если вам нужно немедленно завершить поток, используйте функцию Windows API `TerminateThread`.

Примечание

Метод `Terminate` автоматически вызывается и из деструктора объекта. Поток — объект VCL будет дожидаться, пока завершится поток — объект операционной системы. Таким образом, если поток не умеет завершаться корректно, вызов деструктора потенциально может привести к зависанию всей программы.

Еще одно полезное свойство:

```
property FreeOnTerminate: Boolean;
```

Если это свойство равно `True`, то деструктор потока будет вызван автоматически по его завершении. Это очень удобно для тех случаев, когда вы в своей программе не уверены точно, когда именно завершится поток, и хотите использовать его по принципу "выстрелил и забыл" (fire and forget).

```
function WaitFor: Integer;
```

Метод `WaitFor` предназначен для синхронизации и позволяет одному потоку дождаться момента, когда завершится другой поток. Если вы внутри потока `FirstThread` пишете код:

```
Code := SecondThread.WaitFor;
```

то это означает, что поток `FirstThread` останавливается до момента завершения потока `SecondThread`. Метод `WaitFor` возвращает код завершения ожидаемого потока (см. свойство `ReturnValue`).

```
property Handle: THandle read FHandle;
property ThreadID: THandle read FThreadID;
```

Свойства `Handle` и `ThreadID` дают программисту непосредственный доступ к потоку средствами API Win32. Если разработчик хочет обратиться к потоку и управлять им, минуя возможности класса `TThread`, значения `Handle` и `ThreadID` могут быть использованы в качестве аргументов функций Win32 API. Например, если программист хочет перед продолжением выполнения приложения дождаться завершения сразу нескольких потоков, он должен вызвать функцию API `WaitForMultipleObjects`; для ее вызова необходим массив дескрипторов потоков.

```
property Priority: TThreadPriority;
```

Свойство `Priority` позволяет запросить и установить приоритет потоков. Приоритеты потоков в деталях описаны выше. Допустимыми значениями приоритета для объектов `TThread` являются `tpIdle`, `tpLowest`, `tpLower`, `tpNormal`, `tpHigher`, `tpHighest` и `tpTimeCritical`.

```
procedure Synchronize(Method: TThreadMethod);
```

Этот метод относится к секции `protected`, т. е. может быть вызван только из потомков `TThread`. Delphi предоставляет программисту метод `Synchronize` для

безопасного вызова методов VCL внутри потоков. Во избежание конфликтных ситуаций, метод `Synchronize` дает гарантию, что к каждому объекту VCL одновременно имеет доступ только один поток. Аргумент, передаваемый в метод `Synchronize`, — это имя метода, который производит обращение к VCL; вызов `Synchronize` с этим параметром — это то же, что и вызов самого метода. Такой метод (класса `TThreadMethod`) не должен иметь никаких параметров и не должен возвращать никаких значений. К примеру, в основной форме приложения нужно предусмотреть функцию

```
procedure TMainForm.SyncShowMessage;
begin
  ShowMessage(IntToStr(ThreadList1.Count));
// другие обращения к VCL
end;
```

а в потоке для показа сообщения писать не

```
ShowMessage(IntToStr(ThreadList1.Count));
```

и даже не

```
MainForm.SyncShowMessage;
```

а только так:

```
Synchronize(MainForm.SyncShowMessage);
```

Примечание

Производя любое обращение к объекту VCL из потока, убедитесь, что при этом используется метод `Synchronize`; в противном случае результаты могут оказаться непредсказуемыми. Это верно даже в том случае, если вы используете средства синхронизации, описанные ниже.

```
procedure Resume;
```

Метод `Resume` класса `TThread` вызывается, когда поток возобновляет выполнение после остановки, или для явного запуска потока, созданного с параметром `CreateSuspended`, равным `True`.

```
procedure Suspend;
```

Вызов метода `Suspend` приостанавливает поток с возможностью повторного запуска впоследствии. Метод `Suspend` приостанавливает поток вне зависимости от кода, исполняемого потоком в данный момент; выполнение продолжается с точки останова.

```
property Suspended: Boolean;
```

Свойство `Suspended` позволяет программисту определить, не приостановлен ли поток. С помощью этого свойства можно также запускать и останавли-

вать поток. Установив свойство `Suspended` в значение `True`, вы получите тот же результат, что и при вызове метода `Suspend` — приостановку. Наоборот, установка свойства `Suspended` в значение `False` возобновляет выполнение потока, как и вызов метода `Resume`.

```
property ReturnValue: Integer;
```

Свойство `ReturnValue` позволяет узнать и установить значение, возвращаемое потоком по его завершении. Эта величина полностью определяется пользователем. По умолчанию поток возвращает ноль, но если программист захочет вернуть другую величину, то простая переустановка свойства `ReturnValue` внутри потока позволит получить эту информацию другим потокам. Это, к примеру, может пригодиться, если внутри потока возникли проблемы, или с помощью свойства `ReturnValue` нужно вернуть число не прошедших орфографическую проверку слов.

На этом завершим подробный обзор класса `TThread`. Для более близкого знакомства с потоками и классом Delphi `TThread` создадим многопоточное приложение. Для этого нужно написать всего несколько строк кода и несколько раз щелкнуть мышью.

Пример создания многопоточного приложения в Delphi

Этот раздел содержит описание шагов, необходимых для создания простого, но показательного примера многопоточного приложения. Мы будем пытаться вычислить число "пи" с максимальной точностью после запятой. Конечно, встроенная в Delphi константа `Pi` имеет достаточную точность, правильнее сказать — максимальную, допускаемую самым точным 10-байтным форматом для вещественных чисел `Extended`. Так что превзойти ее нам не удастся. Но этот пример использования потоков может послужить прологом для решения реальных задач.

Первый пример будет содержать два потока: главный (обрабатывающий ввод пользователя) и вычислительный; мы сможем изменять их свойства и наблюдать за реакцией.

Итак, выполните следующую последовательность действий:

1. В среде Delphi откройте меню **File** и выберите пункт **New Application**.
2. Расположите на форме пять меток и один переключатель, как показано на рис. 29.2.

   Переименуйте главную форму в `fmMain`.
3. Откройте меню **File** и выберите пункт **Save Project As**. Сохраните модуль как `uMain`, а проект — как `Threads1`.

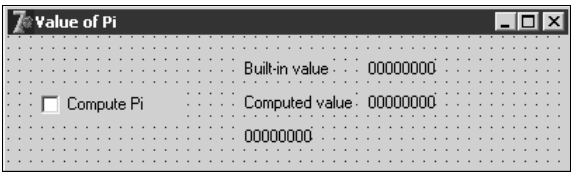


Рис. 29.2. Внешний вид формы для приложения Threads1

4. Откройте меню **File** и выберите пункт **New**. Затем дважды щелкните на объекте типа поток (значок **Thread Object**). Откроется диалоговое окно **New Items**, показанное на рис. 29.3.

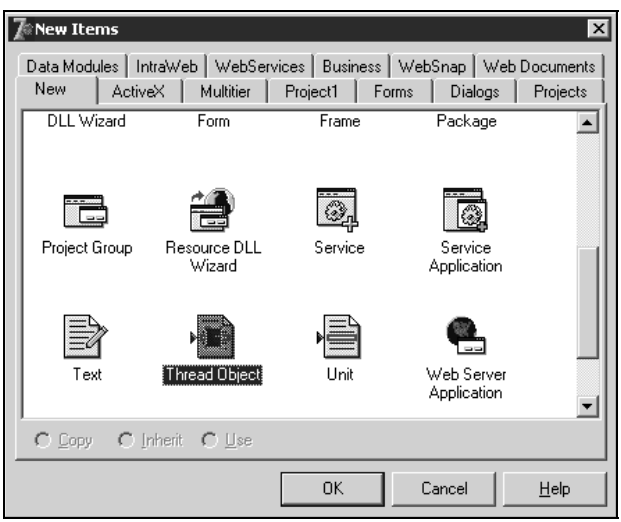


Рис. 29.3. Диалоговое окно **New Items** с выбранным объектом типа "поток"

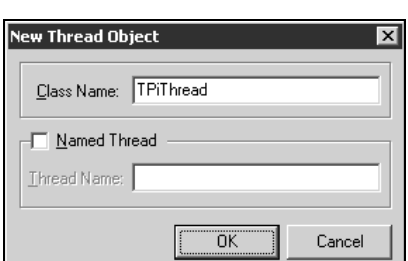


Рис. 29.4. Диалоговое окно **New Thread Object**

5. Когда появится диалоговое окно для именования объекта поток, введите `TPiThread` и нажмите клавишу <Enter> (рис. 29.4). Помимо этого, при желании, вы можете присвоить создаваемому потоку имя, установив флажок **Named Thread** и задав имя в поле **Thread Name**. Так как имя по-

тока используется только для удобства обозначения, эту возможность мы использовать не будем.

Delphi создаст новый модуль и поместит в него шаблон для нового потока.

6. Код, вносимый в метод `Execute`, вычисляет число π, используя сходимость бесконечного ряда Лейбница:

   $\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - \ldots$

   Разумеется, отображать новое значение после каждой итерации — это то же самое, что стрелять из пушки по воробьям. На отображение информации система потратит в десятки раз больше времени, чем на собственно вычисления. Поэтому мы ввели константу `UpdatePeriod`, которая регулирует периодичность отображения текущего значения.

   Код метода `Execute` показан ниже:

```
const
 // Лучше использовать нечетное число для того, чтобы избежать эффекта
 // мерцания
 UpdatePeriod = 1000001;

procedure TPiThread.Execute;
var sign : Integer;
  PiValue, PrevValue : Extended;
  i : Int64;
begin
  { Place thread code here }
  PiValue := 4;
  sign := -1;
  i := 0;
  repeat
  Inc(i);
  PrevValue := PiValue;
  PiValue := PiValue + sign * 4 / (2*i+1);
  sign := -sign;
  if i mod UpdatePeriod = 0 then
   begin
    GlobalPi := PiValue;
    GlobalCounter := i;
    Synchronize(fmMain.UpdatePi);
   end;
  until Terminated or
  (Abs(PiValue - PrevValue)<1E-19);
end;
```

7. Откройте меню **File** и выберите пункт **Save As**. Сохраните модуль с потоком как uPiThread.pas.

8. Отредактируйте главный файл модуля uMain.pas и добавьте модуль `uPiThread` к списку используемых модулей в секции интерфейса. Он должен выглядеть так:

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, StdCtrls, uPiThread;
```

9. В секции `public` формы `TfmMain` добавьте ссылку на создаваемую нить:

```
 PiThread : TPiThread;
```

10. Добавьте в модуль `uMain` две глобальные переменные

```
GlobalPi : Extended;
GlobalCounter : Int64;
```

и метод `UpdatePi`:

```
procedure TfmMain.UpdatePi;
begin
  if IsIconic(Application.Handle) then Exit;
  LaValue.Caption := FloatToStrF(GlobalPi, ffFixed, 18, 18);
  laIterNum.Caption := IntToStr(GlobalCounter) + ' iterations';
end;
```

Этот метод, если вы обратили внимание, вызывается из потока посредством процедуры `Synchronize`. Он отображает текущее значение приближения к числу "пи" и количество итераций.

В случае, если главное окно приложения свернуто, отображение не производится; так что после его развертывания вам, возможно, придется подождать некоторое время для обновления.

11. Выполните двойной щелчок на свободном месте рабочей области формы, при этом создастся шаблон метода `FormCreate`. Здесь мы отобразим значение системной константы `Pi`:

```
procedure TfmMain.FormCreate(Sender: TObject);
begin
 laBuiltIn.Caption := FloatToStrF(Pi, ffFixed, 18, 18);
end;
```

12. Выберите на форме переключатель (его название `cbCalculate`) и назначьте событию `OnClick` код, создающий и уничтожающий вычислительный поток в зависимости от состояния переключателя:

```
procedure TfmMain.cbCalculateClick(Sender: TObject);
begin
 if cbCalculate.Checked then
```

```
    begin
     PiThread := TPiThread.Create(True);
     PiThread.FreeOnTerminate := True;
     PiThread.Priority := tpLower;
     PiThread.Resume;
    end
   else
    begin
     if Assigned(PiThread) then PiThread.Terminate;
    end;
  end;
```

Таким образом, многопоточное приложение готово к запуску. Если все пройдет нормально, вы увидите картинку, подобную той, которая приведена на рис. 29.5.

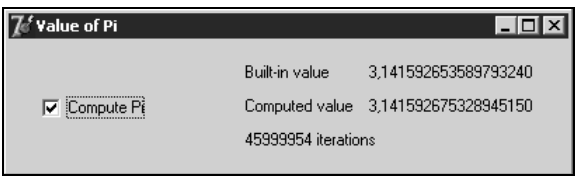


Рис. 29.5. Выполняющееся приложение Threads1

Пока один из авторов писал текст этого раздела, запущенное одновременно приложение Threads1 выполнило пять миллиардов итераций и приблизилось к встроенному значению `Pi` в десятом разряде. Интересно, насколько хватит терпения у вас?

Этот простой пример — первый шаг в усвоении того, как от базового класса `TThread` можно порождать собственные классы. Из-за своей простоты он не лишен недостатков; более того — если бы вычислительных нитей было не одна, а более, кое-какие приемы были бы даже ошибочными. Но — об этом ниже.

Проблемы при синхронизации потоков

К сожалению, простота создания потоков подчас "компенсируется" сложностью их применения. Две типичные проблемы, с которыми программист может столкнуться при работе с потоками, — это *тупики* (deadlocks) и *гонки* (race conditions).

Тупики

Вероятно, вы не раз наблюдали на трамвайной остановке следующую забавную картину (рис. 29.6).

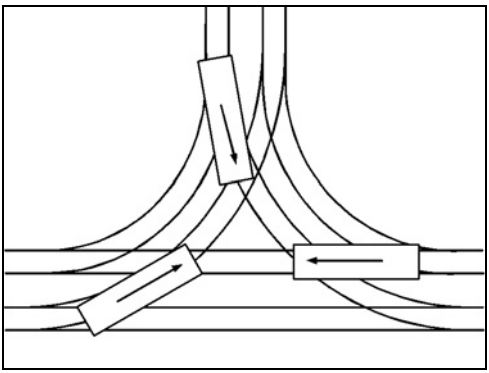


Рис 29.6. Ситуации тупиков возникают не только в программировании

Рисунок дает исчерпывающее пояснение ситуации тупиков. Тупики имеют место, когда поток ожидает ресурс, который в данный момент принадлежит другому потоку. Рассмотрим пример. Поток 1 захватывает ресурс А, и для того чтобы продолжать работу, ждет возможности захватить ресурс Б. В то же время Поток 2 захватывает ресурс Б и ждет возможности захватить ресурс А. Развитие этого сценария заблокирует оба потока; ни один из них не будет исполняться. Ресурсами могут выступать любые совместно используемые объекты системы — файлы, массивы в памяти, устройства ввода/вывода и т. п.

В ситуации на картинке три трамвая захватили по одному ресурсу (перекрестку) и пытаются захватить еще один, что, очевидно, невозможно без освобождения уже захваченных. В жизни ситуация разрешилась просто — самый молодой из водителей был вынужден отъехать. В информационных технологиях все бывает сложнее. Откройте любой документ, сопровождающий очередной пакет обновления к любой версии Windows. Очень часто там можно найти информацию об одной-двух исправленных ситуациях тупиков.

Гонки

Ситуация гонок возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние. Рассмотрим следующий пример. Пусть Поток 1 получил доступ к ресурсу и изменил его в своих интересах; затем активизировался Поток 2 и модифицировал этот же ресурс до завершения Потока 1. Поток 1 полагает, что ресурс остался в том же состоянии, в каком был до переключения. В зависимости от того, когда именно был изменен ресурс, результаты могут варьироваться — иногда код будет выполняться нормально, иногда нет. Программисты не должны строить никаких гипотез относительно порядка исполнения потоков, т. к. планировщик ОС может запускать и останавливать их в любое время.

```
Inc(i);
if i = iSomething then
 DoSomething;
```

Здесь `i` — глобальная переменная, доступная из обоих потоков. Пусть два или более потоков исполняют этот код одновременно. Поток 1 инкрементировал значение переменной `i` и хочет проверить ее значение для выполнения тех или иных условий. Но тут активизируется другой поток, который еще увеличивает значение `i`. В результате первый поток "проскакивает" мимо условия, которое, казалось бы, должно было быть выполнено.

Возникновения как ситуаций гонок, так и тупиков можно избежать, если использовать приемы, обсуждаемые ниже.

Средства синхронизации потоков

Проще всего говорить о синхронизации, если создаваемый поток не взаимодействует с ресурсами других потоков и не обращается к VCL. Допустим, у вас на компьютере несколько процессоров, и вы хотите "распараллелить" вычисления. Тогда вполне уместен следующий код:

```
MyCompThread := TComputationThread.Create(False);
// Здесь можно что-нибудь делать, пока второй поток производит вычисления
DoSomeWork;
// Теперь ожидаем его завершения
MyCompThread.WaitFor;
```

Приведенная схема совершенно недопустима, если во время своей работы поток `MyCompThread` обращается к VCL посредством метода `Synchronize`. В этом случае поток ждет главный поток для обращения к VCL, а тот, в свою очередь, его — классический тупик.

За "спасением" следует обратиться к программному интерфейсу Win32. Он предоставляет богатый набор инструментов, которые могут понадобиться для организации совместной работы потоков.

Главные понятия для понимания механизмов синхронизации — функции ожидания и объекты синхронизации. В Windows API предусмотрен ряд функций, позволяющих приостановить выполнение вызвавшего эту функцию потока вплоть до того момента, как будет изменено состояние какого-то объекта, называемого *объектом синхронизации* (под этим термином здесь понимается не объект Delphi, а объект операционной системы). Простейшая из этих функций — `WaitForSingleObject` — предназначена для ожидания одного объекта.

К возможным вариантам относятся четыре объекта, которые разработаны специально для синхронизации: событие (event), взаимное исключение (mutex), семафор (semaphore) и таймер (timer).

Но кроме специальных объектов можно организовать ожидание и других объектов, дескриптор которых используется в основном для иных целей, но может применяться и для ожидания. К ним относятся: процесс (process), поток (thread), оповещение об изменении в файловой системе (change notification) и консольный ввод (console input).

Косвенно к этой группе может быть добавлена критическая секция (critical section).

Примечание

Перечисленные выше средства синхронизации в основном инкапсулированы в состав классов Delphi. У программиста есть две альтернативы. С одной стороны, в состав библиотеки VCL включен модуль SYNCOBJS.PAS, содержащий классы для события (`TEvent`) и критической секции (`TCriticalSection`). С другой, с Delphi поставляется отличный пример IPCDEMOS, который иллюстрирует проблемы взаимодействия процессов и содержит модуль IPCTHRD.PAS с аналогичными классами — для того же события, взаимного исключения (`TMutex`), а также совместно используемой памяти (`TSharedMem`).

Перейдем к подробному описанию объектов, используемых для синхронизации.

Событие

Объект типа *событие* (event) — простейший выбор для задач синхронизации. Он подобен дверному звонку — звенит до тех пор, пока его кнопка находится в нажатом состоянии, извещая об этом факте окружающих. Аналогично, и объект может быть в двух состояниях, а "слышать" его могут многие потоки сразу.

Класс `TEvent` (модуль SYNCOBJS.PAS) имеет два метода: `SetEvent` и `ResetEvent`, которые переводят объект в активное и пассивное состояние соответственно. Конструктор имеет следующий вид:

```
constructor Create(EventAttributes: PSecurityAttributes; ManualReset,
InitialState: Boolean; const Name: string);
```

Здесь параметр `InitialState` — начальное состояние объекта, `ManualReset` — способ его сброса (перевода в пассивное состояние). Если этот параметр равен `True`, событие должно быть сброшено вручную. В противном случае событие сбрасывается по мере того, как стартует хоть один поток, ждавший данный объект.

На третьем методе:

```
TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);
function WaitFor(Timeout: DWORD): TWaitResult;
```

остановимся подробнее. Он дает возможность ожидать активизации события в течение `Timeout` миллисекунд. Как вы могли догадаться, внутри этого

метода происходит вызов функции `WaitFotSingleObject`. Типичных результатов на выходе `WaitFor` два — `wrSignaled`, если произошла активизация события, и `wrTimeout`, если за время тайм-аута ничего не произошло.

Примечание

Если нужно (и допустимо!) ждать бесконечно долго, следует установить параметр `TimeOut` в значение `INFINITE`.

Рассмотрим маленький пример. Включим в состав нового проекта объект типа `TThread`, наполнив его метод `Execute` следующим содержимым:

```
Var res: TWaitResult;

procedure TSimpleThread.Execute;
begin
e := TEvent.Create(nil,True,false, 'test');
  repeat
  e.ReSetEvent;
  res := e.WaitFor(10000);
  Synchronize(ShowInfo);
  until Terminated;
e.Free;
end;

procedure TSimpleThread.ShowInfo;
begin
  ShowMessage(IntToStr(Integer(res)));
end;
```

На главной форме разместим две кнопки — нажатие одной из них запускает поток, нажатие второй активизирует событие:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 TSimpleThread.Create(False);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 e.SetEvent;
end;
```

Нажмем первую кнопку. Тогда появившийся на экране результат (метод `ShowInfo`) будет зависеть от того, была ли нажата вторая кнопка или истекли отведенные 10 секунд.

События используются не только для работы с потоками — некоторые процедуры операционной системы автоматически переключают их. К числу

таких процедур относятся отложенный (overlapped) ввод/вывод и события, связанные с коммуникационными портами.

Взаимные исключения

Объект типа *взаимное исключение* (mutex) позволяет только одному потоку в данное время владеть им. Если продолжать аналогии, то этот объект можно сравнить с эстафетной палочкой.

Класс, инкапсулирующий взаимное исключение, — `TMutex` — находится в модуле IPCTHRD.PAS (пример IPCDEMOS). Конструктор:

```
constructor Create(const Name: string);
```

задает имя создаваемого объекта. Первоначально он не принадлежит никому. (Но функция API `CreateMutex`, вызываемая в нем, позволяет передать созданный объект тому потоку, в котором это произошло.) Далее метод

```
function Get(TimeOut: Integer): Boolean;
```

производит попытку в течение `TimeOut` миллисекунд завладеть объектом (в этом случае результат равен `True`). Если объект более не нужен, следует вызвать метод

```
function Release: Boolean;
```

Программист может использовать взаимное исключение, чтобы избежать считывания и записи общей памяти несколькими потоками одновременно.

Семафор

Семафор (semaphore) подобен взаимному исключению. Разница между ними в том, что семафор может управлять количеством потоков, которые имеют к нему доступ. Семафор устанавливается на предельное число потоков, которым доступ разрешен. Когда это число достигнуто, последующие потоки будут приостановлены, пока один или более потоков не отсоединятся от семафора и не освободят доступ.

В качестве примера использования семафора рассмотрим случай, когда каждый из группы потоков работает с фрагментом совместно используемого пула памяти. Так как совместно используемая память допускает обращение к ней только определенного числа потоков, все прочие должны быть блокированы вплоть до момента, когда один или несколько пользователей пула откажутся от его совместного использования.

Критическая секция

Работая в Delphi, программист может также использовать объект типа *критическая секция* (critical section). Критические секции подобны взаимным

исключениям по сути, однако между ними существуют два главных отличия:

- взаимные исключения могут быть совместно использованы потоками в различных процессах, а критические секции — нет;
- если критическая секция принадлежит другому потоку, ожидающий поток блокируется вплоть до освобождения критической секции. В отличие от этого, взаимное исключение разрешает продолжение по истечении тайм-аута.

Критические секции и взаимные исключения очень схожи. На первый взгляд, выигрыш от использования критической секции вместо взаимного исключения не очевиден. Критические секции, однако, более эффективны, чем взаимные исключения, т. к. используют меньше системных ресурсов. Взаимные исключения могут быть установлены на определенный интервал времени, по истечении которого выполнение продолжается; критическая секция всегда ждет столько, сколько потребуется.

Возьмем класс `TCriticalSection` (модуль SYNCOBJS.PAS). Логика использования его проста — "держать и не пущать". В многопотоковом приложении создается и инициализируется общая для всех потоков критическая секция. Когда один из потоков достигает критически важного участка кода, он пытается захватить секцию вызовом метода `Enter`:

```
MySection.Enter;
try
 DoSomethingCritical;
finally
 MySection.Leave;
end;
```

Когда другие потоки доходят до оператора захвата секции `Enter` и обнаруживают, что она уже захвачена, они приостанавливаются вплоть до освобождения секции первым потоком путем вызова метода `Leave`. Обратите внимание, что вызов `Leave` помещен в конструкцию `try..finally` — здесь требуется стопроцентная надежность. Критические секции являются системными объектами и подлежат обязательному освобождению — впрочем, как и остальные рассматриваемые здесь объекты.

Процесс. Порождение дочернего процесса

Объект типа *процесс* (process) может быть использован для того, чтобы приостановить выполнение потока в том случае, если он для своего продолжения нуждается в завершении процесса. С практической точки зрения такая проблема встает, когда нужно в рамках вашего приложения исполнить приложение, созданное кем-то другим, или, к примеру, сеанс MS-DOS.

Рассмотрим, как, собственно, один процесс может породить другой. Вместо устаревшей и поддерживаемой только для совместимости функции `WinExec`, перекочевавшей из прежних версий Windows, гораздо правильнее использовать более мощную:

```
function CreateProcess(lpApplicationName: PChar; lpCommandLine: PChar;
  lpProcessAttributes, lpThreadAttributes: PSecurityAttributes;
  bInheritHandles: BOOL; dwCreationFlags: DWORD; lpEnvironment: Pointer;
  lpCurrentDirectory: PChar; const lpStartupInfo: TStartupInfo;
  var lpProcessInformation: TProcessInformation): BOOL;
```

Первые два параметра ясны — это имя запускаемого приложения и передаваемые ему в командной строке параметры. Параметр `dwCreationFlags` содержит флаги, определяющие способ создания нового процесса и его будущий приоритет. Использованные в приведенном ниже листинге флаги означают: `CREATE_NEW_CONSOLE` — будет запущено новое консольное приложение с отдельным окном; `NORMAL_PRIORITY_CLASS` — нормальный приоритет.

Структура `TStartupInfo` содержит сведения о размере, цвете, положении окна создаваемого приложения. В нижеследующем примере (листинг 29.1) используется поле `wShowWindow`: установлен флаг `SW_SHOWNORMAL`, означающий визуализацию окна с нормальным размером.

На выходе функции заполняется структура `lpProcessInformation`. В ней программисту возвращаются дескрипторы и идентификаторы созданного процесса и его первичного потока. Нам понадобится дескриптор процесса — в нашем примере создается консольное приложение, затем происходит ожидание его завершения. "Просигналит" нам об этом именно объект `lpProcessInformation.hProcess`.

Листинг 29.1. Порождение дочернего процесса

```
var
      lpStartupInfo: TStartupInfo;
      lpProcessInformation: TProcessInformation;
begin
 FillChar(lpStartupInfo,Sizeof(lpStartupInfo),#0);
 lpStartupInfo.cb := Sizeof(lpStartupInfo);
 lpStartupInfo.dwFlags := STARTF_USESHOWWINDOW;
 lpStartupInfo.wShowWindow := SW_SHOWNORMAL;
  if not CreateProcess(nil,
                PChar('ping localhost'),
                nil,
                nil,
                false,
```

```
                CREATE_NEW_CONSOLE or NORMAL_PRIORITY_CLASS,
                nil,
                nil,
                lpStartupInfo,
                lpProcessInformation)
  then
   ShowMessage(SysErrorMessage(GetLastError))
  else
   begin
    WaitForSingleObject(lpProcessInformation.hProcess, 10000);
    CloseHandle(lpProcessInformation.hProcess);
   end;
end;
```

Поток

Поток может ожидать другой поток точно так же, как и другой процесс. Ожидание можно организовать с помощью функций API (как в только что рассмотренном примере), но удобнее это сделать при помощи метода `TThread.WaitFor`.

Консольный ввод

Консольный ввод (console input) годится для потоков, которые должны ожидать отклика на нажатие пользователем клавиши на клавиатуре. Этот тип ожидания может быть использован в программе дуплексной связи (`Chat`). Один поток при этом будет ожидать получения символов; второй — отслеживать ввод пользователя и затем отсылать набранный текст ожидающему приложению.

Оповещение об изменении в файловой системе

Этот вид объекта ожидания очень интересен и незаслуженно мало известен. Мы рассмотрели практически все варианты того, как один поток может подать сигнал другому. А как получить сигнал от операционной системы? Ну, например, о том, что в файловой системе произошли какие-то изменения? Такой вид оповещения позаимствован из ОС UNIX и доступен программистам, работающим с Win32.

Для организации мониторинга файловой системы нужно использовать три функции — `FindFirstChangeNotification`, `FindNextChangeNotification` и `FindCloseChangeNotification`. Первая из них возвращает дескриптор объекта файлового оповещения, который можно передать в функцию ожидания. Объект активизируется тогда, когда в заданной папке произошли те или

иные изменения (создание или уничтожение файла или папки, изменение прав доступа и т. д.). Вторая — готовит объект к реакции на следующее изменение. Наконец, с помощью третьей функции следует закрыть ставший ненужным объект.

Так может выглядеть код метода `Execute` потока, созданного для мониторинга файловой системы:

```
var DirName : string;
...
procedure TSimpleThread.Execute;
var r: Cardinal;
  fn : THandle;
begin
 fn := FindFirstChangeNotification(pChar(DirName), True,
 FILE_NOTIFY_CHANGE_FILE_NAME);
 repeat
 r := WaitForSingleObject(fn,2000);
 if r = WAIT_OBJECT_0 then
  Synchronize(Form1.UpdateList);
 if not FindNextChangeNotification(fn) then break;
 until Terminated;
 FindCloseChangeNotification(fn);
end;
```

На главной форме должны находиться компоненты, нужные для выбора обследуемой папки, а также компонент `TListBox`, в который будут записываться имена файлов:

```
procedure TForm1.Button1Click(Sender: TObject);
var dir : string;
begin
 if SelectDirectory(dir,[],0) then
  begin
   Edit1.Text := dir;
   DirName := dir;
  end;
end;
procedure TForm1.UpdateList;
var SearchRec: TSearchRec;
begin
  ListBox1.Clear;
  FindFirst(Edit1.Text+'\*.*', faAnyFile, SearchRec);
  repeat
   ListBox1.Items.Add(SearchRec.Name);
```

```
  until FindNext(SearchRec) <> 0;
  FindClose(SearchRec);
end;
```

Приложение готово. Чтобы оно стало полнофункциональным, предусмотрите в нем механизм перезапуска потока при изменении обследуемой папки.

Локальные данные потока

Интересная проблема возникает, если в приложении будет несколько одинаковых потоков. Как избежать совместного использования одних и тех же переменных несколькими потоками? Первое, что приходит на ум, — добавить и использовать поля объекта — потомка `TThread`, которые можно добавить при его создании. Каждый поток соответствует отдельному экземпляру объекта, и их данные пересекаться не будут. (Кстати, это одно из больших удобств использования класса `TThread`.) Но есть функции API, которые знать не знают об объектах Delphi и их полях и свойствах. Для поддержки разделения данных между потоками на нижнем уровне в язык Object Pascal введена специальная директива — `threadvar`, которая отличается от директивы описания переменных `var` тем, что применяется только к локальным данным потока. Следующее описание:

```
Var
 data1: Integer;
threadvar
 data2: Integer;
```

означает, что переменная `data1` будет использоваться всеми потоками данного приложения, а переменная `data2` будет у каждого потока своя.

Как избежать одновременного запуска двух копий одного приложения

Такая задача возникает очень часто. Многие, особенно начинающие, пользователи не вполне понимают, что между щелчком по значку приложения и его запуском может пройти несколько секунд, а то и десятков секунд. Они начинают щелкать по значку, запуская все новые копии. Между тем, при работе с базами данных и во многих других случаях иметь более одной копии не только не нужно, но и вредно.

Идея заключается в том, чтобы первая создаваемая копия приложения захватывала некий ресурс, а все последующие при запуске пытались сделать то же самое и в случае неудачи завершались.

Пример такого ресурса — общий блок в файле, отображаемом в память. Поскольку этот ресурс имеет имя, можно сделать его уникальным именно для вашего приложения:

```
var UniqueMapping : THandle;
    FirstWindow : THandle;
begin
  UniqueMapping := CreateFileMapping($ffffffff,
  nil, PAGE_READONLY, 0, 32,'MyMap');
  if UniqueMapping = 0 then
  begin
    ShowMessage(SysErrorMessage(GetLastError));
    Halt;
   end
  else if GetLastError = ERROR_ALREADY_EXISTS  then
   begin
    FirstWindow := FindWindowEx(0, 0, TfmMain.ClassName, nil);
    if FirstWindow<>0 then
       SetForegroundWindow(FirstWindow);
    Halt;
   end;
// Нет других копий — продолжение
Application.Initialize;
...
```

Примерно такие строки нужно вставить в начало текста проекта до создания форм. Блок совместно используемой памяти выделяется в системном страничном файле (об этом говорит первый параметр, равный −1, *см. описание функции* `CreateFileMapping`). Его имя — `MyMap`. Если при создании блока будет получен код ошибки `ERROR_ALREADY_EXISTS`, это свидетельствует о наличии работающей копии приложения. В этом случае приложение переключает фокус на главную форму другого экземпляра и завершается; в противном случае процесс инициализации продолжается.

Резюме

Потоки, как и другие мощные инструменты, должны быть использованы с осторожностью и без злоупотреблений, поскольку могут возникнуть ошибки, которые очень трудно найти. Есть очень много доводов за использование потоков, но есть и доводы против этого. Работа с потоками будет проще, если учитывать нижеприведенные положения.

- Если потоки работают только с переменными, объявленными внутри их собственного класса, то ситуации гонок и тупиков крайне маловероятны.

Другими словами, избегайте использования в потоках глобальных переменных и переменных других объектов.

- Если вы обращаетесь к полям или методам объектов VCL, делайте это только посредством метода `Synchronize`.
- Не "пересинхронизируйте" ваше приложение, а не то оно будет работать как один единственный поток. Избыточно синхронизированное приложение теряет все преимущества от наличия нескольких потоков, т. к. они будут постоянно останавливаться и ждать синхронизации.

Потоки предоставляют изящное решение некоторых сегодняшних проблем программирования; но они также усложняют и без того непростой процесс отладки. И все же преимущества потоков однозначно перевешивают их недостатки.

ГЛАВА 30

Многомерное представление данных

Помимо стандартных компонентов отображения данных в VCL Delphi имеются дополнительные компоненты, которые позволяют представлять данные в виде кросстаба. При этом заставить работать кросстаб с двумя и более полями почти так же просто, как и обычный компонент `TDBGrid`. Эти компоненты расположены на странице **Decision Cube** Палитры компонентов.

Кросстабом называется такое табличное представление данных, которое имеет переменную структуру по горизонтали и вертикали. Причем обозначения столбцов по вертикали и строк по горизонтали соответствуют значениям полей набора данных. В ячейках кросстаба содержатся не данные, а суммарные значения для двух полей, которые пересекаются в этой ячейке.

В настоящей главе рассматриваются следующие вопросы:

- для чего необходим кросстаб;
- особенности запросов SQL для многомерного представления;
- компоненты многомерного представления и их взаимосвязь.

Понятие кросстаба

Обычная таблица данных имеет строго заданное число столбцов, причем каждый столбец всегда предназначен для представления данных из одного поля. Для кросстаба число и назначение столбцов зависит от значений какого-либо поля. Число строк в кросстабе не равно числу строк в таблице БД, а также зависит от значений какого-либо поля. В ячейках кросстаба всегда располагается суммирующая информация по значениям полей горизонтали и вертикали (рис. 30.1).

Создать подобную двумерную структуру отображения данных при помощи обычных компонентов со страницы **Data Controls** Палитры компонентов очень непросто и хлопотно.

	1997	1998	1999
Geo Tech Inc.	18470.00	19000.00	—
3D-Pad Corp.	120000.00	3400.80	93773220.00
MPM Corporation	—	7349.50	76300.00

Рис. 30.1. Пример кросстаба

В общем случае горизонтальную и вертикальную структуры кросстаба могут составлять несколько полей одновременно, которые сгруппированы относительно более общих полей.

Для создания наборов данных, которые можно представить в виде кросстаба, используются запросы SQL с применением группирующего оператора `GROUP BY` и агрегатных функций. Если обратиться к топологическим аналогиям, то набор данных такого запроса представляет собой многомерный гиперкуб, каждая сторона которого соответствует одному полю набора данных.

Для дальнейшего изложения необходимо ввести еще одно понятие. Совокупность строк или колонок, имеющих отношение к одному полю набора данных, будем называть *размерностью*. Размерность представляет собой виртуальную плоскость, которая рассекает многомерный куб данных параллельно какой-либо стороне этого куба. Компоненты многомерного представления данных как раз предназначены для того, чтобы визуализировать это n-мерное сечение.

Переходя от пространственных моделей к наборам данных, можно сказать, что размерность представляет собой совокупность значений какого-либо поля в кросстабе относительно других полей.

Взаимосвязь компонентов многомерного представления данных

При создании в приложении формы для многомерного представления данных следует помнить, что при этом обязательно должны решаться следующие задачи:

- должен быть создан группирующий и суммирующий запрос SQL, обеспечивающий открытие набора данных для кросстаба;
- перед отображением данных необходимо настроить параметры размерностей кросстаба;
- непосредственный показ данных в кросстабе;
- работающий кросстаб должен эффективно управляться на уровне размерностей.

Для этого в форме приложения требуется разместить как минимум пять компонентов со страницы **Decision Cube** Палитры компонентов.

Для создания запроса SQL можно использовать компонент `TDecisionQuery` или обычный компонент `TQuery`.

Запрос должен быть связан с компонентом `TDecisionCube`, который осуществляет подготовку набора данных запроса к многомерному показу.

Для соединения многомерного набора данных с компонентом отображения данных используется компонент `TDecisionSource` — полный функциональный аналог `TDataSource`. Этот компонент, в свою очередь, должен связываться и с набором данных, и с инструментом многомерного представления данных.

Непосредственный показ многомерного набора данных проводится при помощи компонентов `TDecisionGrid` и `TDecisionGraph`. Они должны поддерживать соединение с компонентом `TDecisionSource`.

Наконец, управление многомерным представлением данных реализует компонент `TDecisionPivot`, он также должен быть связан с компонентом `TDecisionSource`.

Допустим, что на форме расположены следующие компоненты:

- `TDecisionQuery` по имени `DecisionQuery1`;
- `TDecisionCube` по имени `DecisionCube1`;
- `TDecisionSource` по имени `DecisionSource1`;
- `TDecisionGrid` по имени `DecisionGrid1`;
- `TDecisionPivot` по имени `DecisionPivot1`.

Тогда для того, чтобы связать все эти компоненты в единый работающий механизм многомерного представления данных, нужно установить значения для их важнейших свойств. Значения свойств представлены в табл. 30.1.

Таблица 30.1. *Как связать компоненты многомерного представления данных*

Свойство	Значение	Описание
TBecisionCube		
`DataSet`	`DecisionQuery1`	Определяет компонент доступа к данным, который создает набор данных
TDecisionSource		
`DecisionCube`	`DecisionCube1`	Указывает на компонент формирования многомерного набора данных
TDecisionGrid		
`DecisionSource`	`DecisionSource1`	Ссылается на компонент `TDecisionSource`

Таблица 30.1 *(окончание)*

Свойство	Значение	Описание
TDecisionPivot		
DecisionSource	DecisionSource1	Ссылается на компонент TDecisionSource

Если задать текст запроса SQL и открыть набор данных, то вся цепочка заработает, причем ее поведение ничем не отличается от поведения во время выполнения приложения.

Теперь, когда мы узнали, как объединить компоненты многомерного представления данных в единую систему, настало время более подробно изучить возможности каждого компонента.

Подготовка набора данных

Компоненты многомерного представления данных работают со специально созданным и подготовленным набором данных. Эта работа выполняется специальным компонентом доступа к данным — TDecisionQuery. Его непосредственным предком является компонент TQuery.

Набор данных формируется при помощи запроса, который основан на стандартном синтаксисе SQL 92. Для обеспечения работы многомерного представления данных запрос должен удовлетворять ряду требований.

1. В тексте запроса должны присутствовать только те поля, которые разработчик хочет показать в компонентах многомерного представления данных.
2. Поля запроса должны быть сгруппированы при помощи оператора GROUP BY.
3. Запрос должен содержать агрегатные функции, которые определяют вид информации, отображаемой в ячейках кросстаба.

Компонент TDecisionQuery должен только обеспечить выполнение запроса и создание набора данных, он не имеет никаких дополнительных свойств или методов. Поэтому для создания набора данных можно использовать и обычный компонент TQuery. Преимущество компонента TDecisionQuery состоит в том, что он имеет специализированный редактор для создания текста запроса (рис. 30.2). Он вызывается командой **Decision Query Editor** из всплывающего меню компонента или двойным щелчком на компоненте.

Элементы управления страницы **Dimensions/Summaries** позволяют создавать текст запроса, манипулируя именами полей таблиц. Псевдоним базы данных выбирается в комбинированном списке **Database**. После этого в списке **Table** задается нужная таблица. Если в запросе требуется использовать не-

сколько таблиц, то для их выбора можно воспользоваться утилитой SQL Builder, которая вызывается щелчком на одноименной кнопке.

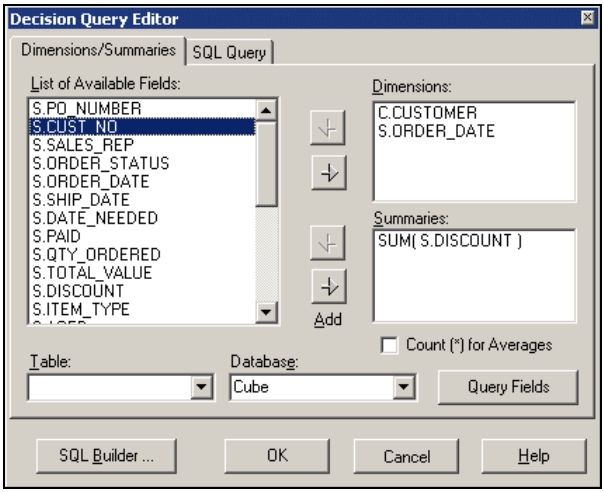


Рис. 30.2. Специализированный редактор компонента `TDecisionQuery`

Из списка доступных полей при помощи кнопок **Add** требуемые поля можно перенести в список полей — размерностей **Dimensions** и список суммирующих полей **Summaries**. Поля из этих списков используются при создании запроса.

Запрос формируется автоматически при работе с описанными элементами управления. Текст запроса доступен для просмотра и редактирования на странице **SQL Query**.

Подготовка набора данных к многомерному представлению осуществляется компонентом `TDecisionCube`. Его основная задача — создание размерностей для многомерной структуры данных на основе каждого поля набора данных. Для каждой размерности в компоненте можно задать ряд параметров, которые определяют ее поведение и внешний вид.

Компонент `TDecisionCube` формирует размерности при открытии набора данных, причем созданное многомерное представление данных полностью работоспособно уже во время разработки. Для этого достаточно присвоить свойству `Active` компонентов `TDecisionQuery` или `TQuery` значение `True`. После этого любой визуальный компонент многомерного представления начинает работать так же, как и во время выполнения.

Компонент `TDecisionCube` также позволяет управлять использованием памяти многомерного представления данных. Дело в том, что при добавлении к многомерному представлению новой размерности объем занимаемой памяти возрастает в арифметической прогрессии. Поэтому возможность огра-

ничения размеров используемой памяти особенно актуальна для больших наборов данных.

Все основные настройки компонента выполняются при помощи специализированного редактора свойства `DecisionMap` (рис. 30.3).

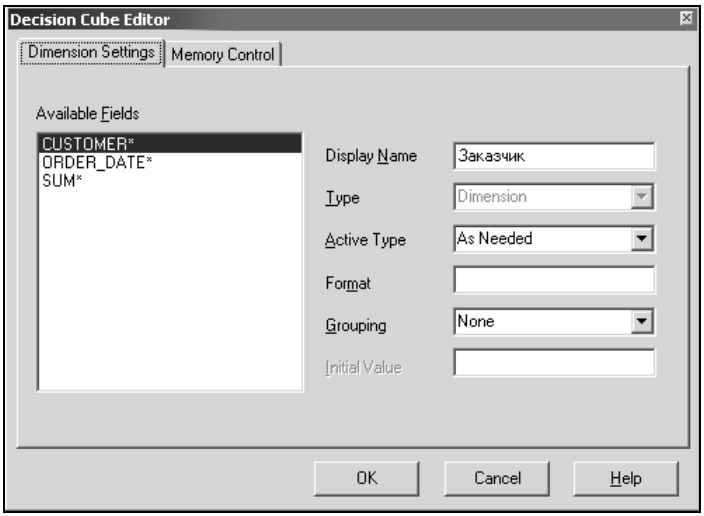


Рис. 30.3. Специализированный редактор свойства `DecisionMap` компонента `TDecisionCube` (страница **Dimension Settings**)

Для настроек размерностей используется страница **Dimension Settings** этого редактора. В расположенном слева списке **Available Fields** содержатся все поля набора данных. В элементах управления справа приведены параметры размерности для выбранного поля.

В однострочном редакторе **Display Name** задается название поля, которое будет присутствовать в визуальных компонентах многомерного просмотра.

Неактивный список выбора **Type** показывает, является ли поле основой для размерности или суммы.

Список выбора **Active Type** определяет, когда данные поля появляются в визуальных компонентах. Его элементы обозначают следующее:

- `Active` — данные поля видны сразу после открытия формы и набора данных во время выполнения или сразу после открытия набора данных во время разработки;
- `As Needed` — данные поля становятся видны после выполнения пользователем во время выполнения или разработчиком во время разработки действий по отображению данных;
- `Inactive` — данные поля не видны.

Однострочный редактор **Format** содержит строку форматирования для данных поля.

Комбинированный список **Grouping** необходим для того, чтобы определить, какие значения будут показаны. Варианты `Year`, `Quarter`, `Month` возможны только для полей с календарным типом данных.

Однострочный редактор **Initial Value** задает начальное значение для поля.

Страница **Memory Control** используется для управления расходом памяти для нужд компонента (рис. 30.4). Однострочные редакторы **Dimensions**, **Summaries** и **Cells** в ряду **Maximum** позволяют задать максимальное число размерностей, сумм и ячеек, соответственно.

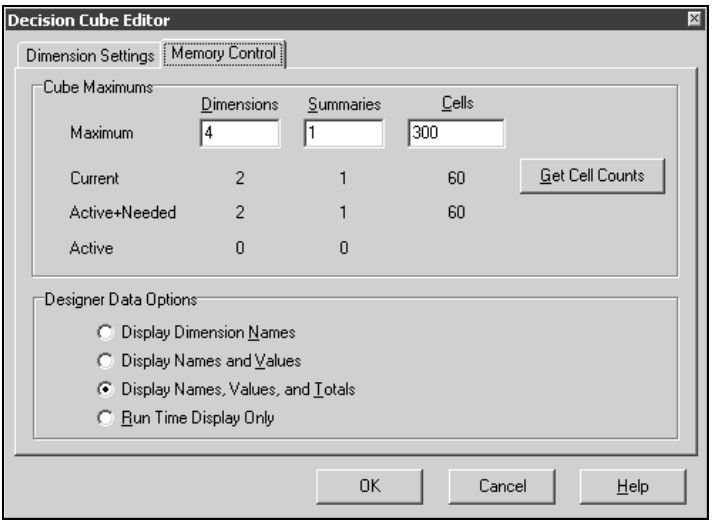


Рис. 30.4. Специализированный редактор свойства `DecisionMap` компонента `TDecisionCube` (страница **Memory Control**)

Аналогичные значения в ряду **Current** показывают текущее число этих структур.

Аналогичные значения в ряду **Active+Needed** показывают общее возможное число размерностей, сумм и ячеек.

Значения в ряду **Active** показывают число видимых размерностей сумм и ячеек.

Кнопка **Get Cell Counts** выполняет запрос, который возвращает число ячеек в кросстабе.

Группа радиокнопок **Designer Data Options** задает режим показа данных во время разработки:

- **Display Dimension Names** — отображаются только названия размерностей;
- **Display Names and Values** — отображаются названия размерностей и значения;
- **Display Names, Values, and Totals** — отображаются названия, значения и суммы размерностей;
- **Run Time Display Only** — визуализация данных осуществляется только во время выполнения.

Подготовленный к использованию для многомерного отображения набор данных необходимо связать с визуальными компонентами. Это делается при помощи компонента `TDecisionSource`. Через один такой компонент с набором данных можно связать несколько визуальных компонентов (см. табл. 30.1).

Кроме этого, компонент `TDecisionSource` позволяет разработчику установить общие для всех связанных с ним визуальных компонентов многомерного представления данных настройки отображения данных.

Компонент *TDecisionQuery*

Компонент доступа к данным `TDecisionQuery` предназначен для создания набора данных, который был бы пригоден для многомерного представления. Для создания набора данных используется запрос SQL. Требования к запросу приведены выше.

Этот компонент является прямым наследником компонента `TQuery` и не имеет собственных свойств и методов. Для создания запросов можно воспользоваться специализированным редактором, который автоматизирует этот процесс.

Компонент *TDecisionCube*

Компонент `TDecisionCube` осуществляет преобразование набора данных, который содержится в компоненте `TDecisionQuery`, к виду, доступному для отображения визуальными компонентами многомерного представления данных (табл. 30.2). Обычную таблицу набора данных компонент преобразует в многомерный кросстаб. Число размерностей создаваемого кросстаба зависит от числа полей данных набора данных. Значения в ячейках кросстаба зависят от типа агрегатной функции в запросе SQL.

Таблица 30.2. *Свойства и методы компонента* `TDecisionCube`

Объявление	Тип	Описание
Свойства		
`property Active: Boolean;`	Pu	Разрешает или запрещает преобразование набора данных в кросстаб

Таблица 30.2 *(продолжение)*

Объявление	Тип	Описание
`property BinData: Boolean;`	Ro	Значение `True` означает, что хотя бы одна размерность находится в свернутом состоянии (данные не отображаются)
`property Capacity: Integer;`	Pu	Определяет число байтов, используемых для хранения многомерного массива
`property CurrentSummary: Integer;`	Pu	Содержит индекс текущей суммы кросстаба
`property DataSet: TDataSet;`	Pb	Ссылка на экземпляр набора данных, который отображается в кросстабе
`type TCubeDesignState = (dsNoData, dsMetaData, dsDimensionData, dsAllData);` `property DesignState: TCubeDesignState;`	Pu	Задает режим отображения данных в кросстабе: • `dsNoData` — во время разработки данные не видны; • `dsMetaData` — видны названия размерностей; • `dsDimensionData` — видны названия размерностей и значения, суммы не видны; • `dsAllData` — видны все данные
`property DimensionCount: Integer;`	Ro	Возвращает число размерностей
`property DimensionMap: TCubeDims;`	Pb	Индексированный список ссылок на объекты параметров размерностей
`property DimensionMapCount: Integer;`	Ro	Общее число полей набора данных, включая поля размерностей и сумм
`property MaxCells: Integer;`	Pb	Задает максимальное число ячеек кросстаба
`property MaxDimensions: Integer;`	Pb	Задает максимальное число размерностей
`property MaxSummaries: Integer;`	Pb	Задает максимальное число сумм
`property ShowProgressDialog: Boolean;`	Pb	При значении `True` при подготовке кросстаба отображается индикатор
`property SummaryCount: Integer;`	Ro	Возвращает число активных сумм кросстаба
Методы		
`function GetDetailSQL(ValueArray: TSmallIntArray; SelectList: string; bActive: Boolean): string;`	Pu	Возвращает текст запроса SQL, который может быть использован для создания набора данных, включающего данные из кросстаба без сумм

Таблица 30.2 *(окончание)*

Объявление	Тип	Описание
`function GetSQL(ValueArray: TSmallIntArray; bActive: Boolean): string;`	Pu	Возвращает текст запроса SQL, который может быть использован для создания набора данных, включающего данные из кросстаба без сумм
`procedure ShowCubeDialog;`	Pu	Вызывает специализированный редактор компонента
`procedure Refresh(DimensionMap : TCubeDims; bForce: Boolean);`		Обновляет список объектов параметров размерностей
Методы-обработчики событий		
`type TCubeRefreshEvent = procedure(DataCube: TCustomDataStore; DimMap: TCubeDims) of object;` `property OnRefresh: TCubeRefreshEvent; property AfterClose: TCubeNotifyEvent;`	Pb	Вызывается сразу после закрытия компонента (`Active := False`)
`property AfterOpen: TCubeNotifyEvent;`	Pb	Вызывается сразу после открытия компонента (`Active := False`)
`property BeforeClose: TCubeNotifyEvent;`	Pb	Вызывается перед закрытием компонента (`Active := False`)
`property BeforeOpen: TCubeNotifyEvent;`	Pb	Вызывается перед открытием компонента (`Active := False`)
`TErrorAction = (eaFail, eaContinue);` `TCapacityErrorEvent = procedure(var EAction: TErrorAction) of object;` `property OnLowCapacity: TCapacityErrorEvent;`	Pb	Вызывается после того, как занимаемый кросстабом объем памяти превысит заданный предел

При помощи методов `GetDetailSQL` и `GetSQL` можно получить тексты запросов, которые возвращают набор данных, соответствующий кросстабу с заданным параметрами состояниями. Массив `ValueArray` содержит условия для полей размерностей. Первой размерности соответствует первый элемент массива, второй размерности — второй элемент и т. д. Если значение элемента меньше нуля, то в результат запроса попадают все значения поля размерности. Значение элемента, равное или больше нуля, определяет индекс значения поля размерности. Параметр `SelectList` содержит разделенный запятыми список дополнительных полей, которые нужно включить в за-

прос. Параметр `bActive` накладывает дополнительное ограничение на размерности. При значении `True` в результат запроса автоматически (без использования параметра `SelectList`) попадают только активные размерности.

Ключевым свойством компонента является свойство `DecisionMap`, которое позволяет установить параметры размерностей и максимальный размер используемой памяти. Для этих целей применяется специализированный редактор (см. рис. 30.4).

Это свойство представляет собой экземпляр класса `TCubeDims`, который инкапсулирует индексированный список экземпляров объектов `TCubeDim`, каждый из которых содержит информацию о параметрах одной размерности. Основные свойства этого класса представлены в табл. 30.3.

Таблица 30.3. *Основные свойства класса* `TCubeDim`

Объявление	Тип	Описание
`type TActiveFlags = (diActive, diAsNeeded, diInactive);` `property ActiveFlag: TActiveFlags;`	Pb	Определяет режим отображения данных размерности
`property BaseName: string;`	Pb	Содержит имя поля размерности в таблице базы данных
`property BinFormat: string;`	Pu	Определяет способ форматирования диапазона значений размерности
`type TBinType = (binNone, binYear, binQuarter, binMonth, binSet, binCustom);` `property BinType: TBinType;`	Pb	Определяет способ группирования данных в размерности
`type TDimFlags = (dimDimension, dimSum, dimCount, dimAverage, dimMin, dimMax, dimGenericAgg, dimUnknown);` `property DimensionType: TDimFlags;`	Pb	Определяет тип размерности
`property FieldName: String;`	Pb	Содержит имя поля в наборе данных
`property FieldType: TFieldType;`	Pu	Определяет тип поля
`property Format: String;`	Pu	Задает форматирование данных размерности
`property Loaded: Boolean;`	Ro	Значение `True` говорит о том, что данный элемент загружен в многомерный набор данных

Таблица 30.3 *(окончание)*

Объявление	Тип	Описание
`property StartDate: TDate;`	Pu	Определяет начальный элемент для группировки по дате
`property StartValue: String;`	Pu	Определяет начальный элемент для группировки по значению
`property ValueCount: Integer;`	Pb	Возвращает число уникальных элементов в размерности

Компонент *TDecisionSource*

Компонент `TDecisionSource` предназначен для связывания визуальных компонентов многомерного представления с компонентом `TDecisionCube` (табл. 30.4). Кроме того, за счет возможности подключения к этому компоненту нескольких визуальных компонентов одновременно, при изменении состояния одного визуального компонента осуществляется синхронизация многомерного представления во всех остальных компонентах.

От набора данных в визуальные компоненты передаются данные, обратно транслируются команды пользователя по управлению многомерным представлением данных.

Таблица 30.4. *Основные свойства компонента* `TDecisionSource`

Объявление	Описание
`type TDecisionControlType = (xtCheck, xtRadio, xtRadioEx);` `property ControlType: TDecisionControlType;`	Определяет способ управления отдельной размерностью в компоненте `TDecisionGrid`
`property CurrentSum: Integer;`	Содержит индекс текущей суммы в компоненте `TDecisionGrid`
`property DecisionCube: TDecisionCube;`	Связывает данный компонент с компонентом `TDecisionCube`. Содержит ссылку на экземпляр компонента `TDecisionCube`
`property Ready: Boolean;`	Значение `True` означает, что данный компонент связан с активным компонентом `TDecisionCube`
`property SparseCols: Boolean;`	При значении `True` из компонента `TDecisionGrid` удаляются пустые колонки

Таблица 30.4 *(окончание)*

Объявление	Описание
`property SparseRows: Boolean;`	При значении `True` из компонента `TDecisionGrid` удаляются пустые строки

Компонент имеет средства для управления состоянием многомерного представления в визуальных компонентах. Для этого используется свойство `ControlType`. Рассмотрим его возможные значения:

- `xtCheck` — щелчок на кнопках размерности в компонентах `TDecisionGrid` и `TDecisionPivot` приводит к открытию или закрытию размерности;
- `xtRadio` — щелчок на кнопках размерности в компонентах `TDecisionGrid` и `TDecisionPivot` приводит к открытию или закрытию данной размерности и закрытию всех остальных в этом направлении;
- `xtRadioEx` — щелчок на кнопках размерности в компонентах `TDecisionGrid` и `TDecisionPivot` приводит к открытию или закрытию данной размерности и закрытию или открытию всех остальных размерностей в этом направлении.

Отображение данных

При работе с кросстабом пользователь имеет дело с двумя визуальными компонентами многомерного представления данных. Это компонент `TDecisionGrid`, который представляет данные в табличном виде, и компонент `TDecisionGraph`, который представляет данные в виде графика.

С компонентом `TDecisionCube` они связаны при помощи компонента `TDecisionSource` *(см. выше)*.

Данные в визуальных компонентах появляются после открытия набора данных в соответствующем компоненте `TDecisionQuery`. Причем эти компоненты полностью работоспособны уже во время разработки приложения.

Компонент *TDecisionGrid*

Компонент `TDecisionGrid` предназначен для многомерного представления данных в табличном виде. Параметры отображаемого набора данных (какие размерности будут видны при открытии, как группировать данные, как управлять размерностями) настраиваются при помощи других компонентов многомерного представления. В компоненте `TDecisionGrid` можно настроить только свойства самой сетки.

Для управления графиком во время выполнения можно использовать компонент `TDecisionPivot`.

В табл. 30.5 приведены основные свойства и методы компонента.

Таблица 30.5. *Свойства и методы компонента* `TDecisionGrid`

Объявление	Тип	Описание
Свойства		
`property Cells[ACol, ARow: Integer]: string;`	Ro	Индексированный массив значений всех ячеек компонента в строковом формате
`property ColCount: Integer;`	Ro	Возвращает общее число колонок в сетке
`property DecisionSource: TDecisionSource;`	Pb	Указывает на компонент `TDecisionSource`, через который осуществляется связь с набором данных
`property Dimensions: TDisplayDims;`	Pb	Объект `TDisplayDims` представляет индексированный список объектов визуальных свойств размерностей
`property FixedCols: Integer;`	Ro	Возвращает число фиксированных колонок, которые используются для отображения информации о размерностях (названия, значения, обозначения)
`property FixedRows: Integer;`	Ro	Возвращает число фиксированных строк, которые используются для отображения информации о размерностях (названия, значения, обозначения)

`type`
`TDecisionGridOption = (cgGridLines, cgOutliner, cgPivotable);`
`TDecisionGridOptions = set of TDecisionGridOption;`
`property Options: TDecisionGridOptions;`
	Pb	Определяет общие настройки компонента:
		• `cgGridLines` — отображаются вертикальные и горизонтальные разделительные линии;
		• `cgOutliner` — отображаются элементы управления в виде знаков "+" и "–" для открытия и закрытия размерностей;
		• `cgPivotable` — размерности можно переупорядочивать при помощи перетаскивания

`property RowCount: Integer;`	Ro	Возвращает общее число строк в сетке

Таблица 30.5 *(окончание)*

Объявление	Тип	Описание
`property ShowCubeEditor: Boolean;`	Pb	Разрешает или запрещает использование специализированного редактора компонента `TDecisionCube`
`property Totals: Boolean;`	Pu	При значении `True` сетка имеет промежуточные суммы по каждой колонке и строке
Методы		

`type`
`TDecisionDrawStates = (dsGroupStart, dsRowCaption, dsColCaption, dsSum, dsRowValue, dsColValue, dsData, dsOpenAfter, dsCloseAfter, dsCloseBefore, dsOpenBefore, dsRowIndicator, dsColIndicator, dsRowPlus, dsColPlus, dsNone);`
`TDecisionDrawState = set of TDecisionDrawStates;`
`function CellDrawState(ACol, ARow: Integer; var Value: string; var DrawState: TDecisionDrawState): boolean;`
	Pu	Позволяет определить назначение любой ячейки сетки. Параметры `ACol` и `ARow` определяют положение ячейки в сетке.
		В параметре `Value` возвращается строка, содержащая значение в том виде, как оно представлено в ячейке.
		Параметр `DrawState` возвращает информацию о назначении ячейки

`function CellValueArray(ACol, ARow: Integer; var ValueArray: TValueArray): boolean;`	Pu	Возвращает индексы всех полей, данные которых суммированы в ячейке. Параметр `ValueArray` содержит индексы полей

Компонент `TDecisionGrid` является предком класса `TCustomGrid` и поэтому обладает всеми базовыми свойствами и методами, присущими сетке.

Для доступа к значению каждой ячейки используется свойство `Cells`. Адресация ячеек осуществляется с левой верхней ячейки, которая имеет индексы [0,0].

Свойство `Dimensions` является экземпляром объекта `TDisplayDims`, который инкапсулирует индексированный список указателей на экземпляры объектов `TDisplayDim`. Каждый такой объект содержит важнейшие визуальные свойства размерностей. При щелчке на кнопке в однострочном редакторе свойства в Инспекторе объектов разворачивается список всех таких объектов.

Компонент *TDecisionGraph*

Компонент TDecisionGraph создает график на основе многомерного представления набора данных. Конкретный вид графика (назначение горизонтальной и вертикальной осей) зависит от настроек компонентов TDecisionCube и TDecisionPivot. По умолчанию к оси абсцисс привязывается первая вертикальная размерность, к оси ординат — первая сумма. Первая горизонтальная размерность отображается в легенде графика.

Одним из предков компонента TDecisionGraph является класс TChart, от которого унаследованы все многочисленные свойства и методы для настройки графика.

Для подключения к графику набора данных используется свойство

```
property DecisionSource: TDecisionSource;
```

которое ссылается на экземпляр компонента источника данных.

Сразу после подключения автоматически строится график с осями, заданными по умолчанию.

Для управления графиком во время выполнения можно использовать компонент TDecisionPivot.

Управление данными

Несомненное преимущество многомерного представления данных в том, что пользователь может легко изменить взаимное положение размерностей одной стороны между собой и переносить размерности с горизонтали на вертикаль и обратно. Для того чтобы сделать размерность видимой или невидимой, пользователю достаточно щелкнуть на кнопке.

Взаимное положение и расположение размерностей по сторонам кросстаба никак не связано с местом полей в запросе компонента TDecisionQuery.

Все операции по управлению многомерным представлением сосредоточены в одном компоненте — TDecisionPivot (см. рис. 30.2). В некоторой степени это аналог компонента TDBNavigator, только TDecisionPivot управляет не записями набора данных, а размерностями многомерного представления данных.

Этот компонент подключается к общей цепочке компонентов многомерного представления данных через компонент TDecisionSource. Поэтому любые действия с компонентом TDecisionPivot немедленно отражаются во всех визуальных компонентах, которые также подключены к этому экземпляру TDecisionSource.

Компонент *TDecisionPivot*

Компонент `TDecisionPivot` предоставляет пользователю средства управления размерностями многомерного представления данных. В стандартном состоянии компонент представляет собой панель, разделенную на три части (табл. 30.6). Каждая часть имеет собственный набор кнопок.

Левая часть имеет единственную кнопку, щелчок на которой позволяет сделать выбор суммирующего поля из развернувшегося списка. Элементы списка соответствуют вычисляемым полям с использованием агрегатных функций из запроса соответствующего компонента `TDecisionQuery`.

Выбор поля приводит к изменению значений в ячейках кросстаба. Например, выбор поля с функцией `SUM` изменит значения в ячейках на суммы полей, поле с функцией `COUNT` произведет подсчет количества элементов в полях.

Средняя часть панели предназначена для размещения кнопок, соответствующих горизонтальным размерностям.

Правая часть панели используется для кнопок, соответствующих вертикальным размерностям.

Кнопка в нажатом состоянии показывает размерность. Одновременно отображается и общая сумма по размерности.

Размерности можно менять местами и перемещать с вертикали на горизонталь и обратно. Для этого можно выбрать команду **Moved to Column Area** из всплывающего меню кнопки. Во время выполнения можно использовать обычное перетаскивание кнопок при помощи мыши.

Всплывающее меню кнопки размерности имеет еще одну команду — **Drilled In**. По этой команде размерность переходит в режим детального просмотра по каждому значению поля. Конкретное значение можно выбрать из списка, который открывается при щелчке на кнопке. В этом случае суммы в ячейках кросстаба рассчитываются не по всей размерности, а только по выбранному значению.

Таблица 30.6. *Свойства и методы компонента* `TDecisionPivot`

Объявление	Тип	Описание
Свойства		
`property DecisionSource: TDecisionSource;`	Pb	Определяет компонент `TDecisionSource`, через который осуществляется управление многомерным представлением данных

Таблица 30.6 *(окончание)*

Объявление	Тип	Описание
`type TDecisionButtonPosition = (xtHorizontal, xtVertical, xtLeftTop);` `property GroupLayout: TDecisionButtonPosition;`	Pb	Задает способ расположения кнопок на панели: • `xtHorizontal` — в ряд слева направо; • `xtVertical` — в колонку сверху вниз; • `xtLeftTop` — кнопки вертикальных размерностей вдоль верхнего края, кнопки горизонтальных размерностей вдоль левого края, кнопка сумм в левом верхнем углу
`type TDecisionPivotOption = (xtRows, xtColumns, xtSummaries);` `TDecisionPivotOptions = set of TDecisionPivotOption;` `property Groups: TDecisionPivotOptions;`	Pb	Управляет видимостью трех групп кнопок
`property GroupSpacing: Integer;`	Pb	Определяет размер в пикселах промежутка между группами кнопок
Методы		
`procedure SetBounds(Left, Top, Height, Width: Integer); override;`	Pu	Переустанавливает размеры компонента в соответствии с параметрами метода

Методы-обработчики компонента унаследованы от класса `TWinControl`.

Пример многомерного представления данных

В качестве примера использования многомерного представления данных рассмотрим демонстрационное приложение `DemoMDCube` (рис. 30.5).

В качестве исходного набора данных используется запрос к таблицам SALES и CUSTOMER общедоступной базы данных EMPLOYEE.GDB в составе поставки InterBase следующего вида:

```
SELECT C.CUSTOMER, S.ORDER_DATE, SUM(S.DISCOUNT)
FROM SALES S
   INNER JOIN CUSTOMER C
   ON  (C.CUST_NO = S.CUST_NO)
GROUP BY C.CUSTOMER, S.ORDER_DATE
```

Запрос удовлетворяет всем требованиям для обеспечения многомерного представления данных. Для выполнения запроса и создания набора данных в проекте существует компонент `DecisionQuery1`.

Набор данных компонента открыт во время разработки, поэтому все компоненты многомерного представления в проекте ведут себя так же, как и во время выполнения программы.

С ним связан компонент `DecisionCube1`, для которого свойство имеет следующее значение:

```
DecisionCube1.DataSet := DecisionQuery1;
```

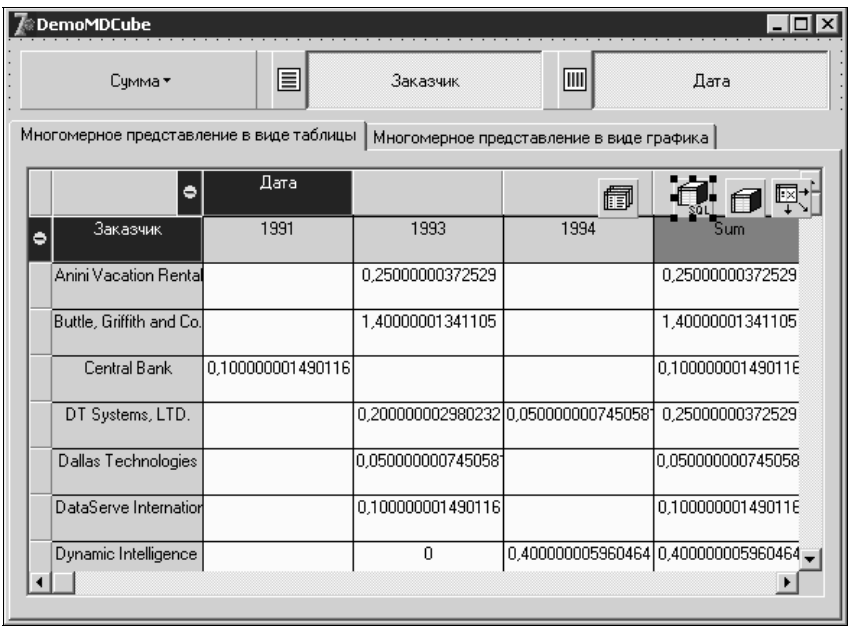


Рис. 30.5. Главная форма проекта DemoMDCube

Компонент `DecisionCube1` выполняет всю работу по созданию многомерного представления набора данных компонента `DecisionQuery1`. Причем, практически все делается без вмешательства пользователя.

При настройке компонента был использован специализированный редактор свойства `DecisionMap`. В нем были заданы названия для размерностей и значения для расчета максимального размера используемой памяти. Все заданные значения соответствуют рекомендованным.

Для связывания набора данных с визуальными компонентами использован компонент `DecisionSource1`. На него замыкаются все три визуальных компонента многомерного представления, использованные в проекте.

Управление осуществляется компонентом `DecisionPivot1`. Основной визуальный компонент `DecisionGrid1` представляет многомерный набор данных в табличном виде. В исходном состоянии по горизонтали расположены размерности номеров накладных и наименований, по вертикали размещаются размерности дат заказов и покупателей.

Дополнительного программного кода проект не имеет.

Резюме

Многомерное представление данных позволяет проводить сложный анализ информации, содержащейся в базах данных. Основой многомерного представления является группирующий запрос (с оператором `GROUP BY`). С точки зрения пользователя анализ с помощью многомерного представления данных очень прост в использовании.

ГЛАВА 31

Использование возможностей Shell API

Разобравшись с механизмами COM, вам наверняка захочется "испытать радость общения" с объектами, имеющимися в составе ОС. Microsoft уверенно идет к тому, чтобы все составные части своих операционных систем, как и прочих продуктов, превратить в COM-объекты. В этом направлении сделаны большие шаги, и оболочка Windows, и ее файловая система предоставляют интерфейсы COM. В Windows 2000, судя по заверениям представителей фирмы, все новые возможности представлены и доступны в виде интерфейсов.

В качестве примера работы с интерфейсом ShellLink вместе с Delphi поставляется приложение Virtual ListView. Но, во-первых, в нем безо всякого документирования вводятся достаточно сложные структуры и интерфейсы; во-вторых, оно содержит только минимум функций для работы с объектами. В этой главе мы постараемся объяснить применяемые там приемы.

Примечание

Интерфейсы функций и COM-объектов `Shell` содержатся в модулях SHELLAPI.PAS и SHLOBJ.PAS, которые имеются в поставке Delphi.

Понятие пространства имен

Необходимость как-то упорядочить все те сущности, с которыми имеет дело современная ОС, всегда вставала перед разработчиками. Довольно успешный подход к этому реализован в платформе Windows. Вооружившись идеями объектного подхода, в Microsoft разбили интерфейс ОС на две части: средства поддержки пространства имен и средства его просмотра.

Под *пространством имен оболочки* (Shell Namespace) мы будем понимать иерархически упорядоченную совокупность имен всех объектов, которые

могут быть просмотрены через средства просмотра — файлы, устройства памяти, принтеры, сетевые ресурсы. В этой совокупности могут встречаться как реально существующие объекты (папки файловой системы), так и виртуальные объекты (папки Принтеры, Мой компьютер и т. п.). Типовым средством просмотра пространства имен является Explorer (Проводник), но можно заменить его на другое средство, в том числе собственноручно разработанное. Обе составные части являются совокупностями COM-объектов, они обладают полиморфизмом и легко расширяемы. Об использовании этих объектов и функций API оболочки ОС и пойдет речь в данной главе.

Размещение значка приложения на *System Tray*

Часто программисту приходится сталкиваться с задачей написания приложения, работающего в фоновом режиме и не нуждающегося в месте на Панели задач. Если вы посмотрите на правый нижний угол рабочего стола Windows, то наверняка найдете там приложения, для которых эта проблема решена: часы, переключатель раскладок клавиатуры, регулятор громкости и т. п. Ясно, что, как бы вы не увеличивали и не уменьшали формы своего приложения, попасть туда обычным путем не удастся. Способ для этого предоставляет Shell API.

Те картинки, которые находятся на **System Tray** — это действительно просто картинки, а не свернутые окна. Они управляются и располагаются панелью **System Tray**. Она же берет на себя еще две функции: показ подсказки для каждого из значков и оповещение приложения, создавшего значок, обо всех перемещениях мыши над ним.

Весь API **System Tray** состоит из 1 (одной) функции:

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData: PNotifyIconData): BOOL;
PNotifyIconData = TNotifyIconData;
TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

Параметр `dwMessage` определяет одну из операций: `NIM_ADD` означает добавление значка в область, `NIM_DELETE` — удаление, `NIM_MODIFY` — изменение.

Ход операции зависит от того, какие поля структуры `TNotifyIconData` будут заполнены.

Обязательным для заполнения является поле `cbSize` — там содержится размер структуры. Поле `Wnd` должно содержать дескриптор окна, которое будет оповещаться о событиях, связанных со значком. Идентификатор сообщения Windows, которое вы хотите получать от системы о перемещениях мыши над значком, запишите в поле `uCallbackMessage`. Если вы хотите, чтобы при этих перемещениях над вашим значком показывалась подсказка, то задайте ее текст в поле `szTip`. В поле `uID` задается номер значка — каждое приложение может поместить на **System Tray** сколько угодно значков. Дальнейшие операции вы будете производить, задавая этот номер. Дескриптор помещаемого значка должен быть задан в поле `hIcon`. Здесь вы можете задать значок, связанный с вашим приложением, или загрузить свой — из ресурсов.

Примечание

Изменить главный значок приложения можно в диалоговом окне **Project/Options** на странице **Application**. Он будет доступен через свойство `Application.Icon`. Тут же можно отредактировать и строку для подсказки — свойство `Application.Title`.

Наконец, в поле `uFlags` вы должны сообщить системе, что именно вы от нее хотите, или, другими словами, какие из полей `hIcon`, `uCallbackMessage` и `szTip` вы на самом деле заполнили. В этом поле предусмотрена комбинация трех флагов: `NIF_ICON`, `NIF_MESSAGE` и `NIF_TIP`. Вы можете заполнить, скажем, поле `szTip`, но если вы при этом не установили флаг `NIF_TIP`, созданный вами значок не будет иметь строки с подсказкой.

Два приведенных ниже метода иллюстрируют сказанное. Первый из них создает значок на **System Tray**, а второй — уничтожает его.

```
const
 WM_MYTRAYNOTIFY = WM_USER + 123;

procedure TForm1.CreateTrayIcon(n:Integer);
var nidata : TNotifyIconData;
begin
 with nidata do
  begin
   cbSize := SizeOf(TNotifyIconData);
   Wnd := Self.Handle;
   uID := n;
   uFlags := NIF_ICON or NIF_MESSAGE or NIF_TIP;
   uCallBackMessage := WM_MYTRAYNOTIFY;
   hIcon := Application.Icon.Handle;
   szTip := 'THis is TrayIcon Example';
  end;
```

```
  Shell_NotifyIcon(NIM_ADD, @nidata);
end;

procedure TForm1.DeleteTrayIcon(n:Integer);
var nidata : TNotifyIconData;
begin
 with nidata do
  begin
   cbSize := SizeOf(TNotifyIconData);
   Wnd := Self.Handle;
   uID := n;
  end;
  Shell_NotifyIcon(NIM_DELETE, @nidata);
end;
```

Примечание

Не забывайте уничтожать созданные вами значки на **System Tray**. Это не делается автоматически даже при закрытии приложения. Значок будет удален только после перезагрузки системы.

Внешний вид значка, помещенного нами на **System Tray**, ничем не отличается от значков других приложений (рис. 31.1).

Рис. 31.1. Над значком, помещенным на панель **System Tray**, видна строка подсказки

Сообщение, задаваемое в поле `uCallbackMessage`, по сути дела является единственной ниточкой, связывающей вас со значком после его создания. Оно объединяет в себе несколько сообщений. Когда к вам пришло такое сообщение (в примере, рассмотренном выше, оно имеет идентификатор `WM_MYTRAYNOTIFY`), поля в переданной в обработчик структуре типа `TMessage` распределены так. Параметр `wParam` содержит номер значка (тот самый, что задавался в поле `uID` при его создании), а параметр `LParam` — идентификатор сообщения от мыши, вроде `WM_MOUSEMOVE`, `WM_LBUTTONDOWN` и т. п. К сожалению, остальная информация из этих сообщений теряется. Координаты мыши в момент события придется узнать, вызвав функцию API `GetCursorPos`:

```
procedure TForm1.WMICON(var msg: TMessage);
var P : TPoint;
begin
 case msg.LParam of
```

```
WM_LBUTTONDOWN:
  begin
   GetCursorPos(p);
   SetForegroundWindow(Application.MainForm.Handle);
   PopupMenu1.Popup(P.X, P.Y);
  end;
 WM_LBUTTONUP :
 end;
end;
```

Обратите внимание, что при показе всплывающего меню недостаточно просто вызвать метод `Popup`. При этом нужно вынести главную форму приложения на передний план, в противном случае она не получит сообщений от меню.

Теперь решим еще две задачи. Во-первых, как сделать, чтобы приложение минимизировалось не на Панель задач (TaskBar), а на **System Tray**? И более того — как сразу запустить его в минимизированном виде, а показывать главную форму только по наступлении определенного события (приходу почты, наступлению определенного времени и т. п.).

Ответ на первый вопрос очевиден. Если минимизировать не только окно главной формы приложения (`Application.MainForm.Handle`), но и окно приложения (`Application.Handle`), то приложение полностью исчезнет "с экранов радаров". В этот самый момент нужно создать значок на панели **System Tray**. В его всплывающем меню должен быть пункт, при выборе которого оба окна восстанавливаются, а значок удаляется.

Чтобы приложение запустилось сразу в минимизированном виде и без главной формы, следует к вышесказанному добавить установку свойства `Application.ShowMainForm` в значение `False`. Здесь возникает одна сложность — если главная форма создавалась в невидимом состоянии, ее компоненты будут также созданы невидимыми. Поэтому при первом ее показе установим их свойство `Visible` в значение `True`. Чтобы не повторять это дважды, установим флаг — глобальную переменную `ShownOnce`:

```
procedure TForm1.HideMainForm;
begin
  Application.ShowMainForm := False;
  ShowWindow(Application.Handle, SW_HIDE);
  ShowWindow(Application.MainForm.Handle, SW_HIDE);
end;

procedure TForm1.RestoreMainForm;
var i,j : Integer;
begin
  Application.ShowMainForm := True;
```

```
  ShowWindow(Application.Handle, SW_RESTORE);
  ShowWindow(Application.MainForm.Handle, SW_RESTORE);
  if not ShownOnce then
  begin
    for I := 0 to Application.MainForm.ComponentCount -1 do
      if Application.MainForm.Components[I] is TWinControl then
        with Application.MainForm.Components[I] as TWinControl do
          if Visible then
          begin
            ShowWindow(Handle, SW_SHOWDEFAULT);
            for J := 0 to ComponentCount -1 do
              if Components[J] is TWinControl then
                ShowWindow((Components[J] as TWinControl).Handle,
SW_SHOWDEFAULT);
          end;
    ShownOnce := True;
  end;
end;

procedure TForm1.WMSYSCOMMAND(var msg: TMessage);
begin
 inherited;
 if (Msg.wParam=SC_MINIMIZE) then
 begin
  HideMainForm;
  CreateTrayIcon(1);
 end;
end;

procedure TForm1.FileOpenItem1Click(Sender: TObject);
begin
  RestoreMainForm;
  DeleteTrayIcon(1);
end;
```

Теперь у вас в руках полноценный набор средств для работы с панелью **System Tray**. В заключение необходимо добавить, что все описанное реализуется не в операционной системе, а в оболочке ОС — Проводнике (Explorer). В принципе, и Windows NT 4/2000, и Windows 95/98 допускают замену оболочки ОС на другие, например DashBoard или LightStep. Там функции панели **System Tray** могут быть не реализованы или реализованы через другие API. Впрочем, случаи замены оболочки достаточно редки.

Интерфейс *IShellLink*

Этот интерфейс представляет собой средство для создания и управления ярлыками (shortcuts). Все читатели этой главы наверняка создавали и перемещали ярлыки для наиболее нужных программ, файлов и папок — на рабочем столе, в главном меню и т. д. С точки зрения ОС эти действия — не что иное, как создание и изменение свойств COM-объекта.

Каждый ярлык содержит следующую информацию:

- путь к объекту, на который ссылается ярлык (Path);
- рабочий каталог для этого объекта (Working Directory);
- список параметров, передаваемый объекту при его активизации (Arguments);
- начальное состояние окна, соответствующего объекту (нормальное, минимизированное, максимизированное) (ShowCmd);
- путь к значку, соответствующему объекту (Icon Location);
- описание объекта (Description);
- сочетание "горячих" клавиш (HotKey).

Для всех этих свойств ярлыка в интерфейсе дано по паре методов — один для чтения, другой для установки значения:

```
IShellLink = interface(IUnknown) { sl }
    [SID_IShellLinkA]
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
      var pfd: TWin32FindData; fFlags: DWORD): HResult; stdcall;
    function GetIDList(var ppidl: PItemIDList): HResult; stdcall;
    function SetIDList(pidl: PItemIDList): HResult; stdcall;
    function GetDescription(pszName: PAnsiChar; cchMaxName: Integer):
HResult; stdcall;
    function SetDescription(pszName: PAnsiChar): HResult; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar; cchMaxPath: Integer):
HResult; stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HResult; stdcall;
    function GetArguments(pszArgs: PAnsiChar; cchMaxPath: Integer):
HResult; stdcall;
    function SetArguments(pszArgs: PAnsiChar): HResult; stdcall;
    function GetHotkey(var pwHotkey: Word): HResult; stdcall;
    function SetHotkey(wHotkey: Word): HResult; stdcall;
    function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
    function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
    function GetIconLocation(pszIconPath: PAnsiChar; cchIconPath: Integer;
      out piIcon: Integer): HResult; stdcall;
```

```
    function SetIconLocation(pszIconPath: PAnsiChar; iIcon: Integer):
    HResult; stdcall;
    function SetRelativePath(pszPathRel: PAnsiChar; dwReserved: DWORD):
    HResult; stdcall;
    function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
    function SetPath(pszFile: PAnsiChar): HResult; stdcall;
  end;
```

Сохраним ярлык для данной программы-примера где-нибудь на диске, скажем, в той же самой папке. Для этого создадим новый объект `NewLink` класса `CLSID_ShellLink`, предоставляющий нам нужный интерфейс:

```
procedure TForm1.Button1Click(Sender: TObject);
var NewLink : IShellLink;
    fn, fp : string;
    ws : WideString;
    hRes : THandle;
    pf : IPersistFile;
begin
 NewLink := CreateComObject(CLSID_ShellLink) as IShellLink;
 fn := ParamStr(0);
 NewLink.SetPath(pchar(fn));
 fp := ExtractFilePath(fn);
 NewLink.SetWorkingDirectory(pchar(fp));
 NewLink.SetDescription(pChar(Application.Title));
 ws := fp+Application.Title+'.lnk';
 hRes := NewLink.QueryInterface(IID_IPersistFile, pf);
 if Succeeded(hRes) then
  pf.Save(pWideChar(ws),False);
end;
```

В этом примере помимо `IShellLink` нужно получить доступ к интерфейсу `IPersistFile`, который "умеет" записывать данные. Задав параметры ярлыка, мы записываем его на диск. При этом проверяется тот факт, что созданный нами объект поддерживает интерфейс `IPersistFile`. Если указатель на этот интерфейс получен, вызывается его метод `Save`.

Среди перечисленных выше методов `IShellLink` особое внимание уделим методу `Resolve`. Он понадобится вам при получении указателя на интерфейс уже существующих ярлыков. Windows пытается вести себя "разумно" и отслеживает перемещения и переименования объекта, на который указывает существующий `IShellLink`. Но если вы записали содержимое ярлыка в поток (или на диск), то отследить соответствие ярлыка объекту должны сами, вызвав метод `Resolve`. Если объект, на который ссылается ярлык, по-прежнему находится на своем месте, метод немедленно завершается с нор-

мальным кодом возврата. Если файл или объект перемещен или переименован, начинается его поиск (рис. 31.2).

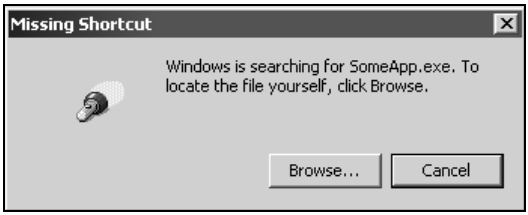


Рис. 31.2. Поиск объекта, на который указывает ярлык

Знакомая картина, не правда ли? Особенно часто она наблюдается в том случае, если пользователь не выработал у себя привычки правильно деинсталлировать раздобытый где-то "софт", стирая его "по старинке". Между тем, за привычным диалоговым окном на рисунке стоит вызов метода `IShellLink.Resolve`. Если в пределах досягаемости поиска окажется файл с тем же именем и размерами, ярлык будет автоматически переадресован на него; в противном случае пользователю будет предложено использовать ближайший по характеристикам файл из просмотренных. Если вы вообще не хотите, чтобы пользователь вмешивался в процесс отыскания соответствия, при вызове метода `Resolve` в параметре `fFlags` укажите значение `SLR_NO_UI` — диалоговое окно появляться в этом случае не будет.

Если вы внимательно изучили рабочий стол своего компьютера, то должны были заметить там ярлыки, ссылающиеся не на файлы, а на специальные объекты — "Мой компьютер", "Сетевое окружение", "Принтеры" и т. п. Чтобы создать такой ярлык самому, нужно обращение к методу `SetIDList`. В качестве параметра ему передается структура `PItemIDList` (`pidl`). О том, где ее взять и как заполнить, рассказано в следующем разделе.

Интерфейс *IShellFolder*

Этот интерфейс соответствует папке — одному из основных элементов пространства имен Проводника. Зачем было вводить термин "папка", когда существовали уже общепринятые "каталог" и "директория"? В отличие от последних двух, папка может быть не просто обычным элементом файловой системы. Она может быть виртуальной — как папки Принтеры, Документы или Панель управления. Любая папка может содержать коллекцию объектов из состава пространства имен.

Получив указатель на интерфейс `IShellFolder`, соответствующий папке, вы можете работать с ней, как с объектом COM. "Верхушкой" (корневой пап-

кой) пространства имен является папка Рабочий стол (Desktop). Получить интерфейс `IShellFolder` этой папки можно путем вызова функции:

```
function SHGetDesktopFolder(var ppshf: IShellFolder): HResult;
```

Логика работы с описываемым интерфейсом такова: сначала необходимо получить интерфейс нужной папки, а затем можно переходить к работе с ее содержимым. Содержимое представляет собой список, а каждый элемент папки представлен структурой `PItemIDList`. Эта структура не типизирована; ее единственное обязательное поле содержит длину в байтах, зная которую можно переместиться к следующему элементу. То есть получается обычная цепочка. Все остальные поля заполняются соответствующими функциями и методами интерфейса `IShellFolder`.

Примечание

Все служебные функции работы со структурами `PItemIDList` — создание, уничтожение, копирование, перемещение по цепочке и т. п. — содержатся в примере Virtual ListView, поставляемом с Delphi. Если вы намерены писать программы, работающие с `IShellFolder`, целесообразно взять их на заметку. В дальнейшем для простоты эти структуры будем именовать `pidl`.

Рассмотрим функции интерфейса `IShellFolder`. Под "текущей папкой" в табл. 31.1 понимается та папка, которая в данный момент представляет интерфейс `IShellFolder`.

Таблица 31.1. *Функции интерфейса* `IShellFolder`

Метод	Описание
`function ParseDisplayName(hwndOwner: HWND; pbcReserved: Pointer; lpszDisplayName: POLESTR; out pchEaten: ULONG; out ppidl: PItemIDList; var dwAttributes: ULONG): HResult;`	Эта функция позволяет получить указатель на элемент `ppidl`, зная только его полное имя (с путем) `lpszDisplayName`
`function EnumObjects(hwndOwner: HWND; grfFlags: DWORD; out EnumIDList: IEnumIDList): HResult;`	Возвращает указатель на специальный интерфейс `IEnumIDList`, предназначенный для организации цикла по всем элементам списка в текущей папке
`function BindToObject(pidl: PItemIDList; pbcReserved: Pointer; const riid: TIID; out ppvOut: Pointer): HResult;`	Возвращает интерфейс папки `pidl`, которая должна находиться в текущей папке (на которую ссылается интерфейс, вызвавший этот метод)
`function CompareIDs(lParam: LPARAM; pidl1, pidl2: PItemIDList): HResult;`	Сравнивает два первых элемента в списках `pidl1` и `pidl2`

Таблица 31.1 *(окончание)*

Метод	Описание
`function CreateViewObject(hwndOwner: HWND; const riid: TIID; out ppvOut: Pointer): HResult;`	Создает визуальный объект для текущей папки и возвращает указатель на него в параметре `ppvOut`
`function GetAttributesOf(cidl: UINT; var apidl: PItemIDList; var rgfInOut: UINT): HResult;`	Возвращает атрибуты элемента под номером `cidl` в списке `apidl`. Результат — набор флагов, устанавливаемых в параметре `rgfInOut`
`function GetUIObjectOf(hwndOwner: HWND; cidl: UINT; var apidl: PItemIDList; const riid: TIID; prgfInOut: Pointer; out ppvOut: Pointer): HResult;`	Создает объект пользовательского интерфейса, связанный с элементом списка `aplidl` под номером `cidl`
`function GetDisplayNameOf(pidl: PItemIDList; uFlags: DWORD; var lpName: TStrRet): HResult;`	Возвращает имя элемента `pidl`. Полнота возвращаемой информации определяется параметром `uFlags`
`function SetNameOf(hwndOwner: HWND; pidl: PItemIDList; lpszName: POLEStr; uFlags: DWORD; var ppidlOut: PItemIDList): HResult;`	Задает новое имя `lpszName` для списка `pidl`. При этом возвращается новый указатель на список — `ppidlOut`

Два метода — `ParseDisplayName` и `GetDisplayNameOf` — взаимно дополняют друг друга. Первый из них нужен, если вы имеете указатель на `IShellFolder` и хотите связать его с конкретной папкой. На практике это сводится к задаче в три действия:

1. Получить указатель на интерфейс какой-либо папки, скажем, рабочего стола, при помощи `ShGetDesktopFolder`.
2. Получить указатель (`pidl`) нужного вам элемента. Это осуществимо многими способами. Первый из них — как раз через вызов метода `IShellFolder.ParseDisplayName`. Если вы хотите получить доступ к одной из виртуальных (специальных) папок, то незаменимой будет следующая функция:

   ```
   function SHGetSpecialFolderLocation(hwndOwner: HWND; nFolder: Integer;
   var ppidl: PItemIDList): HResult;
   ```

   В параметре `nFolder` вы задаете константу, соответствующую выбранной специальной папке. На выходе будет указатель на элемент `ppidl`, соответствующий этой папке.

Примечание

Во многих функциях Shell API и методах его интерфейсов встречается параметр `hwndOwner`. Он должен задавать дескриптор окна на тот случай, если придется выводить диалоговое окно или окно с сообщением об ошибке.

Возможные значения параметра `nFolder` перечислены в табл. 31.2. В комментариях к ним "виртуальная" папка является особым объектом, который предоставляется пользователю при помощи Shell API. Просто "папка" реально существует где-то в файловой системе.

Таблица 31.2. *Константы, определяющие специальные папки*

Значение	Комментарий
`CSIDL_BITBUCKET`	**Корзина** (Recycle bin) — специальная папка для удаленных файлов. Пути к Recycle bin нет в системном реестре во избежание перемещения или удаления, и его не узнать иным методом
`CSIDL_CONTROLS`	**Панель инструментов** (Control Panel) — виртуальная папка, содержащая значки апплетов Панели инструментов
`CSIDL_DESKTOP`	Виртуальная папка **Рабочий стол** (Desktop), корневая в пространстве имен
`CSIDL_DESKTOPDIRECTORY`	Папка файловой системы, реально содержащая объекты рабочего стола
`CSIDL_DRIVES`	Виртуальная папка **Мой компьютер** (My Computer), содержащая элементы для всех накопителей на компьютере подключенных сетевых устройств, папки **Принтеры**, **Панель инструментов**, **Удаленный доступ к сети**
`CSIDL_FONTS`	Виртуальная папка **Шрифты**
`CSIDL_NETHOOD`	Папка, содержащая объекты сетевого окружения
`CSIDL_NETWORK`	Виртуальная папка **Сетевое окружение** (Network Neighborhood)
`CSIDL_PERSONAL`	Папка **Мои документы**
`CSIDL_PRINTERS`	Виртуальная папка **Принтеры** (Printers)
`CSIDL_PROGRAMS`	Папка **Программы** из главного меню, содержащая папки установленных на компьютере программ
`CSIDL_RECENT`	Папка, содержащая ссылки на последние использовавшиеся документы (Recent)
`CSIDL_SENDTO`	Папка, содержащая элементы контекстного меню **Send To...**

Таблица 31.2 *(окончание)*

Значение	Комментарий
CSIDL_STARTMENU	Папка, содержащая элементы главного меню **Пуск** (Start)
CSIDL_STARTUP	Папка, содержащая элементы меню **Автозапуск** (Startup)
CSIDL_TEMPLATES	Папка, содержащая шаблоны типовых документов

Третий вариант получить pidl нужной папки — интерактивный, с помощью функции Shell API.

```
function ShBrowseForFolder(var lpbi: TBrowseInfo): PItemIDList;
```

Перед ее вызовом следует заполнить структуру типа TBrowseInfo, содержащую в частности pidl того элемента, который будет корневым. После вызова функции пользователь увидит перед собой диалоговое окно выбора папки (рис. 31.3).

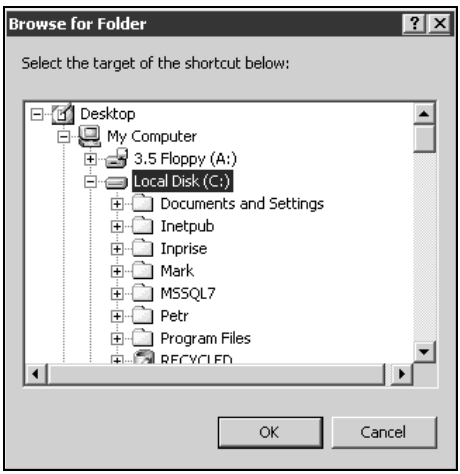


Рис. 31.3. Диалоговое окно выбора папки, созданное при вызове функции ShBrowseForFolder

В данном примере корневой служит виртуальная папка **My Computer**. Пользователю предоставляется возможность выбрать одну из папок файловой системы (за это отвечает флаг TBrowseInfo.ulFlags, равный BIF_RETURNONLYFSDIRS).

На выходе функция возвращает pidl папки, имя которой извлекается из него вызовом еще одной функции Shell — ShGetPathFromList.

```
procedure TForm1.Button1Click(Sender: TObject);
var
BI       : TBrowseInfo;
Image    : integer;
StartPIDL, ResPIDL : PItemIDList;
S, Path     : Array[0..max_path-1] Of WideChar;
begin

OleCheck(SHGetSpecialFolderLocation(Handle, CSIDL_DRIVES, StartPIDL));
With BI do
Begin
 hwndOwner      := Application.Handle;
 pszDisplayName := @S;
 lpszTitle      := 'Выберите необходимую папку';
 ulFlags        := BIF_RETURNONLYFSDIRS;
 pidlRoot       := StartPIDL;
 lpfn           := nil;
 iImage         := 1;
end;

 ResPIDL    := SHBrowseForFolder(BI);
 if SHGETPathFromIDList(ResPIDL, @Path[0])
  then Label1.Caption := StrPas(@Path[0]);

end;
```

Полученное имя здесь отображается при помощи компонента `Label1`.

3. Наконец, перейдем к третьему действию нашей задачи. Теперь, зная `pidl` папки, с которой вы будете работать, можно получить указатель на интерфейс `IShellFolder` вызовом метода `BindToObject`. Мы еще не рассмотрели такой важный аспект работы с папками, как просмотр их содержимого. Верные правилу COM: "каждый должен заниматься своим делом", разработчики Shell предоставили для просмотра еще один интерфейс — `IEnumIDList`. Пугаться нечего, набор возможностей этого интерфейса даже меньше, чем у пульта ДУ в магнитофоне. Его четыре метода — `Next`, `Skip`, `Reset` и `Clone` — позволяют организовать просмотр списка в одном направлении, а также возврат к началу и дублирование (Clone) выбранного элемента списка. Вот как это выглядит на практике.

```
...
Memo1.Clear;
 try
 OleCheck(SHGetDesktopFolder(DeskTop));
 if not Succeeded(DeskTop.ParseDisplayName(Self.Handle,nil,
 StringToWideChar (Edit1.Text,ws, MAX_PATH),n, pidl, attr))
```

```
    then begin ShowMessage('Неизвестное имя'); Exit; end;
   OleCheck(DeskTop.BindToObject(pidl,nil, IID_IShellFolder,
  Pointer(NewShellFolder)));
   OleCheck(NewShellFolder.EnumObjects(Self.Handle,
    SHCONTF_FOLDERS or SHCONTF_NONFOLDERS, Enumerator));
   while Enumerator.Next(1, pidl, Numpidls) = S_OK do
     begin
      NewShellFolder.GetDisplayNameOf(PIDL, SHGDN_FORPARSING, StrRet);
      case StrRet.uType of
      STRRET_CSTR:
        s := StrRet.cStr;
      STRRET_OFFSET:
        begin
          P := @PIDL.mkid.abID[StrRet.uOffset - SizeOf(PIDL.mkid.cb)];
          SetString(s, P, PIDL.mkid.cb - StrRet.uOffset);
        end;
      STRRET_WSTR:
        s := StrRet.pOleStr;
      end;//case

      Memo1.Lines.Add(s);
     end;
   except
    on E:EOleSysError do ShowMessage('');
   end;
```

В этом примере имя нужной папки извлекается из компонента `Edit1`. Получив указатель на интерфейс `IShellFolder` и затем интерфейс `IEnumIDList`, программа заполняет полученными именами файлов список `Memo1.Lines`.

Помимо названия из большинства объектов файловой системы можно "вытащить" массу полезной информации. Чаще всего задаются вопросом: а как извлечь значок, соответствующий данному файлу или хранящийся в нем?

Способов для достижения этой цели несколько. Самый простой — через вызов функции:

```
function SHGetFileInfo(pszPath: PAnsiChar; dwFileAttributes: DWORD;
var psfi: TSHFileInfo; cbFileInfo, uFlags: UINT): DWORD;
```

Параметр `pszPath` может быть указателем как на строку с именем файла, так и на структуру вида `pidl`. Функция заполняет структуру `psfi` (тип `TSHFileInfo`) длиной `cbFileInfo` байт. В зависимости от значения слова флагов (параметр `uFlags`) на выходе может быть разнообразная информация. В частности, если в параметре `uFlags` заданы значения `SHGFI_SYSICONINDEX` и `SHGFI_ICON`, то в структуру `psfi` будет записан номер значка для данного

файла в системном списке изображений, а результатом выполнения функции будет дескриптор этого списка. Воспользоваться им можно (например, для панели инструментов) так:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  FileInfo: TSHFileInfo;
  ImageListHandle: THandle;
begin
  ImageListHandle := SHGetFileInfo('C:\',
                          0,
                          FileInfo,
                          SizeOf(FileInfo),
                          SHGFI_SYSICONINDEX or SHGFI_ICON);
  SendMessage(ToolBar1.Handle, TB_SETIMAGELIST, 0, ImageListHandle);
end;
```

Точно так же можно извлечь значок, соответствующий конкретному файлу.

В составе Shell есть другие функции, созданные для извлечения значков:

- `function ExtractIcon(hInst: HINST; lpszExeFileName: PChar; nIconIndex: UINT): HICON;`

  Эта функция извлекает значок из файла `lpszExeFileName` (это должен быть файл типа EXE, DLL или ICO) и возвращает его дескриптор. Если значок не найден, возвращаемое значение равно 0.

- `function ExtractAssociatedIcon(hInst: HINST; lpIconPath: PChar; var lpiIcon: Word): HICON;`

  Эта функция может работать с файлами разных форматов. Сначала она, как и предыдущая, ищет значок в теле файла. Если его там нет, предпринимается попытка отыскать значок в приложении, связанном с данным типом файлов. Например, из файла с расширением doc будет извлечен один из значков Microsoft Word.

Добавление пунктов в системное контекстное меню

Вы обращали внимание на то, что некоторые приложения после установки добавляют в системное контекстное меню свои собственные пункты? Так поступают многие архиваторы, антивирусные средства и другие утилиты. Эта возможность предоставляется оболочкой Windows.

Когда пользователь щелкает правой кнопкой мыши на любом объекте в пространстве имен, система создает контекстное меню из двух частей: стандартного меню для объектов данного типа и пунктов меню, добавляемых

зарегистрированными обработчиками. Зарегистрированные обработчики — это COM-серверы, запускаемые в адресном пространстве процесса (in-process servers) и реализованные в виде динамических библиотек.

Ваш COM-объект, который расширяет системное контекстное меню, должен поддерживать как минимум два интерфейса — `IShellExtInit` и `IContextMenu`. Существуют и два новых интерфейса — `IContextMenu2` и `IContextMenu3`, но они вносят в логику работы контекстных меню лишь небольшие дополнения и здесь рассмотрены не будут. Интерфейс `IShellExtInit` отвечает за инициализацию меню, а интерфейс `IContextMenu` — за выполнение основных функций.

Методы интерфейса `IContextMenu` приведены в табл. 31.3.

Таблица 31.3. *Методы интерфейса* `IContextMenu`

Метод	Описание
`function QueryContextMenu (Menu: HMENU; indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;`	Добавляет пункт к системному контекстному меню
`function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;`	Осуществляет вызов обработчика
`function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT; PszName: LPSTR; cchMax: UINT): HResult; stdcall;`	Возвращает описание добавленного пункта меню (подсказку или полное название)

Рассмотрим их подробнее. Параметры метода `QueryContextMenu` означают следующее:

- `Menu` — дескриптор системного меню;
- `indexMenu` — позиция в меню, в которую следует вставить пункт (пункты);
- `idCmdFirst`, `idCmdLast` — диапазон допустимых значений для идентификаторов вставляемых пунктов меню;
- `uFlags` — набор флагов, главные из которых означают:
  - `CMF_NORMAL` — обычный вызов контекстного меню, пункты могут быть добавлены. Значение этого флага нулевое, проверять его следует, очистив все биты в параметре `uFlags`, кроме пяти младших (маска $1F);
  - `CMF_DEFAULTONLY` — устанавливается, если пользователь задал с объектом действие по умолчанию (например, двойной щелчок). В этом случае пункты меню добавляться не должны;

- CMF_VERBSONLY — устанавливается, если меню создается для ярлыка объекта, а не для самого объекта. В этом случае многие пункты меню создаваться не должны;
- CMF_EXPLORE — устанавливается, если меню создается для объекта, находящегося на левой панели Проводника.

Для иллюстрации объектов — расширений контекстного меню — выберем пример ContMenu (поставляется с Delphi в папке DEMOS\ACTIVEX \SHELLEXT). В этом примере для объектов типа "проект Delphi" добавляется возможность запуска компилятора в командной строке. При вызове метода QueryContextMenu нужный пункт добавляется с помощью функции InsertMenu:

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
          idCmdLast, uFlags: UINT): HResult;
begin
  Result := 0; // или использовать MakeResult(SEVERITY_SUCCESS,
               //                              FACILITY_NULL, 0);
  if ((uFlags and $0000000F) = CMF_NORMAL) or
     ((uFlags and CMF_EXPLORE) <> 0) then begin
    // Добавить один пункт меню во всплывающее меню
    InsertMenu(Menu, indexMenu, MF_STRING or MF_BYPOSITION, idCmdFirst,
      'Compile...');
   Result := 1; // или использовать MakeResult(SEVERITY_SUCCESS,
                //                              FACILITY_NULL, 1)
  end;
end;
```

Метод GetCommandString предоставляет системе данные о пункте меню, в частности, текст подсказки; эта подсказка будет отображаться в строке состояния Проводника, когда курсор находится в нужном месте меню.

Параметры GetCommandString просты. Первый — idCmd — соответствует идентификатору пункта меню, второй — uType — запрос на тип информации (GCS_HELPTEXT — текст подсказки, GCS_VERB — полное название пункта меню). Наконец, параметры pszName и cchMax задают буфер, в который будут копироваться текстовые данные. Полное название необходимо системе, чтобы с его помощью вызывать предусмотренные в пункте действия программно. В примере ContMenu возврат названия (т. е. обработка запроса GCS_VERB) не предусмотрен, а в ответ на запрос GCS_HELPTEXT возвращается текстовая строка "Compile the selected Delphi project".

Наиболее сложным является метод InvokeCommand. Он вызывается при выборе пользователем вставленного вами пункта меню. По сути дела метод InvokeCommand представляет собой прямой аналог обработчика OnClick обычных пунктов меню (объектов TMenuItem) в Delphi.

Единственным параметром метода является структура типа `TCMInvokeCommandInfo`, поля которой имеют такое предназначение:

- `cbSize` — размер структуры в байтах;
- `hwnd` — задает дескриптор окна, которое будет владельцем диалоговых окон, вызываемых из метода;
- `fMask` — определяет, заданы ли параметры `dwHotkey/hIcon`;
- `lpVerb` — вызываемая команда;
- `lpParameters` — параметры (если есть);
- `lpDirectory` — рабочая папка (поле не обязательно);
- `nShow` — флаг состояния окна, который будет передан в функцию `ShowWindow (SW_*)`;
- `dwHotKey` — "горячая" комбинация клавиш, которая будет сопоставляться приложению, запускаемому из этого пункта меню (только если в параметре `fMask` установлен флаг `CMIC_MASK_HOTKEY`);
- `hIcon` — значок, который будет сопоставляться приложению, запускаемому из этого пункта меню (только если в параметре `fMask` установлен флаг `CMIC_MASK_ICON`);
- `hMonitor` — монитор по умолчанию (поле не обязательно).

Отдельно следует остановиться на описании параметра `lpVerb`. Как уже говорилось, он может представлять из себя как идентификатор пункта меню, так и его текст — строку, заканчивающуюся нулем. Чтобы выяснить это, нужно проверить старшее слово этого 32-разрядного параметра на равенство нулю. В примере ContMenu вызов по тексту не предусмотрен:

```
if (HiWord(Integer(lpici.lpVerb)) <> 0) then
  begin
    Exit;
  end;
```

Для создания расширения контекстного меню мы должны породить объект, поддерживающий эти интерфейсы. К сожалению, мастера, предусмотренные в Delphi, не позволяют в автоматизированном режиме создавать объекты, реализующие уже существующие интерфейсы. Поэтому и описание, и реализацию методов придется делать "по старинке", вручную. В примере ContMenu описание объекта таково:

```
TContextMenu = class(TComObject, IShellExtInit, IContextMenu)
  private
    FFileName: array[0..MAX_PATH] of Char;
  protected
    { IShellExtInit }
```

```
    function IShellExtInit.Initialize = SEIInitialize;
    function SEIInitialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
      hKeyProgID: HKEY): HResult; stdcall;
    { IContextMenu }
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
idCmdLast,
      uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
  end;
```

Вас может насторожить конструкция, описывающая переименование метода `Initialize` интерфейса `IShellExtInit`. На самом деле одноименный метод имеется у объекта `TComObject`, и приведенный синтаксис как раз и предназначен для выхода из подобных ситуаций.

Последняя часть работы — регистрация созданного обработчика. Самое подходящее место для этого — метод `UpdateRegistry` фабрики класса. Разработчики примера ContMenu породили класс `TContextMenuFactory`, который при регистрации COM-сервера регистрирует создаваемые фабрикой объекты:

```
ClassID := GUIDToString(Class_ContextMenu);
CreateRegKey('DelphiProject\shellex', '', '');
CreateRegKey('DelphiProject\shellex\ContextMenuHandlers', '', '');
CreateRegKey('DelphiProject\shellex\ContextMenuHandlers\ContMenu', '',
ClassID);
```

Пример ContMenu иллюстрирует "дельфийский" подход к созданию серверов COM через соответствующие объекты из иерархии объектов Delphi. Но в папке SHELLEXT вы найдете еще один пример создания расширения для контекстного меню, сделанный целиком и только с использованием интерфейсов и функций COM. Присмотритесь к этому примеру внимательнее, если хотите глубже понимать внутреннюю структуру COM-объектов.

Резюме

Несколько тем, затронутых в этой главе, могут дать лишь начальные представления о принципах работы с оболочкой Windows. Вы можете изучить составляющие ее объекты практически сколь угодно глубоко — были бы потребность да желание. Разумеется, чтобы не "наломать дров", перед этим надо отдать себе отчет в полном и правильном понимании механизмов COM.

ПРИЛОЖЕНИЕ

Описание дискеты

Уважаемый читатель!

Ниже приводится описание примеров к этой книге. Каждый пример хранится в отдельной папке и представляет собой полноценный проект, который можно загружать в среду разработки и компилировать.

Название каталогов состоит из номера главы, в которой рассматривается пример, и порядкового номера примера по тексту главы.

Некоторые приложения используют в качестве источника данных демонстрационную базу данных Delphi, которая при стандартной инсталляции находится в папке \Program Files\Common Files\Barland Shared\Data. При работе с такими приложениями вам необходимо самостоятельно настроить соответствующие свойства компонентов доступа к данным. Они преднамеренно обнулены, т. к. на вашем компьютере база данных Delphi может располагаться в другом месте.

В приложениях, использующих генератор отчетов Rave Reports, в компоненте `TRvProject` необходимо указать полный путь к файлу проекта RAV, поскольку, хотя этот файл и располагается в папке приложения, отсутствие полного пути может приводить к ошибке.

Ниже приводится описание демонстрационных приложений.

Папка	Приложение
05_1	Пример использования компонентов `TTreeView` и `TTreeList` для просмотра информации из системного реестра Windows
05_2	Простой пример разработки собственного компонента
06_1	Ресурсы манифеста Windows XP
07_1	Пример использования в приложении списка объектов на основе класса `TList`

(продолжение)

Папка	Приложение
07_2	Пример использования в приложении списка строк на основе класса `TStringList`
10_1	Пример приложения, использующего компоненты Delphi для отображения графики
10_2	Приложение для просмотра растровых приложений JPG, JPEG, BMP
11_1	Пример простейшего приложения баз данных
12_1	Пример использования параметров компонентов запросов SQL и взаимодействия таких компонентов на основе передачи значений параметров
14_1	Приложение баз данных, использующее отношение "один-ко-многим" между таблицами базы данных
14_2	Приложение баз данных, демонстрирующее варианты поиска записей в таблице базы данных
14_3	Пример использования закладок (класс `TBookmark`) в наборах данных Delphi
15_1	Приложение баз данных, использующее компоненты синхронного просмотра
16_1	Приложение баз данных, напрямую использующее API BDE для полного удаления записей из таблиц базы данных
16_2	Приложение баз данных, напрямую использующее API BDE для представления данных
17_1	Приложение баз данных, демонстрирующее возможности технологии dbExpress
19_1	Приложение баз данных, демонстрирующее возможности технологии ADO
21_1	Пример простого распределенного приложения баз данных
25_1	Пример использования компонента проекта отчета Rave Reports и разработки простых отчетов в визуальной среде Rave Reports
26_1	Пример использования настраиваемого соединения на основе компонента `TRvCustomConnection`
26_2	Пример отчетов Rave Reports для приложений баз данных
27_1	Пример реализации Drag-and-Drop
27_2	Пример реализации Drag-and-Dock
27_3	Пример реализации управления мышью
28_1	Пример создания динамической библиотеки

(окончание)

Папка	Приложение
29_1	Приложение, использующее отдельный поток к памяти для расчета числа π
30_1	Приложение, использующее компоненты многомерного представления данных
31_1	Пример приложения, использующего Shell API

Предметный указатель

A

Abort 82
Action 175
Action client 175
Action target 175
ADO 483
API BDE 392
as 39

B

BDE 379
brcc32.exe 111

C

Comctl32.dll 98
ConnectionString 492
Critical section 722

D

DataSnap 570, 575
dbExpress 424
default 22
Device Dependent Bitmap 239
Device Independent Bitmap 239
DLL_PROCESS_ATTACH 691
DLL_PROCESS_DETACH 691
DLL_THREAD_ATTACH 691
DLL_THREAD_DETACH 692
Drag-and-Dock 669
Drag-and-Drop 664
dynamic 28, 32
Dynamic Method Table 32

E

EAbort 82
EAssertionFailed 82
EFCreateError 218
EFOpenError 218
EInOutError 69
EIntError 71
EReadError 218
EReconcileError 591
Event 721
EWriteError 218
ExceptAddr 76
Exception 65
ExceptObject 76

I

IAccessor 488
IAppServer 555, 558, 571
IColumnsInfo 488
ICommand 489
ICommandPrepare 490
ICommandProperties 490
ICommandText 490
ICommandWithParameters 490
IConvertType 488
IDBCreateSession 487
IDBInitialize 487
IDBProperties 487
inherited 19, 32
IOpenRowset 488

IOResult 69
IProviderSupport 558, 572
IRowset 488
IRowsetChange 489
IRowsetIdentity 489
IRowsetIndex 489
IRowsetInfo 488
IRowsetLocate 489
IRowsetUpdate 489
is 39
IShellFolder 760
IShellLink 758
ISourcesRowset 486
ISQLCommand 449
ISQLConnection 448
ISQLCursor 450
ISQLDriver 447
ITransaction 488
ITransactionJoin 488
ITransactionLocal 488
ITransactionObject 488

J

JPEG 243

M

Microsoft ActiveX Data Objects 483
Mutex 721

O

OLE DB 484
OnException 75
overload 34
override 33

P

private 35
property 20
protected 36
public 35
published 36

R

raise 68, 72, 78
reintroduce 35

S

Self 17, 25
Semaphore 721
Sender 24
ShowException 74
SQL Links 380

T

TAction 175, 179
TActionList 175
TADOCommand 521
TADOConnection 492
TADODataSet 519
TADOQuery 520
TADOStoredProc 521
TADOTable 520
TAnimate 263
TBDEDataSet 406
TBitmap 238
TBrush 227
TCanvas 63, 227
TCaption 57
TChart 372
TClass 41
TClientDataSet 572, 574
TClipboard 254
TCollection 171
TCollectionItem 172
TColor 57
TColumn 349
TCommonCalendar 127
TComponent 51
TConnectionBroker 551
TControl 54
TControlState 58
TControlStyle 59
TCORBADataModule 558
TCubeDim 742
TCubeDims 742
TCursor 57
TCustomADODataSet 506
TCustomClientDataSet 573
TCustomControl 63
TCustomSQLDataSet 435
TCustomTabControl 101
TDatabase 401
TDataSet 279, 281, 305, 574
TDataSetField 587
TDataSetProvider 545, 554, 572

TDataSource 268, 269, 274, 300, 371
TDateTimePicker 127
TDBChart 372
TDBCheckBox 365
TDBComboBox 366
TDBCtrlGrid 359
TDBDataSet 412
TDBGrid 349
TDBGridColumns 349
TDBImage 367
TDBListBox 366
TDBLookupComboBox 372
TDBLookupControl 369
TDBLookupListBox 372
TDBMemo 367
TDBNavigator 277, 362
TDBRadioGroup 366
TDBRichEdit 368
TDBText 364
TDCOMConnection 539
TDecisionCube 736, 739
TDecisionGraph 744, 747
TDecisionGrid 744
TDecisionPivot 747, 748
TDecisionQuery 735, 739
TDecisionSource 743
TField 313
TFileStream 215
TFont 57, 225
TFontDialog 226
TGraphic 233
TGraphicControl 63
THandleStream 215
Theme API 149
TIBClientDataSet 573
TIBDatabase 458
TIBDatabaseInfo 479
TIBDataSet 472
TIBEvents 478
TIBGeneratorField 471
TIBQuery 470
TIBSQL 474
TIBSQLMonitor 481
TIBStoredProc 471
TIBTable 469
TIBTransaction 461
TIcon 238
TImage 245
TImageList 110
TIndexDef 295
TJPEGImage 243
TList 163
TListView 118
TLocalConnection 550
TMemoryStream 217
TMetafile 237
TMonthCalendar 126
TMTSDataModule 558
TNotifyEvent 24
TObject 28, 41, 47
TOpenPictureDialog 247
TPageControl 100
TParam 299
TParameter 516, 517
TPen 226
TPicture 235
TQuery 419
TRect 56
TRemoteDataModule 559
TRvNDRWriter 635
TRvProject 601, 602, 633
TRvRenderHTML 638
TRvRenderRTF 638
TRvRenderText 638
TRvSystem 601, 605, 633
try..except 69
try..finally 69
TSavePictureDialog 247
TScreen 256
TSearchRec 211
TSharedConnection 551
TSimpleDataSet 440
TSimpleObjectBroker 548
TSOAPDataModule 558
TSocketConnection 540
TSQLClientDataSet 573
TSQLConnection 426, 431, 434
TSQLDataSet 438
TSQLMonitor 451
TSQLQuery 439
TSQLStoredProc 439
TSQLTable 438
TStandardColorMap 151
TStatusBar 129
TStatusPanel 129
TStoredProc 421
TStream 213
TStringList 155
TStrings 154
TStringStream 218

TTabControl 100
TTable 273
TTabSheet 101
TThread 711
TToolBar 105
TToolButton 105
TTreeView 112
TTwilightColorMap 152
TWebConnection 543
TWebDataModule 558
TWidgetStyle 91
TWinControl 60
TXPColorMap 151
TXPManifest 145

V

virtual 28, 32
Virtual Method Table 32

W

widget 88, 90

X

XSQLDA 467
XSQLVAR 468

А

Агрегатные поля 586
Агрегаты 583
Агрегаты-объекты 584
Атрибуты файла 211

Б

Базовый приоритет 708
Битовая карта 238
Блок защиты ресурсов 73
Буфер Delta 577

В

Взаимное исключение 721, 724
Визуальный стиль 146
Виртуальный метод 30
Вложенные наборы данных 587
Внешний провайдер данных 573
Внутренний провайдер данных 573
Вызовы в DLL 689
Вычисляемое поле 320

Г

Группировка агрегатов 587

Д

Действие 175
Делегирование 24
Дескриптор окна 60
Деструктор 18
Диапазон 344
Динамическая библиотека ресурсов 700
Динамические поля 311
Динамический приоритет 709
Директива:
◊ pascal 690
◊ register 689
◊ safecall 690
◊ stdcall 690
◊ cdecl 690
Дочерний класс 27
Драйверы dbExpress 425

З

Закладка 342, 414
Значок 238

И

Индекс 293
Инициализация DLL 690
Инкапсуляция 27

К

Кисть 227
Класс 17
Клиент многозвенных приложений 570
Коллекции 170
Колонка 349
Команда ADO 489
Компоненты доступа к данным 269
Компоненты отображения данных 276, 347
Конструктор 18
Контекст устройства 224
Конфигурация BDE 383
Критическая секция 722, 724
Кросстаб 732

М

Манифест 143
Маска 330
Метафайл 237
Метод 17, 21
◊ абстрактный 31
◊ виртуальный 32
◊ динамический 32
◊ класса 18, 42
◊ перегружаемый 34
◊ статический 31
Метод-обработчик событий 24
Модуль ShareMem 703

Н

Набор данных 268, 279
Наследование 27
Неявный вызов DLL 694
Нормальный приоритет 708

О

Область видимости 35
Обработка исключительных ситуаций 64
Объект 18
◊ поля 309
Объект-источник данных ADO 487
Объект-команда ADO 489
Объект-набор рядов ADO 488
Объект-перечислитель ADO 486
Объект-сессия ADO 487
Объект-транзакция ADO 488
Объекты:
◊ в списках 157
◊ синхронизации 721
Объектный тип 17
Ограничения данных 328
Опережающее объявление класса 17
Отложенный ввод/вывод 209
Отношение 334, 336
Отображаемый файл 220

П

Пакет данных 575
Панель инструментов 105
Папка обмена 254
Параметры запроса SQL 298
Первичный индекс 296, 580
Перо 226
Позднее связывание 32
Поиск данных 337
Поиск файла 212
Поле 17, 20
Полиморфизм 30
Пользовательский интерфейс 268
Потоки 213, 704
Провайдер ADO 485, 490
Провайдер данных 545
Проект CLX 86
Проект динамической библиотеки 685
Процесс 725
Процессор баз данных 379
Псевдоним базы данных 383

Р

Родительский класс 27

С

Свойство 20
Семафор 721, 724
Сервер приложений 533, 554
Синхронизация потоков 721
Синхронный просмотр данных 368
Ситуация гонок 720
Событие 23, 721
Соответствие типов 29
Списки 154
◊ указателей 162

Статические поля 311
Страничный файл 220

Т

Тема 146
Типы:
◊ данных 323
◊ полей 314, 317
Тонкий клиент 535
Трехзвенная модель 534
Тупики 720

У

Удаленный модуль данных 537, 558
Указатель на метод 25

Ф

Файл DPR 19
Файловые переменные 196
Файлы, отображаемые в память 259
Фильтр 340, 406
Фокус ввода 674
Фоновые процедуры 707
Фоновый приоритет 708
Функция:
◊ IOResult 211
◊ блочного ввода/вывода 200
◊ ввода/вывода 197
◊ обратного вызова 691

Ц

Цвет 57

Ш

Шрифт 57, 225

Э

Экземпляр класса 18
Экспорт функций DLL 687

Я

Явный вызов DLL 696