Дмитрий Осипов

Delphi XE2

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.068+800.92Delphi XE2
ББК 32.973.26-018.1
О-74

Осипов Д. Л.

О-74 Delphi XE2. — СПб.: БХВ-Петербург, 2012. — 912 с.: ил. — (В подлиннике)

ISBN 978-5-9775-0825-4

Книга посвящена одному из самых совершенных языков программирования Delphi XE2. В ней излагаются основы программирования на языке Delphi XE2, подробно рассматривается визуальная библиотека компонентов (VCL), описывается порядок разработки программного обеспечения для 32- и 64-разрядных версий Windows с использованием функций Win API, предоставляется обзор новейшей кроссплатформенной библиотеки FireMonkey, позволяющей создавать программное обеспечение не только для ОС Microsoft Windows, но и для Mac OS X. Примеры проектов из книги размещены на сайте издательства.

Для программистов и студентов

УДК 681.3.068+800.92Delphi XE2
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*
Зав. производством	*Николай Тверских*

Подписано в печать 05.03.12.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 73,53.
Тираж 1200 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0825-4

Оглавление

Введение **1**

ЧАСТЬ I. ОСНОВЫ ЯЗЫКА DELPHI **3**

Глава 1. Знакомство **5**
Структура консольного приложения 6
Комментарии 7
Перевод листинга программы в машинные коды 8
Первая программа 10

Глава 2. Типы данных Delphi **11**
Переменные 11
Константы 13
Строки-ресурсы 13
Правила объявления идентификаторов 14
Типы данных 14
Простые типы 16
Целые числа 16
Символьный тип 16
Логический тип 17
Перечисления 18
Диапазоны 19
Обслуживание данных порядкового типа 19
Действительные типы 20
Строковый тип 21
Указатели 22
Вариантный тип 24

Глава 3. Структурные типы **25**
Множества 26
Записи 27
Вариантные поля 29
Усовершенствованная запись 30

Массивы ... 31
Объявление массива ... 32
Обращение к ячейкам массива ... 32
Динамический массив ... 33
Вариантные массивы ... 34

Глава 4. Операторы и выражения ... 36
Оператор присваивания ... 36
Арифметические операторы ... 36
Оператор конкатенации строк ... 37
Логические операторы ... 38
Операторы поразрядного сдвига ... 39
Операторы отношения ... 40
Операторы множеств ... 40
Составной оператор *begin..end* ... 41
Условный оператор *if..then..else* ... 42
Оператор-селектор *case* ... 43
Оператор перехода *goto* ... 45
Оператор *with..do* ... 45
Организация циклов ... 46
Цикл с параметром *for..do* ... 46
Инструкция *in* в цикле *for..do* ... 48
Цикл с предусловием *while..do* ... 48
Цикл с постусловием *repeat..until* ... 49
Вложенные циклы ... 49
Операторы *break* и *continue* ... 50

Глава 5. Процедуры и функции ... 52
Процедуры ... 53
Функции ... 55
Особенности передачи параметров ... 58
Директивы для процедур и функций ... 62
Перегрузка функций: директива *overload* ... 62
Опережающее объявление: директива *forward* ... 63
Внешнее объявление: директива *external* ... 65
Встроенная функция: директива *inline* ... 65
Рекурсивная функция ... 65
Процедурный тип данных ... 66
Анонимные функции ... 66

Глава 6. Файлы и каталоги ... 68
Типизированные файлы ... 68
Пример работы с типизированным файлом ... 72
Особенности удаления записей из больших файлов ... 76
Текстовые файлы ... 77
Двоичные файлы ... 79
Управление дисками, каталогами и файлами ... 80
Работа с дисками компьютера ... 81
Организация поиска каталогов и файлов ... 83
Проверка существования файла и каталога ... 85

Расположение системных каталогов ... 85
Создание, удаление, копирование и перемещение ... 86
Размер файла ... 87
Дата и время создания файла и каталога ... 88
Атрибуты файла и каталога ... 89

Глава 7. Концепция ООП ... 90
Основные понятия ООП ... 90
Абстрагирование ... 91
Инкапсуляция ... 92
Модульность ... 92
Наследование ... 93
Класс Delphi ... 93
Проектирование класса ... 94
Управление жизненным циклом объекта ... 96
Опережающее объявление класса ... 97
Ограничение видимости членов класса ... 99
Свойства объекта ... 100
Особенности объявления методов ... 101
Поля класса и методы класса ... 101
Иерархия наследования ... 102
Полиморфизм ... 103
Операторы класса ... 105
Аннотация класса ... 106
Создание и подключение атрибутов с аннотацией ... 106
Извлечение аннотации ... 107

Глава 8. Шаблоны ... 109
Обобщенный тип данных в полях записей ... 109
Обобщения в процедурах и функциях ... 111
Обобщенные типы данных в шаблонах классов ... 112
Наследование шаблона класса ... 113
Перегрузка методов с параметром обобщенного типа ... 114
Шаблон массива, класс *TArray* ... 115

ЧАСТЬ II. ПЛАТФОРМА VCL ... 117

Глава 9. Опорные классы VCL ... 119
Класс *TObject* ... 119
Управление жизненным циклом объекта ... 121
Информирование о классе ... 123
Класс *TPersistent* ... 125
Основа компонента, класс *TComponent* ... 126
Владение компонентом ... 127

Глава 10. Массивы указателей, наборы строк и коллекции ... 129
Массив указателей, класс *TList* ... 130
Контейнер объектов, класс *TObjectList* ... 133
Контейнер компонентов, класс *TComponentList* ... 133

Наборы строк, класс *TStrings* ... 134
Редактирование списка и управление данными ... 135
Загрузка и сохранение строк ... 137
Объединение строк ... 138
Поиск строки и объекта ... 139
Обслуживание данных "параметр — значение" ... 139
Оптимизация производительности ... 139
Особенности класса *TStringList* ... 140
Коллекции ... 141
Элемент коллекции, класс *TCollectionItem* ... 142
Создание и уничтожение коллекции ... 143
Доступ к элементу коллекции ... 143
Манипуляции элементами коллекции ... 143
Поиск элемента коллекции ... 144
Сравнение коллекций ... 144

Глава 11. Классы потоков данных ... 145
Прототип потоков данных, класс *TStream* ... 145
Потоки с дескриптором, класс *THandleStream* ... 147
Файловый поток данных, класс *TFileStream* ... 147
Пример работы с файловым потоком данных ... 149
Потоки данных в памяти ... 151
Поток данных в памяти *TMemoryStream* ... 152
Поток байтов *TBytesStream* ... 152
Поток строк *TStringStream* ... 153
Поток с возможностью сжатия данных ... 154
Сжатие данных *TZCompressionStream* ... 154
Восстановление данных *TZDecompressionStream* ... 155

Глава 12. Визуальные элементы управления и класс *TControl* ... 157
Принадлежность к родительскому контейнеру ... 157
Размещение и размеры элемента управления ... 158
События, связанные с изменением размеров ... 160
Пересчет клиентских и экранных координат ... 160
Выравнивание элемента управления в контейнере ... 161
Видимость и активность элемента управления ... 162
Внешний вид ... 163
Вывод текста ... 163
Оперативная подсказка ... 163
Контекстное меню ... 164
Командный объект ... 164
Поддержка естественного ввода ... 165
Обработка событий мыши ... 165
Щелчки кнопками мыши ... 165
Перехват щелчков мыши ... 169
Перемещение указателя мыши ... 170
Вращение колесика мыши ... 172
Операция перетаскивания drag and drop ... 173
Пример реализации операции drag and drop ... 175

Глава 13. Оконные элементы управления и класс *TWinControl* 178
Дескриптор окна 178
Управление подчиненными элементами 180
Выравнивание подчиненных элементов 181
Фокус ввода 182
Обработка событий клавиатуры 183
Операция буксировки drag and dock 185
Буксировка программным способом 189

Глава 14. Приложение VCL 191
Приложение VCL, класс *TApplication* 193
Оконная процедура 193
Общие возможности *Application* 195
Доступ к основным объектам приложения 197
Обработка поступающих сообщений 197
Управление процессом создания приложения 199
Завершение работы приложения 201
Сворачивание и восстановление размеров окна приложения 202
Диалоговое окно приложения 202
Осуществление оперативной подсказки 203
Подключение к справочной системе 204
События приложения, класс *TApplicationEvents* 205
Значок в области уведомлений 206
Пример работы с компонентом *TTrayIcon* 208
Стили оформления приложения 209
Менеджер стилей *TStyleManager* 211
Утилита создания стилей оформления 213

Глава 15. Форма, фрейм и модуль данных 215
Форма проекта VCL, класс *TForm* 215
Описание формы в dfm-файле 216
Стиль, поведение и оформление формы 218
Состояние формы 219
Создание, отображение и уничтожение форм 220
Вывод формы в модальном режиме 221
Закрытие формы 222
Уничтожение формы 223
Подключение меню 223
Фокус ввода 223
Полосы прокрутки 224
Особенности графического вывода формы 225
Выбор монитора для вывода формы 225
Масштабирование шрифта 226
Эффект прозрачности 226
Дескрипторы окна 228
Прикрепление формы к границам экрана 228
Обработка событий формы 228
Жизненный цикл формы 229
Нажатие быстрых клавиш 231

Пользовательские интерфейсы SDI и MDI ... 232
Особенности проекта MDI ... 233
Фрейм *TFrame* ... 237
Создание простого фрейма ... 237
Диалоги-помощники на основе фреймов ... 239
Модуль данных *TDataModule* ... 242

Глава 16. Исключительные ситуации ... 244
Защищенные от ошибок секции ... 244
Конструкция *try..except* ... 245
Конструкция *try..finally* ... 245
Вложенные конструкции *try* ... 246
Объектная модель исключительных ситуаций ... 247
Базовый класс *Exception* исключительной ситуации ... 249
Тихая исключительная ситуация *EAbort* ... 251
Исключительная ситуация отладки *EAssertionFailed* ... 252
Определение новых классов ИС ... 253
Расширенные возможности *try..except* ... 254
Централизованная обработка ошибок в приложении ... 255
Настройка поведения Delphi при обработке исключительных ситуаций ... 256

Глава 17. Компоненты отображения и редактирования текста ... 257
Компоненты отображения текста ... 257
Метка *TLabel* ... 257
Статический текст *TStaticText* ... 260
Метка-ссылка *TLinkLabel* ... 260
Компоненты редактирования текста ... 261
Основа текстовых редакторов, класс *TCustomEdit* ... 262
Ограничения на ввод ... 263
Выделение части текста ... 264
Взаимодействие с буфером обмена ... 265
Отмена изменений ... 265
Строка ввода *TEdit* ... 266
Строка ввода с кнопками *TButtonedEdit* ... 266
Строка ввода с меткой *TLabeledEdit* ... 266
Строка ввода с маской *TMaskEdit* ... 267
Многострочный текстовый редактор *TMemo* ... 268
Редактор расширенного текстового формата *TRichEdit* ... 269
Форматирование абзаца — класс *TParaAttributes* ... 270
Атрибуты текста — класс *TTextAttributes* ... 271
Особенности обработки событий ... 271

Глава 18. Кнопки и компоненты выбора значений ... 274
Кнопка *TButton* ... 274
Кнопка с рисунком *TBitBtn* ... 277
Кнопка-флажок *TCheckBox* ... 278
Кнопки изменения значения *TUpDown* ... 279
Кнопка выбора *TRadioButton* ... 280
Группа переключателей *TRadioGroup* ... 280

Группа кнопок *TButtonGroup* ... 281
Категории кнопок *TCategoryButtons* ... 284

Глава 19. Меню приложения ... 290
Опорный класс меню *TMenu* ... 291
Главное меню *TMainMenu* ... 292
Контекстное меню *TPopupMenu* ... 293
Элемент меню *TMenuItem* ... 294
Элемент меню в виде флажка ... 295
Элементы меню в виде группы выбора ... 296
Родительские и дочерние элементы меню ... 297
Присвоение элементам меню значков ... 298
Динамическое создание элементов меню ... 298
Удаление элементов меню ... 300
Элементы-разделители ... 300
Особенности прорисовки пункта меню ... 301

Глава 20. Управление приложением с помощью команд ... 303
Команда *TAction* ... 304
Связь с элементом управления ... 307
Выполнение команды ... 307
Установка в актуальное состояние ... 308
Связь команды с контейнером ... 308
Менеджеры команд ... 308
Общие черты менеджеров команд ... 309
Список команд *TActionList* ... 310
Менеджер команд *TActionManager* ... 310
Командные панели ... 311
Класс *TActionClientItem* ... 312
Опорный класс командных панелей *TCustomActionBar* ... 314
Панель главного меню *TActionMainMenuBar* ... 315
Инструментальная панель *TActionToolBar* ... 316
Контекстное командное меню *TPopupActionBar* ... 316
Настройка интерфейса во время выполнения приложения, диалог *TCustomizeDlg* ... 317
Редактор "горячих" клавиш *THotKey* ... 319

Глава 21. Списки ... 321
Опорный класс списков *TCustomListControl* ... 323
Общие черты списков, список *TListBox* ... 324
Замедление перебора элементов списка ... 327
Особенности обработки событий ... 327
Список с флажками выбора *TCheckListBox* ... 330
Список выбора цвета *TColorListBox* ... 331
Комбинированные списки, *TComboBox* ... 333
Улучшенный комбинированный список *TComboBoxEx* ... 335
Список просмотра *TListView* ... 337
Стиль представления данных ... 338
Особенности работы списка со стилем *vsReport* ... 339
Колонка *TListColumn* ... 340

Коллекция элементов списка *TListItems* ... 340
Элемент списка *TListItem* ... 342
Редактирование заголовка элемента ... 343
Выбор элементов списка ... 343
Упорядочивание элементов ... 345
Поиск элементов ... 345
Группировка элементов ... 346
Операции перерисовки ... 346
Пример работы с *TListView* ... 348

Глава 22. Сетки ... 353
Общие черты сеток, сетка *TDrawGrid* ... 354
Адресация ячейки ... 356
Обработка событий ... 356
Расширенные возможности по оформлению сетки ... 358
Сетка строк *TStringGrid* ... 359
Редактор списка значений *TValueListEditor* ... 363

Глава 23. Иерархические данные и компонент *TTreeView* ... 367
Сохранение и загрузка дерева ... 369
Выбор узла в дереве ... 369
Одновременный выбор нескольких узлов ... 370
Узел дерева *TTreeNode* ... 371
Положение узла в дереве ... 372
Родительские узлы ... 372
Дочерние узлы ... 373
Методы перехода между узлами дерева ... 373
Перемещение узла ... 374
Удаление узла ... 375
Значок узла ... 375
Свертывание и развертывание узла ... 376
Хранилище узлов класс *TTreeNodes* ... 377
Добавление узлов ... 378
Сортировка узлов ... 381
Удаление узлов из коллекции ... 382
Редактирование текста узла ... 383
Оформление дерева ... 383

Глава 24. Панели-контейнеры ... 387
Простые панели ... 387
Простая панель *TPanel* ... 388
Панель *TFlowPanel* ... 390
Панель-сетка *TGridPanel* ... 390
Область группировки *TGroupBox* ... 392
Контейнеры с возможностью скроллинга ... 393
Область с полосами прокрутки *TScrollBox* ... 393
Страница с кнопками прокрутки *TPageScroller* ... 394
Разделитель панелей, компонент *TSplitter* ... 395

Глава 25. Инструментальные планки ... 397
Инструментальная планка *TToolBar* ... 398
Кнопка *TToolButton* ... 398
Управление кнопками ... 402
Пользовательские настройки ... 403
Оформление ... 403
Планка *TCoolBar* ... 404
Дочерняя полоса *TCoolBar* ... 405
Планка управления *TControlBar* ... 406
Панель состояния *TStatusBar* ... 408

Глава 26. Наборы закладок и блокноты ... 412
Набор закладок, *TTabControl* ... 413
Закладки *TTabSet* и *TDockTabSet* ... 416
Блокнот *TPageControl* ... 416
Страница блокнота *TTabSheet* ... 418

Глава 27. Работа с датой и временем ... 420
Отсчет времени, таймер *TTimer* ... 421
Компоненты-календари, базовый класс *TCommonCalendar* ... 422
Календарь *TMonthCalendar* ... 424
Выбор даты/времени, компонент *TDateTimePicker* ... 425

Глава 28. Диалоговые окна ... 427
Окна вывода сообщений ... 427
Окна выбора действия ... 428
Создание многоразового окна выбора действия ... 431
Окна ввода данных ... 431
Окна выбора файлов и папок ... 432
Компоненты-диалоги ... 433
Диалоги открытия и сохранения файлов ... 435
Универсальные диалоги *TOpenDialog* и *TSaveDialog* ... 435
Особенности графических диалогов *TOpenPictureDialog* и *TSavePictureDialog* ... 441
Особенности текстовых диалогов *TOpenTextFileDialog* и *TSaveTextFileDialog* ... 441
Диалоги поиска и замены текста ... 441
Выбор шрифта *TFontDialog* ... 445
Выбор цвета *TColorDialog* ... 446
Параметры страницы *TPageSetupDialog* ... 447
Настройка печати *TPrinterSetupDialog* ... 449
Отправка задания на печать *TPrintDialog* ... 450
Диалог управления задачей *TTaskDialog* ... 451

Глава 29. Технология естественного ввода ... 454
Описание жеста ... 454
Реакция элементов управления на жест ... 456
Пример обработки стандартных жестов ... 458
Компоненты поддержки естественного ввода ... 459
Менеджер жестов *TGestureManager* ... 459
Доступ к жестам и их сохранение ... 460

Просмотр жестов, *TGestureListView* и *TGesturePreview* ... 462
Область ввода жеста *TGestureRecorder* ... 462
Виртуальная клавиатура *TTouchKeyboard* ... 463

Глава 30. Управление графическим выводом ... 465
Получение сведений об устройствах видеовывода ... 465
Изменение настроек дисплея ... 468
Исследование текущего состояния устройства ... 470
Взаимодействие с экраном, класс *TScreen* ... 471
Информация о рабочем столе ... 472
Управление видом указателя мыши ... 472
Информация о шрифтах системы ... 472
Информация о формах проекта ... 473
Информация об устройствах видеовывода ... 474
Реакция на события ... 474
Взаимодействие с дисплеем, класс *TMonitor* ... 474

Глава 31. Холст *TCanvas* ... 476
Представление цвета ... 477
Кисть *TBrush* ... 479
Перо *TPen* ... 481
Шрифт *TFont* ... 484
Холст *TCanvas* в VCL ... 487
Закраска области ... 487
Градиентная заливка ... 488
Графические примитивы ... 489
Линии ... 490
Простейшие геометрические фигуры ... 491
Дуги ... 492
Сплайн Безье ... 492
Копирование части холста ... 494

Глава 32. Растровая и векторная графика ... 496
Абстрактный базовый класс *TGraphic* ... 497
Значок *TIcon* ... 499
Формат BMP, класс *TBitmap* ... 500
Формат JPEG, класс *TJPEGImage* ... 504
Формат GIF, класс *TGifImage* ... 506
Управление фреймами рисунка GIF ... 508
Оптимизация рисунка GIF ... 509
Обработка событий ... 511
Формат PNG, класс *TPngImage* ... 511
Векторная графика, метафайл *TMetaFile* ... 514
Холст метафайла *TMetafileCanvas* ... 515
Универсальный контейнер *TPicture* ... 516
Универсальный контейнер *TWICImage* ... 517
Коллекция изображений *TImageList* ... 518
Загрузка образов в контейнер ... 519

Особенности отображения значков ... 521
Прозрачность ... 522
Экспорт значков из контейнера ... 523

Глава 33. Сложные графические задачи ... 524
Растровые операции ... 524
Управление прозрачностью ... 527
Системы координат и режимы отображения ... 530
Перенос начала координат ... 531
Управление страничными координатами ... 532
Мировые координаты и аффинные преобразования ... 534

Глава 34. Управление печатью ... 539
Описание принтера в Delphi, класс *TPrinter* ... 540
Выбор принтера ... 540
Управление страницей документа ... 541
Формирование и отправка задания на печать ... 541
Отмена задания ... 542
Печать многострочного текста ... 543
Особенности печати изображений ... 544
Пример печати изображений ... 544
Окно предварительного просмотра ... 545
Отправка задания на печать ... 550

ЧАСТЬ III. VCL И WINDOWS API ... 551

Глава 35. Реестр Windows ... 553
Класс *TRegistryIniFile* ... 554
Чтение из реестра ... 555
Запись в реестр ... 556
Удаление подраздела ... 557
Пример ... 557
Класс *TRegistry* ... 559
Создание и уничтожение экземпляра реестра ... 559
Работа с удаленным реестром ... 559
Доступ к разделам реестра ... 559
Чтение и запись значений в параметры ... 561
Получение информации о разделе ... 562
Получение сведений о параметре ... 563
Экспорт и импорт разделов реестра ... 563

Глава 36. Управление процессами ... 565
Создание процесса ... 566
Доступ к процессу ... 568
Приоритет процесса ... 569
Время выполнения процесса ... 570
Завершение процесса ... 571
Сбор информации о процессах Windows ... 572
Получение сведений о версии ОС ... 575

Глава 37. Многопоточные приложения ... 576
Поток *TThread* ... 576
Метод ожидания ... 580
Управление приоритетом потока ... 581
Время выполнения потока ... 582
Синхронный и асинхронный вызовы внешнего метода ... 582
Пример многопоточного приложения ... 582
Синхронизация потоков ... 586
Синхронизация событием *TEvent* ... 587
Критическая секция *TCriticalSection* ... 590
Мьютекс *TMutex* ... 590
Семафор *TSemaphore* ... 592

Глава 38. Взаимодействие процессов ... 594
Обмен данными через буфер обмена ... 594
Регистрация пользовательского формата буфера обмена ... 597
Обмен сообщениями ... 601
Поиск окна ... 602
Регистрация пользовательских сообщений ... 604
Пример обмена сообщениями между процессами ... 605
Файловое отображение ... 608

Глава 39. Сетевое взаимодействие ... 613
Почтовые слоты ... 613
Определение имени почтового слота ... 614
Управление почтовым слотом ... 615
Получение и отправка корреспонденции ... 615
Пример почтового приложения ... 616
Именованные каналы ... 619
Определение имени именованного канала ... 619
Создание именованного канала ... 620
Управление соединением с клиентом ... 622
Состояние канала ... 623
Подключение к каналу клиентского приложения ... 624
Разработка класса сервера именованного канала ... 625
Разработка класса клиента именованного канала ... 628
Сокеты ... 629
Классы сокетов в VCL ... 629
Общие черты сокетов, опорный класс *TIPSocket* ... 630
Отправка и получение данных ... 633
Сервер, компонент *TTCPServer* ... 634
Клиенты, компоненты *TTCPClient* и *TUDPSocket* ... 636
Пример приложения ... 636
Сокет-клиент ... 637
Сокет-сервер ... 638

Глава 40. Сервисы Windows ... 641
Менеджер управления сервисами ... 642
Управление сервисом ... 644
Состояние службы ... 646

Конфигурирование службы ... 647
Удаление службы ... 647
Сервис в VCL, класс *TService* ... 647
Идентификация ... 647
Тип сервиса ... 648
Определение прав на управление сервисом ... 648
Загрузка и запуск службы ... 648
Статус службы ... 649
Сбои при старте сервиса ... 650
Остановка и возобновление службы ... 650
Инсталляция и деинсталляция сервиса ... 651
Выполнение службы, поток *TServiceThread* ... 652
Ведение протокола службы ... 653
Приложение-сервис *TServiceApplication* ... 654
Пример ... 654
Регистрация сервиса в ручном режиме ... 657
Апплеты Панели управления ... 658
Апплет Панели управления, класс *TAppletModule* ... 659
Приложение Панели управления *TAppletApplication* ... 660
Пример апплета управления сервисом Windows ... 661
Приложение управления сервисом ... 661
Апплет Панели управления ... 662

Глава 41. Динамически подключаемые библиотеки ... 665
Создание проекта DLL ... 666
Объявление и экспорт функций в DLL ... 667
Соглашение о вызовах ... 667
Пример экспорта функций ... 668
Пример хранения форм в библиотеке ... 669
Вызов библиотеки из приложения ... 670
Неявное подключение DLL ... 671
Явное подключение DLL ... 673

Глава 42. Многокомпонентная модель COM ... 675
COM-объект ... 676
Понятие интерфейса ... 676
Базовый интерфейс *IUnknown* ... 678
Реализация интерфейса ... 679
Порядок вызова сервера COM ... 680
Интерфейс *IClassFactory* и библиотека COM ... 681
Реализация фабрики класса, класс *TComObjectFactory* ... 683
Реализация COM-объекта в Delphi ... 685
Класс *TComObject* ... 686
Класс *TTypedComObject* ... 686
Класс *TComServer* ... 687
Пример COM-проекта ... 688
COM-сервер ... 688
Помощник настройки COM-объекта ... 689
Шаблон кода с описанием класса ... 690
Библиотека типов ... 691

Главная форма сервера ... 696
Регистрация сервера ... 697
COM-клиент ... 698
Импорт библиотеки типов ... 698
Обращение к COM-объекту ... 700

Глава 43. Автоматизация ... 701
Интерфейс *IDispatch* ... 702
Диспинтерфейсы и дуальные интерфейсы ... 703
Контроллер автоматизации без применения библиотеки типов ... 703
Контроллер автоматизации с поддержкой библиотеки типов ... 705
Сервер автоматизации, базовый класс *TAutoObject* ... 708
Регистрация сервера автоматизации в таблице ROT ... 710
События автоматизации ... 711
Фабрика класса объекта автоматизации ... 714
Пример проекта автоматизации с поддержкой событий ... 714
Сервер автоматизации ... 715
Клиент автоматизации ... 722

Глава 44. Интерфейс *IShellFolder* ... 728
Идентификация объекта Shell ... 729
Диалоговое окно получения PIDL ... 729
Получение пути к системным папкам ... 731
Интерфейс *IShellFolder* ... 732
Получение PIDL из файлового пути ... 733
Получение интерфейса дочерней папки ... 733
Получение названия объекта по PIDL ... 734
Изменение названия объекта ... 735
Сбор дочерних объектов папки, интерфейс *IEnumIDList* ... 735
Атрибуты объекта ... 737
Сравнение объектов папки ... 738

Глава 45. DataSnap ... 739
Архитектура проекта DataSnap ... 740
Компоненты сервера ... 742
Сервер *TDSServer* ... 742
Обработка событий ... 742
Класс сервера *TDSServerClass* ... 745
Транспортные компоненты *TDSTCPServerTransport* и *TDSHTTPService* ... 746
Менеджер аутентификации *TDSAuthenticationManager* ... 748
Компоненты клиента ... 749
Соединение *TSQLConnection* ... 750
Проект DataSnap с использованием мастера ... 752
Подготовка клиентского приложения ... 756
Создание нового метода на сервере DataSnap ... 758
Доступ к новому методу из клиентского приложения ... 759
Проект DataSnap на основе пользовательского класса ... 760
Сервер ... 760
Клиент ... 762
Механизм обратного вызова ... 764

Глава 46. LiveBindings ... 767
Вводный пример LiveBindings ... 767
Класс *TBindExpression* ... 770
Выражение LiveBindings ... 772
Программная связь, класс *TBindings* ... 773

ЧАСТЬ IV. FIREMONKEY ... 777

Глава 47. Платформа FireMonkey ... 779
Опорный класс *TFmxObject* ... 779
Создание и уничтожение экземпляра класса ... 781
Сохранение объекта в памяти ... 781
Управление дочерними объектами ... 781
Сопоставление дополнительных данных ... 782
Элемент управления FMX — класс *TControl* ... 782
Размещение и выравнивание элемента управления ... 783
Выравнивание объекта ... 784
Масштабирование и вращение объекта ... 785
Видимость и прозрачность элемента управления ... 786
Обработка событий ... 786
Простейшие события — щелчок ... 786
Клавиатурные события ... 787
События мыши ... 787
События получения и утраты фокуса ввода ... 788
Событие изменения размера ... 788
События перетаскивания drag and drop ... 788
Особенности прорисовки элемента управления ... 790

Глава 48. Приложение FireMonkey ... 791
Выбор целевой платформы для проекта ... 791
Приложение *FMX.Forms.TApplication* ... 793
Общие черты форм HD и 3D ... 794
Форма HD *FMX.Forms.TForm* ... 795
Стили оформления формы, компонент *TStyleBook* ... 795
Трехмерная форма *FMX.Forms.TForm3D* ... 796
Пример 3D-проекта ... 798

Глава 49. Обзор компонентов для проектов HD ... 801
Панель-выноска *TCalloutPanel* ... 801
Разворачивающаяся панель *TExpander* ... 802
Компонент *TArcDial* ... 803
Компонент *TNumberBox* ... 803
Компонент *TComboTrackBar* ... 803
Компонент *TPopupBox* ... 804
Сетки *TGrid* и *TStringGrid* ... 805

Глава 50. Анимация ... 808
Анимация ... 808
Простой пример анимации ... 809

Общие черты компонентов-аниматоров, класс *TAnimation* ... 810
Индивидуальные особенности компонентов-аниматоров ... 812
Цветовая анимация, компонент *TColorAnimation* ... 813
Градиентная анимация, компонент *TGradientAnimation* ... 813
Анимированная картинка, компонент *TBitmapAnimation* ... 813
Анимированный ряд, компонент *TBitmapListAnimation* ... 813
Анимация числовых свойств, компонент *TFloatAnimation* ... 814
Анимация прямоугольной области, компонент *TRectAnimation* ... 814
Анимация траектории, компонент *TPathAnimation* ... 814
Управление графической производительностью ... 815

ПРИЛОЖЕНИЯ ... 817

Приложение 1. Математика, статистика и тригонометрия ... 819

Приложение 2. Работа со строками и символами ... 825
Системные настройки форматирования и класс *TFormatSettings* ... 829

Приложение 3. Работа с датой и временем ... 831
Представление даты и времени в текстовом формате ... 840

Приложение 4. Работа с памятью ... 843

Приложение 5. Управление ходом выполнения программы ... 845

Приложение 6. Работа с именами папок и файлов ... 846

Приложение 7. Модуль *IOUtils* ... 848

Приложение 8. Константы CSIDL ... 855

Приложение 9. Холст *FMX.Types.TCanvas* ... 859
Управление холстом ... 860
Кисть *FMX.Types.TBrush* ... 861
Внешний вид линий ... 862
Шрифт *FMX.Types.TFont* ... 863
Заливка замкнутых областей ... 863
Вывод простейших фигур ... 865
Вывод текста ... 865
Отображение рисунков ... 866
Отсечение ... 867
Сохранение и восстановление состояния холста ... 867

Приложение 10. Описание электронного архива ... 868

Предметный указатель ... 869

Введение

Главная задача книги, которую вы держите в своих руках, — вооружить читателя всеми необходимыми знаниями для создания профессиональных программных продуктов. В этом нашим помощником станет один из самых совершенных языков программирования Delphi XE2, входящий в программный пакет Embarcadero RAD Studio XE2. Delphi XE2 — это глубоко продуманный, высокоэффективный и многогранный язык программирования, позволяющий создавать приложения любой степени сложности, предназначенные для работы под управлением современных операционных систем Microsoft Windows, Mac OS X и iOS.

В чем причина популярности языка Delphi? У Delphi много достоинств, но если их упорядочить по значимости, то на первом месте окажутся простота и дружелюбие, за счет которых достигается высочайшая скорость разработки приложений. Эти положительные черты очень нравятся начинающим программистам — несколько щелчков мышью, и готово простейшее приложение! Четверть века назад о таком способе разработки программного обеспечения не могли даже и мечтать самые оптимистически настроенные программисты.

Хотя язык Delphi очень удобен для первых шагов начинающего программиста, в первую очередь язык предназначен для профессионалов. Ведь на Delphi можно написать программное обеспечение для всего спектра задач: системные утилиты, графические и текстовые редакторы, системы автоматизации производства, сетевое программное обеспечение, клиент-серверные системы, 3D-проекты мультимедиа, приложения баз данных. Перечень бесконечен... В качестве примера самого успешного проекта, написанного на Delphi, можно привести собственно среду проектирования Embarcadero RAD Studio XE2.

Книга разделена на четыре части.

- Если вы начинающий программист, то благодаря главам *части I* вы получите полное представление об основах программирования на языке Delphi. Здесь вы:
  - познакомитесь с понятием типа данных;
  - научитесь объявлять переменные и константы;
  - изучите структурные типы;
  - освоите операторы и выражения языка;
  - получите полное представление о порядке описания процедур и функций;
  - приобретете основополагающие знания о концепции объектно-ориентированного программирования.
- *Часть II* окажется одинаково полезной как для начинающего, так и для программиста, имеющего опыт работы в Delphi. Главы части посвящены подробному изучению визит-

ной карточки языка Delphi — визуальной библиотеки компонентов (Visual Components Library, VCL). Свою работу мы начнем с самых "глубин" библиотеки, где нас ждет встреча с фундаментальными классами VCL. В главах этой части рассматривается более сотни классов и компонентов Delphi, из которых, как из кирпичиков, складываются программы для Windows. Популярность современных версий Windows в первую очередь основана на развитом графическом интерфейсе пользователя, именно поэтому в *части II* книги мы досконально изучим вопросы применения деловой графики GDI в проектах Delphi для Windows. Здесь вы не только столкнетесь со стандартными способами вывода простейших геометрических фигур, текста и изображений, но и приобретете знания о сложных приемах работы с графикой и научитесь управлять стилями приложений Delphi.

- *Часть III* предназначена для программистов, разрабатывающих профессиональное программное обеспечение для 32- и 64-разрядных версий Windows с использованием функций Win API. Здесь представлена исчерпывающая информация о:
  - управлении процессами и потоками;
  - разработке служб Windows;
  - организации межпрограммного взаимодействия на основе буфера обмена, сообщений Windows и объектов файлового отображения;
  - проектировании динамически подключаемых библиотек;
  - использовании технологии COM и автоматизации;
  - работе с системным реестром;
  - проектировании сетевого программного обеспечения.

  Кроме того, в *части III* представлена фирменная технология Embarcadero, позволяющая разрабатывать клиент-серверные приложения DataSnap, и новейший механизм LiveBindings, предназначенный для организации "живого" взаимодействия между объектами Delphi.
- Заключительная часть книги *(часть IV)* знакомит читателя с новейшим достижением компании Embarcadero — кроссплатформенной библиотекой FireMonkey. FireMonkey позволяет разрабатывать программное обеспечение не только для 32- и 64-разрядных версий Windows, но и для операционных систем Mac OS X и iOS. Инновационная библиотека компонентов FMX способна создавать высококачественный двух- и трехмерный графический интерфейс пользователя.

Материал излагается системно и последовательно, предоставляя читателю профессиональную методику разработки современного программного обеспечения. Поэтому книга окажется одинаково полезной как начинающему программисту, так и хорошо подготовленному разработчику, который сможет использовать ее в качестве справочника. Электронный архив *(см. приложение 10)* содержит многочисленные примеры и дополнительные материалы.

ЧАСТЬ I

Основы языка Delphi

Глава 1. Знакомство
Глава 2. Типы данных Delphi
Глава 3. Структурные типы
Глава 4. Операторы и выражения
Глава 5. Процедуры и функции
Глава 6. Файлы и каталоги
Глава 7. Концепция ООП
Глава 8. Шаблоны

ГЛАВА 1

Знакомство

Часть I книги призвана научить читателя основам программирования на языке Delphi — потомке знаменитого языка Pascal, созданного известным швейцарским ученым, профессором Цюрихской высшей технической школы, обладателем премии Тьюринга (высшей награды для специалистов по информационным технологиям) Никлаусом Виртом (Niklaus Wirth).

Появившийся на свет в начале 1970-х годов язык Pascal предназначался для обучения студентов основам структурного программирования. За более чем четыре десятилетия своего развития Pascal не просто доказал свою жизнеспособность, он стал одним из ведущих языков разработки программного обеспечения. Во многом своему первому успеху язык Pascal обязан компании Borland, в стенах которой в 1992 г. появилась среда разработки Borland Pascal with Objects 7.0, и немного позднее, в 1995 г., — знаменитая Borland Delphi 1.0 для разработки программного обеспечения (ПО) для 16-разрядной ОС Microsoft Windows 3.*x*.

> ***Замечание***
>
> Имя Pascal было выбрано не случайно — язык был назван в честь французского математика Блеза Паскаля.

Успех первой версии Delphi и ее *библиотеки визуальных компонентов* (Visual Components Library, VCL) был столь оглушителен, что несколько лет спустя, опираясь на библиотеки Delphi, компания Borland создала еще один шедевр — среду C++Builder (на этот раз с базовым языком Си). Так, уже почти 20 лет, Delphi и C++Builder идут вместе. Эти два языка функционируют в одной среде разработки, обладают практически идентичными возможностями и отличаются лишь особенностями построения синтаксических конструкций, поэтому программист Delphi легко адаптируется к программам, написанным на C++Builder, и, наоборот, разработчик, имеющий опыт работы в среде C++Builder, поймет программу на Delphi.

Сегодня права на язык Delphi принадлежат компании Embarcadero Technologies. Новые владельцы Delphi не просто воспользовались успешными наработками программистов Borland, но и внесли в язык и среду разработки многочисленные улучшения и усовершенствования. В частности, современная Delphi позволяет разрабатывать 32- и 64-разрядные приложения для Windows на базе популярной библиотеки VCL и способна создавать кроссплатформенные приложения (для Windows, Mac OS X и iOS) на основе новейшей библиотеки FireMonkey. На момент выхода в свет Delphi XE2 осенью 2011 г. подобными возможностями не обладала ни одна среда разработки!

В настоящее время существуют две базовые разновидности Delphi:

- Delphi XE2 (16-я версия языка), предназначенная для создания программного обеспечения для Microsoft Windows, Mac OS X и iOS;
- Delphi Prism XE2 — язык разработки для платформы .NET (Microsoft Windows).

Эта книга посвящена наиболее новой версии языка — Delphi XE2, однако более половины примеров из книги сохранят работоспособность и в более ранних версиях среды разработки.

Структура консольного приложения

Для начала следует обратиться к фундаментальным основам языка Delphi, поэтому чтобы не "оглушить" читателя обилием пока непонятных терминов, мы, насколько возможно, постараемся абстрагироваться от сложной для начинающего программиста визуальной библиотеки компонентов (VCL). Девизом этой части книги можно считать разумный минимализм. Именно поэтому первые шаги изучения языка разработки мы сделаем в элементарных консольных приложениях, где нет отвлекающих новичка элементов управления и код максимально прост и линеен.

Для создания заготовки консольного приложения Delphi в главном меню среды разработки выберите пункт меню **File | New | Other** (Файл | Создать | Другое). Если все сделано верно, то перед нами появится окно **New Items** (Новый элемент) с открытым набором проектов **Delphi Projects**. Найдите на этой странице значок **Console Application** (Консольное приложение) и щелкните по кнопке **OK** (рис. 1.1). За этот труд Delphi отблагодарит нас заготовкой для самого простейшего приложения — консольного окна Windows.

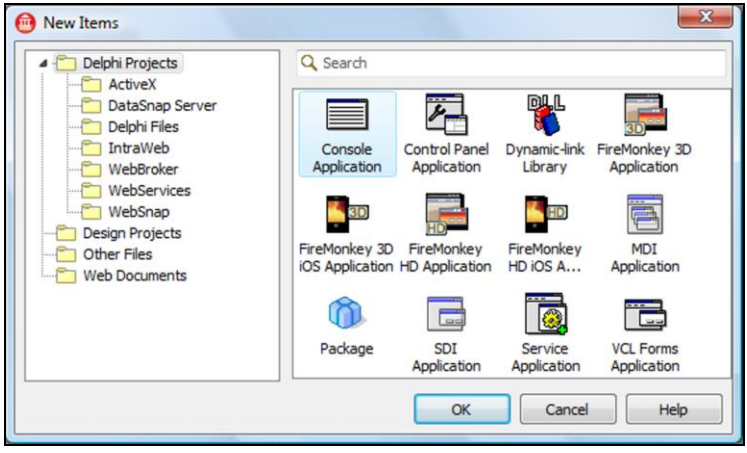


Рис. 1.1. Окно выбора типа проекта Delphi XE2

Сразу приучим себя к одному из ключевых правил программистов — исходный код программы должен регулярно сохраняться на жестком диске компьютера. Поэтому, прежде чем мы приступим к изучению заготовки консольного приложения, немедленно сохраним только что созданный проект. Для этого находим пункт главного меню среды разработки **File | Save All** (Файл | Сохранить все), в появившемся диалоге сохранения выбираем целевую (желательно пустую) папку, в которую отправятся файлы проекта. Среда проектирования обязательно спросит, под каким именем следует сохранить проект. Присвоим своему пер-

вому проекту имя (для первого раза вполне подойдет название по умолчанию — Project1.dproj). В завершение нажмем кнопку **OK**.

Обязательно откройте целевую папку в любом файловом менеджере, например в Проводнике Windows, и посмотрите, какие файлы были созданы Delphi во время сохранения нашего проекта:

- Project1.dproj — файл в формате расширяемого языка разметки (eXtensible Markup Language, XML), содержащий основные конфигурационные и справочные сведения о нашем проекте;
- Project1.dpr — головной файл с кодом проекта на языке Delphi;
- Project1.res — файл с ресурсами приложения.

После компиляции проекта появится еще ряд файлов, в том числе исполняемый файл с расширением exe. Но сейчас для нас наибольшую ценность представляет головной файл проекта — Project1.dpr. Именно в нем находится самое важное — исходный код будущего приложения. Этот же исходный код отобразится в редакторе Delphi (рис. 1.2).

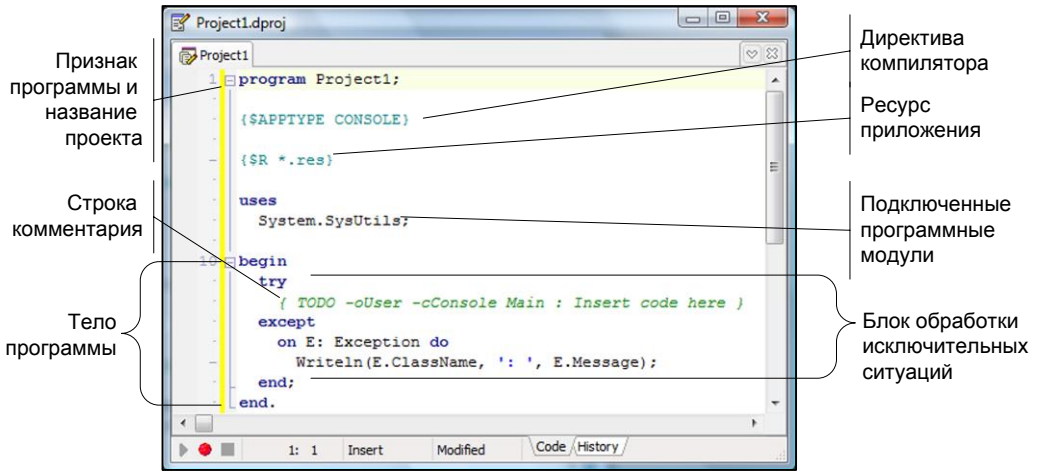


Рис. 1.2. Код консольного приложения Delphi XE2

Простейшее консольное приложение содержит немногим больше десяти строк (см. рис. 1.2). Листинг начинается с зарезервированного слова `program`. Это признак того, что перед нами листинг будущей программы. За словом `program` следует имя программы (в нашем случае `Project1`). Строкой ниже вы найдете директиву, определяющую порядок компиляции проекта. Еще ниже, после ключевого слова `uses`, приводится перечень подключаемых к проекту дополнительных программных модулей: в примере это всего лишь один модуль системных утилит — `SysUtils`, принадлежащий *пространству имен* `System`. Основной код программы располагается между ключевыми словами — `begin` и `end`. Пока здесь мы обнаружим только секцию обработки ошибок `try..except` и строку комментария.

Комментарии

Ни один, даже очень профессиональный программист, при разработке программного обеспечения не может обойтись без услуг комментариев. *Комментарий* — это не что иное, как краткое пояснение того, что происходит в тех или иных строках листинга. Человеческая

память не идеальна, и нередко нас подводит. А теперь представьте себе, что вы решили доработать код годовалой давности. Попробуйте быстро сообразить, для чего предназначалась переменная с "мудреным" названием `X` или что произойдет после вызова процедуры `MySuperProcedure()`?

Другими словами, в комментариях разработчик кратко поясняет смысл рожденных в его голове команд. В результате листинг программы становится более понятным и доступным для изучения.

Для того чтобы при компиляции программы текст комментариев не воспринимался Delphi в качестве команд и не служил источником ошибок, приняты следующие соглашения. Комментарием считается (листинг 1.1):

- отдельная строка, начинающаяся с двух наклонных черт `//`;
- весь текст, заключенный в фигурные `{ }` или в круглые скобки с символами звездочек `(* *)`.

Листинг 1.1. Примеры комментариев

```
//Одна строка комментария

{Текст
 многострочного
 комментария}

(*Это также
комментарий*)
```

Будьте внимательны! Если внутри фигурных скобок на первой позиции окажется символ `$`, то это не строка комментария, а директива компилятора. В шаблоне только что созданного нами приложения (см. рис. 1.2) такая директива имеется — `{$APPTYPE CONSOLE}`. Это означает, что наш проект должен стать консольным приложением.

> ***Внимание!***
>
> Директивы компилятора начинаются с символа открывающейся фигурной скобки и символа доллара `{$...}`. Начинающему программисту ни в коем случае не следует удалять, редактировать или изменять местоположение директив компилятора, в противном случае вы рискуете привести свой проект в негодность.

Перевод листинга программы в машинные коды

Для того чтобы листинги программы превратились в полноценное приложение, необходимо сделать их понятными для центрального процессора компьютера. Процессор не понимает языка Delphi (как, впрочем, и любых других языков высокого уровня), вместо этого процессор работает с машинными командами.

Чтобы листинг программы превратился в машинные коды, необходима помощь трех специальных подпрограмм: *препроцессора*, *компилятора* и *компоновщика*.

Препроцессор подготавливает программу к компиляции. Для этого он изучает и выполняет имеющиеся в программе директивы.

Компиляция — это процесс преобразования команд языка программирования в машинный код, понятный процессору компьютера. Во время компиляции программа-компилятор просматривает исходный код программы на предмет синтаксических ошибок и выполняет смысловой анализ кода.

На финальной стадии в дело вступает *компоновщик*. Он подключает к нашей программе все необходимые для ее работы модули, в первую очередь модули дополнительных библиотек.

В результате работы препроцессора, компилятора и компоновщика Delphi получается исполняемый файл с расширением exe. Полученный файл можно переносить на другой компьютер и использовать как самостоятельную программу для Windows.

> ***Замечание***
>
> Далее в книге процесс перевода листинга программы в машинные коды мы станем называть *компиляцией*.

Компилятор Delphi является высокотехнологичной программой и представляет собой ядро всей среды разработки. Для того чтобы полученная в результате компиляции программа стала высокопроизводительной, профессиональные программисты *оптимизируют параметры компиляции*. Для вызова окна с опциями компилятора следует воспользоваться пунктом меню **Project | Options** (Проект | Параметры) и в появившемся окне настроить параметры компилятора, выбрав элемент **Delphi Compiler** (рис. 1.3).

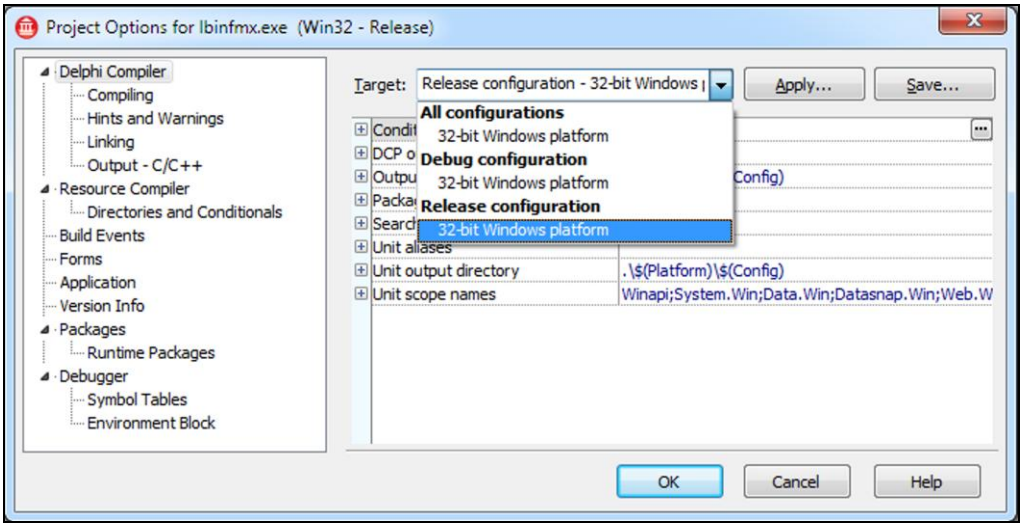


Рис. 1.3. Настройка параметров компиляции проекта

Без особой на то причины начинающему программисту не стоит экспериментировать с параметрами компилятора, однако необходимо знать следующее. По умолчанию компилятор настроен на работу в *режиме отладки* (debug). Этот режим используется во время проектирования приложения Delphi и позволяет эффективно выявлять и устранять ошибки в исходном коде. После окончательной победы программиста над всеми ошибками отладка программы завершается, самая последняя компиляция проекта осуществляется в режиме *выпуска конечного продукта* (release). Заказчику программного обеспечения ваши программы должны передаваться откомпилированными в режиме конечного продукта. Для этого следует изменить конфигурацию компилятора (**Build Configuration**), выбрав в раскрывающемся списке (см. рис. 1.3) вариант **Release configuration**.

Для старта компиляции проекта Delphi достаточно выбрать пункт меню **Run | Run** или (что еще проще) нажать клавишу <F9>. Если компилятор работает в режиме настроек по умолчанию, то в результате этих действий в папке с проектом появится дочерняя папка Debug, "порывшись" в которой вы обнаружите готовую к работе программу с расширением exe.

Первая программа

Настал тот час, когда мы готовы написать свою первую консольную программу на языке Delphi. Создайте новый проект консольного приложения и сохраните его в отдельную папку под любым именем (я выбрал имя Hello.dproj).

По давно сложившейся традиции (у ее истоков стояли известные программисты Брайан Керниган (Brian Kernighan) и Деннис Ритчи (Dennis Ritchie)) самой первой программой, открывающей книгу по программированию, станет программа-приветствие "Hello, world!", ее код представлен в листинге 1.2.

Листинг 1.2. Программа "Hello, world!"

```
program Hello;

{$APPTYPE CONSOLE}//это директива компилятора, которую мы не трогаем!

uses System.SysUtils;  (*строка подключения внешних библиотек подпрограмм, хотя,
между нами говоря, в этой программе внешние модули нам не нужны*)

begin
WriteLn('Hello, world!'); //выводим текст приветствия
ReadLn; //ожидаем ввод — нажатие любой клавиши завершит работу консоли
end.
```

Введя код программы, обратитесь к главному меню Delphi и выберите пункт **Run | Run** или воспользуйтесь клавишей <F9>. Если вы были внимательны и переписали строки листинга, не совершив ни одной ошибки, то за считанные доли секунды на экране появятся плоды нашего коллективного творчества — консольное окно со строкой "Hello, world!". Если же была допущена ошибка, то компилятор просигнализирует о ней, выделив в листинге подозрительную строку или следующую за ней строку.

Для вывода текстовой строки на экран компьютера нам понадобились услуги процедуры `WriteLn()`. Эта процедура еще не раз встретится нам в других примерах к этой книге, пока вам достаточно запомнить то, что `WriteLn()` позволяет отобразить в окне консольной программы текст, но при условии, что символы текста заключены в одинарные кавычки.

Завершая вводную главу, попрошу запомнить вас еще одну особенность синтаксиса языка Delphi — каждое выражение должно завершаться *точкой с запятой*, а признаком окончания всей программы является ключевое слово `end` *с точкой на конце*.

ГЛАВА 2

Типы данных Delphi

Функциональное назначение любой программы, будь это простейшее консольное приложение, выводящее на экран приветствие "Hello, world", или сложный математический пакет, моделирующий термоядерную реакцию, заключается в получении, обработке и возврате пользователю определенных результатов. Обслуживаемые программой данные размещаются в памяти компьютера в виде разнообразных структур, в простейшем случае это обычная последовательность байтов, а в более сложных ситуациях структура представляет собой весьма неординарную конструкцию, которая способна не просто хранить, но и управлять своими данными.

Сложность программы состоит в прямой зависимости от способов представления данных в памяти и задействованных для обслуживания этих данных алгоритмов. Николаусу Вирту принадлежит меткое определение программы:

Программа = структуры + алгоритмы.

Заметьте, что в определении профессора Вирта на первом месте стоят структуры, а алгоритмы только на втором. Это не случайность. Дело в том, что даже самый совершенный и глубоко проработанный алгоритм окажется малоэффективным, если подлежащие обработке данные предоставляются алгоритму в ненадлежащем виде. Здесь все как в автомобиле, никакой даже лучший на свете двигатель (читай — алгоритм) не в состоянии сдвинуть автомобиль с места, если к нему не прикручены колеса.

В этой главе мы начнем обсуждение первого ингредиента программы — структур. Здесь мы рассмотрим простейшие способы представления данных в программах Delphi, более сложные решения будут вынесены в следующую главу книги.

Переменные

Наиболее распространенный способ хранения данных реализуется при посредничестве *переменных*. Переменные позволяют размещать в памяти значения различных типов (целые и вещественные числа, символы, текстовые строки). Термин "переменная" (variable) подсказывает, что в ходе выполнения программы содержимое памяти может изменяться.

Объявление переменной включает упоминание слова `var` и указание имени и типа переменной.

```
var имя_переменной : тип данных;
```

Если необходимо объявить несколько однотипных переменных, то разрешается перечислить их, разделяя имена запятыми так, как представлено в листинге 2.1.

Листинг 2.1. Объявление переменных

```
var a: integer;  //переменная для хранения целого числа
    b: boolean;  //переменная для хранения логического значения
    c, d : real; //переменные для обслуживания действительных чисел
```

Более подробно о типах данных мы поговорим немного позднее, а пока достаточно понимать, что тип данных определяет особенность хранения и представления значений в переменных. Если необходимо оперировать целыми числами, то, скорее всего, вам пригодится тип данных `integer`, для обслуживания вещественных чисел воспользуйтесь типом `real`, для манипуляций отдельными символами понадобится `Char`.

> ***Примечание***
>
> Пока, при встрече с термином "тип данных", нам достаточно понимания того, что тип данных определяет число битов памяти, отдаваемых в распоряжение переменной. Чем больше битов, тем данные большего размера могут быть переданы в переменную. Добравшись до последних страниц главы, читатель увидит, что тип данных представляет собой более сложное понятие.

Современные версии Delphi позволяют проинициализировать содержимое переменной в момент ее объявления (листинг 2.2).

Листинг 2.2. Инициализация переменной во время объявления

```
var i: integer=10;    //целочисленная переменная i содержит значение 10
    b: boolean=false; //логическая переменная b содержит false
    c: char='!';      //символьная
```

Унаследованный от языка Pascal классический стиль программирования строго устанавливает области кода, в которых допускается объявление переменных. В консольных приложениях переменные должны объявляться перед телом программы, которое, в свою очередь, заключается внутрь составного оператора `begin..end`. При описании функции (процедуры, метода) все объявления локальных переменных должны быть сосредоточены между заголовком процедуры и началом ее реализации (листинг 2.3).

Листинг 2.3. Объявление локальной переменной в функции

```
function MyFunction: integer;
var i: integer; //локальная переменная i
begin
  i:=0; //переменная i будет доступна только внутри функции
  ...
end;
```

В зависимости от места объявления переменной можно говорить о ее локальном или глобальном статусе. *Глобальные переменные* проекта доступны или, как говорят программисты, видимы из всех областей программы. К такой переменной можно обратиться из любой процедуры или функции. В противовес глобальным переменным, область видимости их локальных "коллег" существенно у́же. Например, переменная, объявленная внутри функции (см. листинг 2.3), может быть задействована только в границах этой функции. Попытка

прочитать содержимое локальной переменной извне области ее видимости обречена на неудачу.

Константы

В отличие от переменных *константы* (constants) предназначены для хранения заранее известных и неизменяющихся во время выполнения программы значений. Объявление константы начинается с ключевого слова `const` и включает имя, необязательное указание типа и значение константы.

```
const имя_константы: [необязательный тип] = значение;
```

Пример объявления констант предложен в листинге 2.4. Обратите внимание на то, что для определения значения константы могут применяться выражения, например константа `c` станет хранить результат операции деления 5 на 3.

Листинг 2.4. Порядок объявления констант

```
const a = 3.14;
      b = 65535;
      c : extended = 5/3;
```

Так же как и в случае с переменными, в Delphi константа должна быть объявлена в строго отведенном для этого месте программы (идентичном местам объявления переменных). По аналогии с переменными константы обладают областью видимости, поэтому различают *глобальные* и *локальные константы*.

Строки-ресурсы

К одной из разновидности констант стоит отнести *строки-ресурсы* (resource strings). Строки-ресурсы состоят из текстовой информации и специальных символов форматирования (применяемых в функциях `Format()` и `FormatDateTime()`), вместо которых позднее можно вставить дополнительную информацию.

Листинг 2.5 демонстрирует один из вариантов применения строк-ресурсов.

Листинг 2.5. Порядок объявления констант

```
resourcestring ResDateTime = 'Дата d.mm.yyyy Время hh:mm';
var s:string;
begin
  S := FormatDateTime(ResDateTime, Now);
  WriteLn(S);
end.
```

В предложенном примере символы `d.mm.yyyy` замещаются текущей датой, а символы `hh:mm` — текущим временем.

Правила объявления идентификаторов

Все используемые в программах Delphi переменные, константы, процедуры, функции, объекты должны обладать уникальным именем — *идентификатором*.

В языке Delphi длина идентификатора не ограничена, но значащими являются только первые 255 символов. Идентификатор может содержать любые символы латинского алфавита, цифры и символ нижнего подчеркивания. Первый символ идентификатора обязан быть буквой. Само собой, в роли идентификаторов не допускается применять зарезервированные слова.

Различают *неквалифицированные* и *квалифицированные идентификаторы*. Синтаксис последних выглядит следующим образом:

```
идентификатор1.идентификатор2
```

Подобное обращение необходимо в тех ситуациях, когда требуемая переменная (функция, метод, свойство, объект и т. п.) принадлежит другому модулю (объекту, классу и т. д.). Таким образом, квалифицированный идентификатор знает всю цепочку своих владельцев.

Неквалифицированные идентификаторы не требуют явного указания своих владельцев. Это допускается в том случае, когда в принадлежности идентификатора сомнений не возникает.

> ***Замечание***
>
> В отличие от идентификаторов языка Си, идентификаторы языка Delphi *не чувствительны к регистру символов*, другими словами, объявив переменную `myvariable`, программисту разрешено обращаться к ней, не реагируя на регистр символов, например `MYVARIABLE` или `MyVariable`.

Типы данных

Отправной точкой процесса построения любой системы хранения и обработки данных по праву считается выбор *типа данных*. Вне зависимости от того, какой язык программирования выбран для разработки проекта, первым шагом становится рассмотрение типов данных, имеющихся в распоряжении системы.

В первую очередь тип данных определяет порядок хранения данных в памяти компьютера. Физическая концепция хранения данных зависит от особенностей конкретной платформы, на базе которой функционирует программное обеспечение. В первую очередь это утверждение относится к указателям. Например, в 32-разрядных ОС размер указателя равняется 4 байтам. В 64-битных системах это утверждение ошибочно, здесь для указателя требуется 8 байт.

> ***Внимание!***
>
> В Delphi XE2 появилась возможность создавать 64-разрядные приложения для Windows. Во всех предыдущих версиях Delphi эта опция отсутствовала.

Типизация хранимых значений не просто указывает на размер в байтах, которую должна выделить система для размещения в памяти того или иного значения. Преследуемая цель еще более значима. Типизация определяет, какие операции могут быть осуществлены с теми или иными данными. Delphi не позволит новичку передать результаты деления в целочисленную переменную, ведь в этом случае есть риск потерять дробную часть результата. Delphi станет отчаянно сопротивляться, если мы попробуем просуммировать символьный и вещественный типы данных, пусть даже в символьной переменной хранится числовое зна-

чение. В последнем случае перед проведением математической операции необходимо выполнить преобразование данных.

Программы функционируют в интересах человека, поэтому результаты их работы должны представляться в удобном для оператора виде. Во многом это достигается благодаря концепции типа данных, например при выводе на экран действительных чисел отображается целая и дробная части, разделенные запятой, а при выводе значений, специализирующихся на работе с датой и временем, дата и время форматируются так, как в принято в вашей стране.

Подытожим все вышесказанное. Понятие "тип данных" интегрирует в себе три компонента:

- ограничение множества значений, принадлежащих типу;
- дефиниция[1] набора операций, применяемых к типу;
- определение способа отображения (внешнего представления) значений типа.

Язык Delphi предоставляет в распоряжение программиста весьма широкий спектр *предопределенных типов данных* и, кроме того, разрешает конструировать собственные *пользовательские типы*. Вариант классификации фундаментальных типов данных Delphi представлен на рис. 2.1.

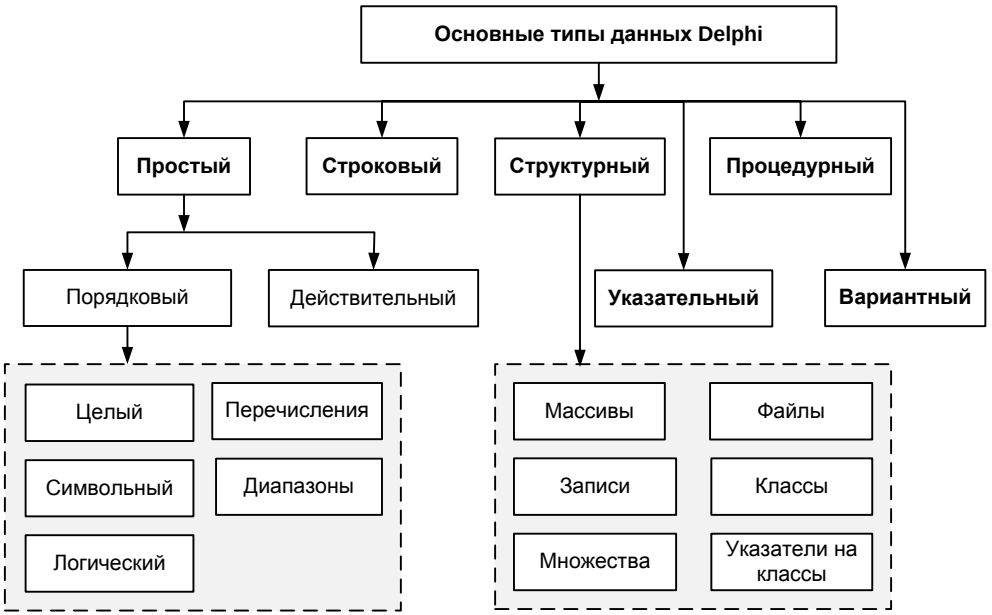


Рис. 2.1. Классификация типов данных Delphi

Все существующие в языке Delphi типы данных можно разделить на шесть ветвей:

- простые;
- строковые;
- структурные;
- указательные;
- процедурные;
- вариантные типы данных.

[1] Краткое определение какого-либо понятия, содержащее наиболее существенные его признаки.

В этой главе мы изучим простые, строковые и указательные типы данных, с оставшимися типами мы познакомимся в других главах книги.

Простые типы

Из представленных на рис. 2.1 типов данных наиболее востребованным является *простой*. Простой тип специализируется на обслуживании числовых данных и упорядоченных последовательностей, тип разделяется на два направления: *порядковые* и *действительные* типы.

Своим названием порядковый тип обязан тому, что между элементами этого типа существуют отношения порядка, другими словами, они могут расположиться в виде упорядоченной последовательности. К еще одной особенности порядковых типов данных стоит отнести то, что все без исключения простые типы по своей сути являются целыми числами.

Целые числа

Характерным представителем порядкового типа считаются целые числа, их характеристики вы найдете в табл. 2.1.

Таблица 2.1. *Целые числа*

Тип	Диапазон значений	Размер (байт)
`Shortint`, `Int8`	–128...127	1
`Smallint`, `Int16`	–32 768...32 767	2
`Longint`, `Integer`, `Int32`	–2 147 483 648...2 147 483 647	4
`Int64`	$-2^{63}...2^{63}-1$	8
`Byte`, `UInt8`	0...255	1
`Word`, `UInt16`	0...65 535	2
`Longword`, `UInt32`	0...4 294 967 295	4
`UInt64`	$0...2^{64}-1$	8

Символьный тип

Второй по счету представитель простых типов — символьный тип данных — специализируется на хранении кода символа. В современных версиях Delphi по умолчанию используется кодировка Unicode. Один символ в такой кодировке может занимать до 4 байт, что позволяет закодировать все символы национальных алфавитов. Вместе с тем, Delphi поддерживает и более старую однобайтовую кодировку ANSI (табл. 2.2).

Таблица 2.2. *Символьный тип данных*

Тип	Кодировка	Размер в байтах
`AnsiChar`	ANSI	1
`WideChar`, `Char`	Unicode	От 1 до 4

Вас не должен смущать тот факт, что символьный тип отнесен к порядковому, а значит, и к целочисленному типу данных. Дело в том, что каждому символу сопоставлен его числовой код. В этом вы сможете убедиться, повторив небольшой пример, который представлен в листинге 2.6.

Листинг 2.6. Демонстрация типа данных `Char`

```
program typechr;
{$APPTYPE CONSOLE}

uses System.SysUtils;
var c:Byte;
begin
  for c:=33 to high(Byte) do Write(c,'=',Char(c),#9);
  ReadLn(c);
end.
```

В результате выполнения кода на экране отобразится таблица (рис. 2.2), отражающая соответствие между символом (в кодировке ANSI) и его целочисленным значением. Обратите внимание, что латинскому символу "A" соответствует целое значение 65, "B" — 66, "C" — 67 и т. д.

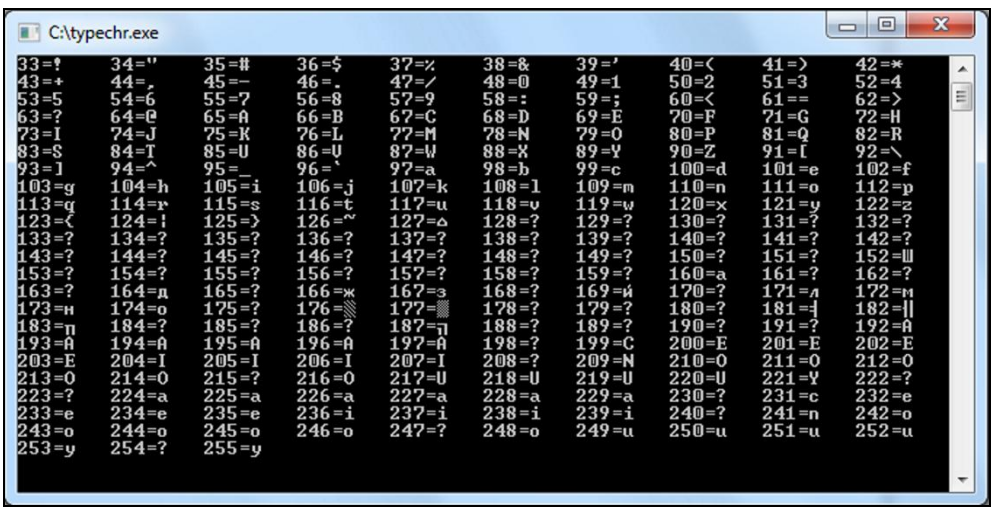


Рис. 2.2. Экранный снимок программы с таблицей символов в кодировке ANSI

Внимание!

Не так давно (до выхода Delphi 2009) тип данных `Char` соответствовал кодировке ANSI и занимал 1 байт памяти. Этот факт надо учитывать при переносе проектов, написанных на устаревших версиях языка.

Логический тип

Логический (булев) тип данных необходим для хранения данных, способных принимать всего-навсего два значения: 1 (`true`/истина) и 0 (`false`/ложь). Несмотря на многообразие

булевых типов (табл. 2.3), в подавляющем большинстве проектов достаточно услуг типа `boolean`.

Таблица 2.3. *Логический тип данных*

Тип	Диапазон значений	Размер в байтах
`boolean`	0 — `false`; 1 — `true`	1
`ByteBool`	От 0 до 255, где 0 — `false`, 1...255 — `true`	1
`WordBool`	От 0 до 65 535, где 0 — `false`, 1...65 535 — `true`	2
`LongBool`	От 0 до 4 294 967 295, где 0 — `false`, 1...4 294 967 295 — `true`	4

Перечисления

При проектировании программного обеспечения очень часто возникает потребность объявить определенную переменную (поле, свойство), которая может принимать одно из предопределенных целочисленных значений. В подобной ситуации стоит задуматься над объявлением особого пользовательского типа данных — *перечисления* (enumeration).

У перечислений много достоинств, среди них: простота и наглядность конструкций; гарантия того, что переменной присвоится приемлемое значение; удобство набора и редактирования кода программы.

В качестве простейшего примера представим перечислимый тип, соответствующий названиям семи музыкальных нот (листинг 2.7).

Листинг 2.7. Перечисление музыкальных нот

```
type TNote =(C {До}, D {Ре}, E {Ми}, F {Фа}, G {Соль}, A {Ля}, B {Си});

var Note1, Note2 : TNote;
begin
  Note1 := C; //нота До
  Note2 := B;
  ...
end;
```

Основная особенность перечислимого типа в том, что каждому его элементу соответствует целое число — это порядковый номер. По умолчанию отсчет начинается с 0. При желании программист в состоянии изменить вес одного, нескольких или даже всех элементов перечисления (листинг 2.8).

Листинг 2.8. Перечисление месяцев года с изменением нумерации элементов

```
type TMonth= (jan=1,  feb=2, march=3, apr=4,  may=5,  june=6,
              july=7, aug=8, sept=9,  oct=10, nov=11, dec=12);
var  Month:TMonth;
begin
  Month:=jan; //январь
  //...
  Month:=TMonth(3); //март
end;
```

Так как каждому именованному элементу перечисления соответствует целочисленное значение, то это позволяет проводить операции сравнения между однотипными переменными.

```
if Note1<Note2 then ...
```

При необходимости перечислимый тип допускается сравнивать с целым числом, но для этого необходимо осуществлять приведение типов.

```
If Month>TMonth(10) then ... else ...
```

Диапазоны

Еще один способ ограничения области действия переменной заключается в определении диапазона доступных значений. Переменная, входящая в диапазон, может принимать значения только в пределах границ заранее заданного диапазона (листинг 2.9).

Листинг 2.9. Демонстрация диапазона чисел

```
type THundred =1..100;
var A,B,C:THundred;
begin
  A:=5;
  B:=10;
  C:=A+B;
end;
```

При попытке присвоить переменной значение вне спектра `THundred` компилятор Delphi откажется иметь с нами дело.

Так как символьный тип данных принадлежит когорте простых типов, допускается определять диапазон символов (листинг 2.10).

Листинг 2.10. Демонстрация диапазона символов

```
type TCharRange= 'A'..'Z';
var CR: TCharRange;
begin
  CR:= 'F';
end;
```

Обслуживание данных порядкового типа

В табл. 2.4 представлен ряд базовых функций Delphi, существенно упрощающих работу программиста с данными порядкового типа.

Таблица 2.4. *Функции/процедуры, применяемые с порядковыми типами*

Функция/процедура	Описание
`function Ord(X): Longint;`	Возврат порядкового значения `X`
`function Odd(X: Longint): Boolean;`	Если `X` — нечетное число, то `true`, иначе `false`

***Таблица 2.4** (окончание)*

Функция/процедура	Описание
`function Succ(X);`	Следующее по порядку значение `X`
`function Pred(X);`	Предыдущее значение `X`
`procedure Inc (var X [; N: Longint]);`	Приращение `X` на `N`, аналог `X:=X+N`
`procedure Dec (var X[; N: Longint]);`	Уменьшение `X` на `N`, аналог `X:=X-N`
`function Low(X);`	Минимальное порядковое значение `X`
`function High (X);`	Максимальное порядковое значение `X`

Для всех порядковых типов допустима операция явного *определения (приведения) типа*. Ее смысл заключается в преобразовании одного значения перечислимого типа к другому подходящему типу. Для этого преобразуемое значение (или хранящую это значение переменную) заключают в круглые скобки, а перед ними ставят название типа, к которому мы намерены привести переменную (листинг 2.11).

Листинг 2.11. Пример приведения типов с порядковым типом данных

```
type TWeekDay =(Mon, Tu, We, Th, Fr, Sa, Su);
var  WD:TWeekDay;
     B:Byte;
     C:AnsiChar;
begin
  WD:=TWeekDay(3); //аналог явного присвоения WD:=Th;
  B:=Byte('A');    //в переменной B окажется числовой код символа 'A'
  C:=AnsiChar(B);  //в переменной C окажется символ 'A'
end;
```

Перед проведением операции приведения типов необходимо убедиться в соответствии диапазонов преобразуемых значений. Допустим, что в нашем распоряжении имеется переменная `B` типа `Byte`. Как вы думаете, какое значение окажется в этой переменной, если мы попросим Delphi выполнить операцию `B:=Byte(256)`?

Язык Delphi не станет возражать, если в коде программы встретится попытка передать в переменную типа `Byte` значение, не входящее в диапазон от 0 до 255, но при условии, что это будет операция приведения типа. Однако переменная `B` физически неспособна хранить значение 256, поэтому вместо 256 в ней окажется 0. Если же мы попытаемся выполнить операцию `B:=Byte(257)`, то обнаружим в `B` единицу, и т. д.

Действительные типы

Действительный тип данных специализируется на обслуживании вещественных чисел, т. е. чисел, имеющих как целую, так и дробную часть (табл. 2.5). При обработке чисел с плавающей точкой будьте готовы к незначительным ошибкам округления, т. к. действительные типы данных имеют предел точности. Наибольшей точностью отличается тип `Extended`, а наименьшей — `Single`.

Таблица 2.5. *Действительные типы данных*

Тип	Диапазон значений	Количество знаков	Размер в байтах
`Real48`	2.9×10^{-39} ... 1.7×10^{38}	11—12	6
`Single`	1.5×10^{-45} ... 3.4×10^{38}	7—8	4
`Double`, `Real`	5.0×10^{-324} ... 1.7×10^{308}	15—16	8
`Extended`	3.6×10^{-4951} ... 1.1×10^{4932}	19—20	10
`Comp`	$-2^{63}+1$ ... $2^{63}-1$	19—20	8
`Currency`	–922 337 203 685 477.5808... 922 337 203 685 477.5807	19—20	8

ВНИМАНИЕ!

Тип данных `Currency` предназначен специально для обработки денежных величин. Значение, переданное в переменную этого типа данных, хранится в памяти в формате `INT64`. При этом Delphi полагает, что четыре последних знака этого значения — знаки после десятичной точки. Таким образом, действительное число обрабатывается как целое, но при выводе на экран и при проведении арифметических операций Delphi делит значение на 10 000, тем самым соблюдая статус-кво.

Строковый тип

Строковый тип данных предназначен для хранения последовательностей символов (табл. 2.6).

Таблица 2.6. *Строковый тип данных*

Тип	Максимальная длина	Размер строки в памяти
`ShortString`	255 8-битовых символов	2 ... 256 байт
`AnsiString`	Около 2^{31} 8-битовых символов	4 байта ... 2 Гбайт
`WideString`	Около 2^{30} 16-битовых символов	4 байта ... 2 Гбайт
`UnicodeString`, `String`	Около 2^{30} символов в формате UTF-16	4 байта ... 2 Гбайт

Начиная с Delphi 2009, доминирующим способом представления текста стал символ в кодировке Unicode. Поэтому, объявляя строковую переменную

```
var S:string;
```

в современных версиях Delphi 2009/2010/XE/XE2, вы по умолчанию создаете переменную в формате `UnicodeString`, которая, в свою очередь, является строкой UTF-16.

Различают несколько способов представления символов в формате Unicode, основные из них — UTF-8, UTF-16 и UTF-32[1]. Самая экономная кодировка — UTF-8, т. к. в ней символ может занимать от 1 до 4 байт памяти. Здесь один байт отведен для хранения символов латинского алфавита и других наиболее распространенных символов. Благодаря экономичности кодировка UTF-8 получила широкое распространение в сетевых приложениях, в частности в языках разметки XML и HTML. Кодировка UTF-16 для хранения отдельного сим-

[1] UTF — Unicode Transformation Format.

вола может запросить от 2 до 4 байт. Наибольшей расточительностью отличается UTF-32, здесь для любого символа задействуются 4 байта.

Зачастую в целях экономного использования памяти для хранения текстовых данных программисты используют короткие строки с заранее указанным предельным значением. При объявлении переменных такого типа в квадратных скобках определяется длина строки (листинг 2.12).

Листинг 2.12. Конкретизация числа символов в строке типа `ShortString`

```
var ss:string[20];
```

Если вам по каким-то причинам необходима помощь строки ANSI с 8-битным символом, то такую строку следует объявлять, как `AnsiString`.

```
var AnsiS:AnsiString;
```

При объявлении ANSI-строк допускается конкретизировать используемую кодовую страницу, так можно легко русифицировать консольные приложения (листинг 2.13).

Листинг 2.13. Уточнение кодовой страницы при объявлении строк ANSI

```
type String866 = type AnsiString(866);
var  S:String866;
begin
  S:='Вывод надписи на русском языке';
  WriteLn(S);
  ReadLn;
end.
```

> ***Замечание***
>
> Стоит отметить, что в Delphi 2009/2010/XE если бы мы не воспользовались кодовой страницей 866 и объявили строковую переменную, как обычную строку `AnsiString` или `UnicodeString`, то в окне консоли вместо текста "Привет, мир!" отобразился бы не поддающийся чтению набор символов. В Delphi XE2 эта проблема снята.

Указатели

С точки зрения хранения данных любая переменная или константа представляют собой область памяти, в которую занесено определенное значение. Например, при объявлении целочисленной переменной `X` типа `integer` в памяти компьютера резервируется 4 байта для хранения значения этой переменной. Для обращения к содержимому переменной можно пойти традиционным путем, просто вызвав переменную по ее имени. Однако существует еще один способ, позволяющий читать и записывать данные непосредственно в область памяти. Для этого предназначены указатели.

Указатель представляет собой адрес определенной области памяти. Повторяюсь еще раз, т. к. это очень важно — указатель хранит *не содержимое* памяти, а *адрес к ячейкам памяти*, в которых размещены данные. Поэтому указатель не занимает никакого места, кроме того, которое необходимо для хранящегося в нем адреса. Таким образом, в 32-разрядных системах указатель затребует всего 4 байта памяти, а в 64-разрядных — 8 байт.

Чтобы сразу уяснить порядок работы с указателем, достаточно повторить листинг 2.14.

Листинг 2.14. Пример работы с типизированным указателем в 32-разрядной Windows

```
var X:Integer=10;//переменная
    P:^Integer;  //указатель
begin
  WriteLn('P=h ',IntToHex(Longword(P),8)); {указатель не определен,
                                            он пока никуда не ссылается}
  P:=@X;  //перевели указатель на область памяти переменной X
  WriteLn('X=',X);
  WriteLn('P=h ',IntToHex(Longword(P),8)); //адрес памяти
  X:=156; //изменим X
  WriteLn('P=h ',IntToHex(Longword(P),8)); //адрес памяти не изменился
  WriteLn('P^=X=',P^); //посмотрим, что хранится в памяти
  ReadLn;
end.
```

В нашем распоряжении имеются переменная `X`, в которой хранится число 10, и указатель `P` на эту переменную. Обратите внимание, что при объявлении указателя мы воспользовались знаком вставки `^` и следующим за ним обозначением типа данных `Integer`. На языке программистов это означает, что мы объявили типизированный указатель `P`, который хранит в себе ссылку на 4 байта памяти (напомню, что размер типа данных `Integer` равен 4 байтам).

Работу листинга 2.14 иллюстрирует рис. 2.3. Первая команда `WriteLn()` выводит на экран в шестнадцатеричном формате адрес ячейки памяти, на которую в данный момент ссылается указатель. Но пока указатель *не определен* (пуст) — адрес ячейки `h00000000`.

> ***Внимание!***
>
> Если указатель пуст, то он возвращает особое значение `nil`. Пустой указатель называют *неопределенным*.

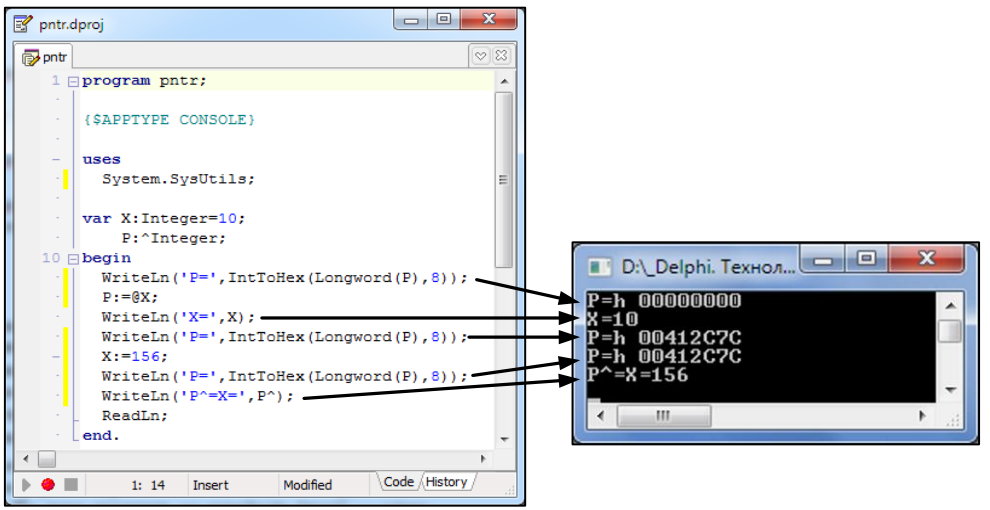


Рис. 2.3. Демонстрация работы с указателем

Для того чтобы указатель начал ссылаться на область памяти, в которой размещена переменная X, мы воспользовались услугами оператора *присвоения адреса* @.

Вновь проверим адрес указателя, на этот раз он показывает на ячейку h00412C7C[1]. Именно по этому адресу и "проживает" переменная X, в которой на данный момент хранится число 10. Изменив значение 10 на любое другое (например, 156), мы вновь проверяем адрес указателя — как и положено, он остался прежним.

Обязательно обратите внимание на последнюю команду WriteLn(). С ее помощью мы пытаемся прочитать содержимое ячеек памяти, на которые ссылается указатель P. Здесь вновь задействован оператор ^, но на этот раз он расположен после имени указателя.

Эквивалентом оператора @ служит функция

```
function Addr(X): Pointer;
```

Она возвращает указатель на расположенный в памяти компьютера объект X .

Кроме типизированных указателей, в языке Delphi существуют *нетипизированные* указатели, они описываются типом Pointer. Как и их типизированные "коллеги", нетипизированные указатели хранят адрес ячейки памяти, начиная с которой в памяти компьютера располагается переменная (или более сложная структура), однако они не владеют информацией о том, какой объем памяти задействован под хранение переменной.

Вариантный тип

Вариантный тип является универсальным типом данных. В основном он предназначен для работы с результатами заранее неизвестного типа. Но за универсальность приходится платить. Расходы заключаются в дополнительном захвате под переменную еще 2 байт памяти (это не считая байтов, необходимых для хранения обычной типизированной переменной).

Ни один из классических типов данных Delphi не позволит таких вольностей, как Variant. В приведенном примере (листинг 2.15) переменной V по очереди присваиваются значения различного типа.

Листинг 2.15. Вариантный тип данных

```
var V: variant;
    I: integer;
    S: string;
    R: real;
begin
  I:=1; S:='Привет, мир!'; R:=1.987;
  V:=I; V:=S;              V:=R;
end;
```

Вариантный тип переменной полезен при вызове объектов, поддерживающих технологию автоматизации, при обращении к программам сторонних производителей, при создании вариантных массивов. Обо всем этом мы поговорим в других главах книги. Вместе с тем не стоит злоупотреблять возможностями Variant: если программисту заранее известен подлежащий обработке тип данных, то в таких программах вариантный тип применять не стоит, т. к. он "поглощает" слишком много памяти.

[1] Если вы повторите эксперимент на своем компьютере, то адрес окажется другим.

ГЛАВА 3

Структурные типы

Если бы мы с вами захотели построить уютный дом, то работа над мечтой началась бы не с рытья котлована, а с обращения к архитектору, который начертит для нас проект будущего дома. Если мы с вами захотим собрать новый сверхзвуковой самолет, то нашим первым шагом станет поиск авиаконструктора, который предложит эскизы будущего лайнера. Любой раздел науки и техники предполагает, что первым этапом решения задачи должен стать чертеж, математическое описание, химическая формула или другой, принятый в этой области знаний, способ моделирования. Информационные технологии и самая их сложная отрасль разработки программного обеспечения не являются исключением из этого правила — процесс создания программы начинается с проектирования структур, предназначенных для хранения данных.

Все программы обрабатывают данные. Все без исключения. Даже самая тривиальная программка, специализирующаяся на приветствии пользователя фразой "Hello, world!", нуждается в помощи структур. Для хранения текста приветствия нам придется воспользоваться строковым типом данных `AnsiString` или массивом символов `Char`, или массивом байтов `Byte`. Не исключено, что особенности эксплуатации программы "Hello, world!" натолкнут разработчика на иные решения. Вполне вероятно, что вы предпочитаете кодировку Unicode, хранящую символ не в одном, а в 2 (3, 4) байтах памяти, тогда вы воспользуетесь типом данных `String`.

> ***Внимание!***
>
> Начиная с 2009 года, пакет Embarcadero RAD Studio был переведен в режим поддержки кодировки Unicode по умолчанию. Это решение повлекло радикальное изменение таких широко используемых типов данных, как `Char`, `pChar` и `String`. С этого момента символ `Char` стал работать в кодировке UTF, указатель `pChar` превратился в указатель на символ Unicode, строка `String` стала `UnicodeString`.

Для хранения и обработки данных каждой из программ требуются помощники: в простейшем случае — переменные и константы, а при проектировании сложных приложений — структуры. От умения программиста подобрать наиболее подходящую структуру зависит половина успеха.

Различают следующие основные разновидности структурных типов:

- множества;
- записи;
- массивы;

- файлы;
- классы;
- указатели на классы.

Первые три типа будут рассмотрены в этой главе, а файлы, классы и указатели на классы несколько позже.

Множества

Множество представляет собой коллекцию однотипных значений, в состав которой могут одновременно входить все значения, часть значений или ни одного значения (пустое множество). Язык Delphi предлагает несколько способов объявления множеств (листинг 3.1).

Листинг 3.1. Способы объявления множеств

```
type TWeekDaysSet = set of (Mo, Tu, We, Th, Fr, St, Su);
type TByteSet = set of Byte; //множество от 0 до 255
type TTeenSet= set of 1..10; //множество от 1 до 10

type TMyRange=60..100; //пока это не множество, а диапазон
type TMyRangeSet= set of TMyRange; //множество от 60 до 100
```

> ***Внимание!***
>
> Элементы множества представляют собой целые числа и не могут принимать значения меньше нуля.

Наиболее нетерпеливый читатель спросит, так чем же отличаются множества от рассмотренных в предыдущей главе перечислений и диапазонов? Постараюсь объяснить на небольшом примере. Представьте себе, что вы владелец замка, в котором есть три башни. Каждое утро вы взбираетесь на одну из башен, а может быть (если у вас хорошее настроение и нет дождя), вы посещаете две или даже три башни и поднимаете над каждой из них флаг. На правах барона вы можете выбрать любую понравившуюся комбинацию флагов над башнями замка. Если ассоциировать три флага с ячейками памяти компьютера, то это всего-навсего 3 бита. Единичка в ячейке свидетельствует о поднятии, нолик — об опускании соответствующего флага. Если множество содержит всего три элемента, то общее количество возможных комбинаций составляет $2^3 = 8$. Зарезервированное слово `Set` способно определять множество размером до 256 элементов. Возведите 2 в степень 256 и получите $1{,}15792089237316195423570985008687 \times 10^{77}$ вариантов. На практике такое количество вариаций вряд ли когда понадобится, поэтому старайтесь объявлять множества разумного размера.

А теперь предложим несколько строк кода (листинг 3.2), демонстрирующих работу с множествами.

Листинг 3.2. Управление элементами множества

```
type TFlagsSet = set of (Flag1, Flag2, Flag3);
...
var Flags : TFlagsSet;
```

```
begin
  Flags:=[];                //[0,0,0] — полностью очистили множество
  Flags:=[TFlagsSet.Flag1]; //[1,0,0] — включили 1-й элемент
  Flags:=Flags+[TFlagsSet.Flag3]; //[1,0,1]- добавили 3-й элемент
  Flags:=Flags-[TFlagsSet.Flag1]; //[0,0,1]- отключили 1-й элемент
  Flags:=[TFlagsSet.Flag1, TFlagsSet.Flag2, TFlagsSet.Flag3];
                      {[1,1,1] — включили все три элемента}
  ...
end;
```

Для проверки, включен ли элемент во множество, применяют оператор `in`.

```
If (Flag2 in Flags) then <операция 1> else <операция 2>;
```

Множества Delphi поддерживают все стандартные операции, применимые к математическим множествам. В первую очередь это операции сложения и вычитания.

```
[Flag1, Flag2]+[Flag3] = [Flag1, Flag2, Flag3]
[Flag1, Flag2, Flag3]-[Flag2, Flag3] = [Flag1]
```

Еще одна очень важная для множеств операция пересечения позволяет вычислить общие для двух множеств элементы:

```
[Flag1, Flag2]*[ Flag2, Flag3] = [Flag2]
```

Если определены два однотипных множества, то между ними вполне допустимы операции сравнения (<=, >=, =, <>). В операциях сравнения множество `X` меньше или равно множеству `Y` (выражение `(X<=Y)=True`), если каждый элемент множества `X` является членом множества `Y`. Множество `X` равно множеству `Y` (выражение `(X=Y)=True`) в случае, если все элементы множества `X` точно соответствуют элементам `Y`. Множество `X` неравно `Y` (выражение `(X<>Y)=True`), если хотя бы один элемент множества `X` отсутствует во множестве `Y`.

Записи

Запись (record) позволяет объединять несколько разнотипных полей в общую структуру. Идея построения составных структур появилась еще во времена языка Pascal и сразу очень понравилась разработчикам. В простейшей нотации запись современного языка Delphi ничем не отличается от аналогичной структуры старого доброго языка Pascal. Листинг 3.3 демонстрирует объявление записи, которое будет поддержано во всех, базирующихся на языке Pascal средах разработки.

Листинг 3.3. Пример работы с записью

```
type TDemoRecord=record //описание типа записи
  A: Byte;              //поля записи
  B: Integer;
  C: Word;
  D: Integer;
end;

var DemoRecord:TDemoRecord; //объявление переменной
begin
  {присвоение полям записи значений}
```

```
  DemoRecord.A:=50;
  DemoRecord.B:=0;
end.
```

Вроде бы все просто и нет никаких подводных камней. А теперь проверим наши знания арифметики. Сколько на ваш взгляд займет места в памяти компьютера переменная `DemoRecord:TDemoRecord` из листинга 3.1? Если кто-то подзабыл, то напомню: для хранения переменной типа `Byte` необходим 1 байт памяти, для `Integer` — 4 байта, для `Word` — 2 байта. Попробуем решить задачку так, как нас учили в школе: 1 (`A`) + 4 (`B`) + 2 (`C`) + 4 (`D`) = 11 байт.

Все согласны? Оказывается, нет. Первый, кто откажется утвердить наши подсчеты, будет компьютер с Delphi "на борту". С помощью функции `SizeOf()` проверим, сколько байтов насчитал наш кремневый друг:

```
WriteLn('TDemoRecord=',SizeOf(TDemoRecord));//размер=16 байт
```

Мы получим результат, равный 16 байтам! Неординарный подход к подсчету суммы, не правда ли? Однако, это еще не все. Как вы думаете, изменяется ли сумма от перестановки мест слагаемых? Не спешите отвечать, а просто откорректируйте очередность объявления полей в структуре и вновь воспользуйтесь услугами функции `SizeOf()` так, как это предложено в листинге 3.4.

Листинг 3.4. Изменение порядка следования полей

```
type TDemoRecord2=record
  A : Byte;
  C : Word;
  B,D : Integer;
end;
begin
  WriteLn('TDemoRecord2=', SizeOf(TDemoRecord2));//размер=12 байт
  ReadLn;
end.
```

Как видите, чудеса арифметики не закончились. Стоило нам поменять местами поля `B` и `C`, и размер записи удивительным образом уменьшился на 4 байта и составил 12 байт. Несмотря на то, что новый размер структуры оказался существенно ближе к расчетному, он по-прежнему подозрителен. Однако не спешите бежать в магазин, чтобы заменить "неисправный" процессор, его расчеты не противоречат здравой логике. Подтверждение этому мы обнаружим на рис. 3.1.

Чудеса компьютерной арифметики легко объяснимы. При размещении данных в памяти в 32-разрядных системах поля записей и других структур автоматически *выравниваются* по 32-битовой границе. Если в качестве примера взять первую версию записи (левая часть рис. 3.1), то вы увидите, что к занимающему 1 байт первому полю `A` записи `TDemoRecord` добавились 3 незначащих байта, а к полю `C` присоединились 2 пустых байта. Так и накопились 5 незадействованных байт. Во второй версии записи (правая часть рис. 3.1) благодаря оптимизации очередности полей поля `A` и `C` с удобствами разместились в рамках первого 32-битного участка. Как следствие — число пустых байтов существенно сократилось. Мы создали более совершенную структуру.

```
type
TDemoRecord=record
  A:Byte;
  B:Integer;
  C:Word;
  D:Integer;
end;
```

```
type
TDemoRecord2=record
  A:Byte;
  C:Word;
  B,D:Integer;
end;
```

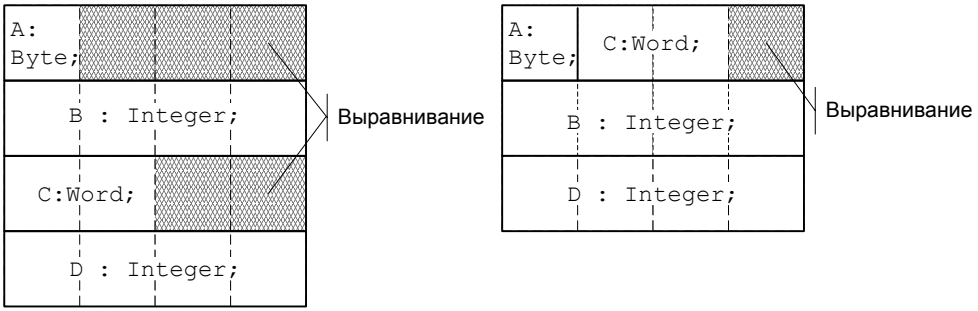


Рис. 3.1. Выравнивание данных по 32-битовой границе

ВНИМАНИЕ!

В Delphi все 8-байтные значения (например, тип данных `double`) выравниваются по 64-битовой границе. Такое решение позволяет ускорить операции с плавающей точкой, но не всегда рационально задействует память.

Язык Delphi позволяет создавать *упакованные записи*, т. е. записи, в которых по возможности отсутствует выравнивание полей. Режим упаковки активируется у записей, помеченных инструкцией `packed` (листинг 3.5).

Листинг 3.5. Упаковка записи

```
type TDemoRecord3 = packed record
  B,D: Integer;
  C: Word;
  A: Byte;
end;
```

Если провести анализ нового варианта структуры с помощью функции `SizeOf()`, то мы, наконец, получим так долго ожидаемый размер в 11 байт. Однако достигнутая экономия при хранении данных имеет обратную сторону — несколько увеличивается время на доступ к полям структуры, особенно если они начинаются не с начала 32-битовой области и переходят через границу этой области. Поэтому при проектировании упакованных записей рекомендуется сначала размещать поля, размер которых кратен 4 байтам (например: `Integer` и `Cardinal`), а поля остальных типов смещать к концу структуры.

Вариантные поля

В реальной жизни один и тот же объект может иметь несколько способов описания. Одна и та же точка на поверхности земного шара идентифицируется географическими или прямоугольными координатами, вес предмета может быть измерен в граммах или фунтах, расстояние до объекта можно указать в километрах или в милях.

Создатели языка Delphi позаботились о том, чтобы используемые в наших программах структуры обладали максимальной гибкостью. Именно поэтому в состав полей записи допускается включать вариантное поле (листинг 3.6).

Листинг 3.6. Запись с вариантным полем

```
type TDimensionMode=(kilometer,mile);

type TDistance=record
   ObjectName:string[30];
   case DimensionMode:TDimensionMode of
     kilometer : (km :real);
     mile      : (ml :real);
   end;
```

Запись `TDistance` позволяет сохранить расстояние до объекта в километрах или милях. Для того чтобы указать, к какому из вариантов относится вводимое значение, надо воспользоваться *полем признака* `DimensionMode` и передать в него значение `kilometer` или `mile`.

Усовершенствованная запись

Современные версии Delphi способны превратить обычную запись (всю жизнь до этого специализирующуюся лишь на хранении данных) в комплексную структуру с набором возможностей, очень близким к функционалу класса. Как следствие, "простушка" запись из обычного хранилища разнотипных полей превращается в интеллектуальную систему обслуживания данных. Безусловно, степень разумности такой записи определяется глубиной заложенного в структуру программного кода. В листинге 3.7 предложен пример объявления записи с расширенными возможностями. Запись `TStudent` предназначена не только для хранения фамилии, имени и даты рождения студента, ко всему прочему запись способна вычислить возраст студента.

Листинг 3.7. Запись с расширенными возможностями

```
Uses SysUtils, DateUtils;
...
type TStudent=record
  private                      //секция частных объявлений
    function GetAge: Integer; //метод записи
  public                       //секция публичных объявлений
    SName,FName:string[15];    //фамилия и имя
    BDate:TDate;               //дата рождения
    property Age:integer read GetAge; //свойство записи — возраст
  end;
{ TStudent }
function TStudent.GetAge: integer; //реализация метода
begin
  {узнаем разницу между текущей датой и датой рождения}
  Result:=YearsBetween(Now, BDate);
  {для работы функции YearsBetween необходимо подключить модуль DateUtils}
end;
```

По сравнению с классическим способом описания записи наш пример (см. листинг 3.7) содержит много необычного:

- появились секции частных и публичных объявлений;
- запись приобрела право обладать методами;
- у записи могут быть свойства.

Безусловно, даже усовершенствованная запись не способна заменить класс (записи не поддерживают наследование, у них не могут быть объявлены виртуальные и динамические методы, запись не может обладать деструктором), но такая задача перед программистами Embarcadero и не ставилась. Новый синтаксис записей просто предоставляет программисту Delphi существенно улучшенный инструмент для разработки программ.

> ***Примечание***
>
> Начиная с Delphi 2009, при описании типа записи допускается задавать поля с заранее неопределенным (обобщенным) типом данных. Эту возможность мы рассмотрим в *главе 8*.

Массивы

Массивы вряд ли нуждаются в представлении для человека, имеющего даже самый небольшой опыт программирования, ведь это очень востребованная структура данных. *Массив* (array) предназначен для хранения некоторого количества однотипных элементов. Ключевое достоинство массива в том, что он обеспечивает максимально быстрый доступ к любому из своих элементов. Такая особенность массива имеет простое объяснение. Так как все элементы массива задействуют одно и то же число байтов, то они хранятся последовательно один за другим в памяти компьютера (рис. 3.2).

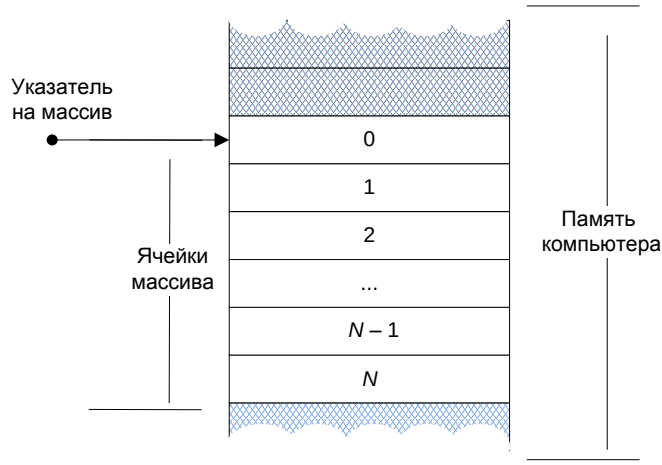


Рис. 3.2. Представление массива в памяти

Для адресации нужного элемента массива достаточно знать адрес первого элемента. Элемент с нужным индексом легко вычисляется путем элементарного умножения индекса на размер элемента в байтах, результат произведения прибавляется к адресу первой ячейки.

ВНИМАНИЕ!

В составе модуля `Generics.Collections.pas` объявлен шаблон класса `TArray`, специализирующийся на объектно-ориентированном представлении массива. Одно из основных достоинств класса в том, что в нем реализованы универсальные методы сортировки и поиска данных в массиве.

Объявление массива

Массив, как и переменную, необходимо объявить. Для этого следует указать размер массива и тип хранимых данных:

```
var <имя>: array [<нижняя граница>..<верхняя граница>] of <тип_элементов>;
```

Если известно, что в программе будет задействовано несколько одинаковых массивов, то предварительно стоит определить тип массива и затем создавать массивы на базе объявленного типа (листинг 3.8).

Листинг 3.8. Объявление двух однотипных массивов

```
type TMyArray = Array [0..9] of integer; //массив из 10 элементов
var A1, A2 : TMyArray;
```

Допускается объявлять массив-константу с одновременным заполнением данными. В предложенном в листинге 3.9 примере двенадцать ячеек массива используются для хранения количества дней в месяцах високосного года.

Листинг 3.9. Объявление массива с постоянными значениями

```
const DaysInMonth:
array [1..12] of byte = (31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
```

Наряду с одномерными, вполне реально объявлять "квадратные", "кубические" массивы и массивы более высоких размерностей. Например, объявление пары двумерных массивов предложено в листинге 3.10.

Листинг 3.10. Объявление двумерных массивов

```
var A1 : Array[0..99,0..99] of cardinal;
    A2 : Array[0..99] of Array[0..99] of cardinal;
```

Несмотря на разницу в коде объявления массивов `A1` и `A2`, оба массива идентичны — они представляют собой матрицы размером 100×100 ячеек.

Обращение к ячейкам массива

Для обращения к элементам массива с целью чтения или записи достаточно указать индекс элемента в массиве. В листинге 3.11 предложен фрагмент кода, демонстрирующий порядок заполнения одномерного массива случайными значениями.

Листинг 3.11. Заполнение массива случайными числами

```
const N=9;
var A:Array[0..N] of integer;
    i:cardinal;
begin
  Randomize;
  for i := 0 to N do
    A[i]:=Random(100);
end.
```

Мы постарались сделать код максимально универсальным. Размер массива определяется с помощью константы `N`, в нашем примере это значение 9.

Динамический массив

У всех рассмотренных ранее способов объявления массива есть один существенный недостаток — объявив в коде программы размер массива (сделав его *статическим*), мы не сможем выйти за его границы. Зачастую, во время программирования разработчику неизвестен объем подлежащих хранению данных. Что делать, если в одном случае приложению потребуется массив из 10 элементов, а в другом из 100? В таких случаях стоит воспользоваться услугами *динамических* массивов (листинг 3.12). Для этого к нашим услугам две очень полезные процедуры Delphi. Процедура `SetLength()` во время выполнения программы распределит память под массив, функция `High()` возвратит индекс старшего элемента в массиве.

Листинг 3.12. Создание динамического массива

```
type  String866 = type AnsiString(866);
const S:String866='Введи размер массива';
var A:Array of integer;  //объявление динамического массива
    N,i:cardinal;
begin
  WriteLn(S);      //запрашиваем размер массива
  ReadLn(N);       //пользователь ввел число элементов
  SetLength(A,N); //распределяем память под массив
  Randomize;
  for i := 0 to High(A) do
    A[i]:=Random(100);  //заполняем массив случайными значениями
end;
```

> ***Внимание!***
>
> В отличие от статических массивов, допускающих любой способ нумерации своих элементов, отсчет элементов динамического массива всегда начинается с нуля.

Допускается создавать многомерные динамические массивы. Объявление такого массива и распределение памяти рассмотрено в листинге 3.13.

Листинг 3.13. Двумерный динамический массив целых чисел

```
var I: array of array of Integer;
begin
  SetLength(I,10);
```

При работе с однотипными динамическими массивами наиболее эффективным способом копирования данных из одного массива в другой считается вызов функции `Copy()`. Функция позволяет копировать как массив целиком, так и только некоторую часть его элементов (листинг 3.14).

Листинг 3.14. Копирование данных динамического массива

```
var A1, A2 : array of integer;
    i:integer;
begin
  SetLength(A1,10);
  for i:=0 to High(A1) do A1[i]:=Random(100); //заполнение массива
  A2:=Copy(A1, 0,4 );  //копирование части массива
  A2:=Copy(A1);        //полное копирование
```

Заметьте, что мы не задаем размер массива `A2` явным образом, размер будет определен автоматически с вызовом метода `Copy()`.

Вариантные массивы

При организации взаимодействия между приложением Delphi и программными продуктами сторонних производителей может возникнуть необходимость поместить в универсальную переменную `Variant` данные из массива. К сожалению, несмотря на всеядность типа данных `Variant`, прямая передача в переменную такого типа элементов статического массива невозможна. Поэтому в подобной ситуации следует воспользоваться услугами вариантного массива, а еще точнее — функции, создающей такой массив:

```
function VarArrayCreate(const Bounds: array of Integer;
                        AVarType: TVarType): Variant;
```

Функция содержит два параметра: `Bounds` — подлежащий преобразованию исходный массив; `AVarType` — формат преобразования (`varInteger`, `varDouble`, `varBoolean`, `varString` и т. д.). Для подключения функции к проекту необходимо в строке `Uses` упомянуть модуль `Variants`.

Пример работы с вариантным массивом предложен в листинге 3.15.

Листинг 3.15. Создание вариантного массива

```
Uses SysUtils, Variants;

var V : variant;
    I:integer;
begin
  V:=VarArrayCreate([0,99], varInteger);//создание вариантного массива
  V[0]:=15; //запись в ячейку вариантного массива
  I:=V[1];  //чтение из ячейки массива
  ...
end.
```

В нашем примере мы создаем вариантный массив целых чисел. Кроме того, пример демонстрирует порядок обращения к элементам массива.

```
WriteLn('Index=?');        //индекс нового элемента
ReadLn(index);

N:=High(A)+1;              //узнаем количество элементов
SetLength(A,N+1);          //расширим массив на 1 элемент
for i:=N downto index do   //смещаем элементы на 1 позицию
  A[i]:=A[i-1]; //можно так: Move(A[i-1],A[i],SizeOf(integer));
  A[Index]:=X;             //вставка в массив нового элемента
end;
```

Алгоритм можно существенно ускорить, воспользовавшись процедурой `Move()`, позволяющей переместить данные за ячейки в ячейку без операции присвоения. При работе с большими массивами стоит научить программу за один прием смещать несколько элементов в памяти. Однако это небезопасное мероприятие — при проведении операций с памятью от ошибки не застрахован даже опытный программист.

Самый непреодолимый недостаток массива — его неспособность сохранить данные после выключения компьютера. Массив располагает данные в энергозависимой оперативной памяти, которая после отключения питания обнуляется.

ГЛАВА 4

Операторы и выражения

После знакомства с основами объявления переменных и констант и построения простейших структур нам предстоит перейти на очередной уровень освоения языка — научиться использовать в программах операторы и выражения Delphi.

В терминах программирования под *выражением* понимается логически законченный фрагмент исходного кода программы, предоставляющий способ получения (вычисления) некоторого значения. Простейшим примером выражения может стать строка кода `X:=Y+Z`, возвращающая результат суммирования двух значений. Предложенное выражение содержит три *операнда* (`X`, `Y` и `Z`) и два *оператора*: `:=` и `+`.

Перечень операторов входящих в состав языка Delphi весьма обширен, при классификации операторов можно выделить следующие группы:

- оператор присваивания;
- арифметические операторы;
- оператор конкатенации строк;
- логические операторы;
- операторы поразрядного сдвига;
- операторы отношения;
- операторы множеств;
- строковые операторы;
- составной оператор;
- условные операторы.

Оператор присваивания

Едва ли не самый популярный среди всех операторов Delphi — оператор присваивания нам уже хорошо знаком. Комбинация символов ":=" уже неоднократно встречалась на предыдущих страницах книги, с ее помощью мы передавали значения в переменные. Например,

```
X:=10; //присвоить переменной X значение 10
```

Благодаря оператору `:=` в переменной `X` окажется новое значение.

Арифметические операторы

Как и следует из названия, арифметические операторы необходимы для осуществления математических действий с целыми и вещественными типами данных. Помимо известных еще из курса начальной школы операторов сложения, вычитания, умножения и деления, Delphi обладает еще двумя операторами целочисленного деления (табл. 4.1).

Таблица 4.1. *Арифметические операторы Delphi*

Оператор	Операция	Входные значения	Результат операции	Пример	Результат
+	Сложение	integer, double	integer, double	X:=3+4;	7
-	Вычитание	integer, double	integer, double	X:=10-3.1;	8.9
*	Умножение	integer, double	integer, double	X:=2*3.2;	4.4;
/	Деление	integer, double	double	X:=5/2;	3.5;
div	Целочисленное деление	integer	integer	X:=5 **div** 2;	2
mod	Остаток от деления	integer	integer	X:=5 **mod** 2;	1

При объявлении участвующих в расчетах переменных следует учитывать тип данных, возвращаемый в результате выполнения того или иного оператора. Допустим, нам следует разделить число 4 на 2 (листинг 4.1).

Листинг 4.1. Операция деления возвращает вещественное число

```
var {X:integer; — неподходящий тип данных}
     X:extended;{- правильно}
begin
  X:=4/2; //результат должен быть передан в переменную вещественного типа
  WriteLn(X);
end.
```

Даже ученик начальной школы знает, что 4/2=2, другими словами, в результате деления мы получим целое число. Однако Delphi обязательно забракует код, если мы попытаемся поместить результат операции деления в целочисленную переменную, и уведомит об этом программиста сообщением о несовместимости типов.

> ***Примечание***
>
> Операторы + и - могут применяться не только для сложения и вычитания, но и для определения знака значения. Например: X:=-5.

Оператор конкатенации строк

Оператор конкатенации строк позволяет объединять две текстовые строки в одну. Для простоты запоминания еще со времен языка Pascal в качестве объединяющего оператора используется тот же самый символ, что и для сложения двух числовых величин — символ плюса + (листинг 4.2).

Листинг 4.2. Конкатенация строк

```
const S1='Hello';

var S:String='World';
    CH:Char='!';
```

```
begin
  S:=S1+', '+S+CH;
  WriteLn(S); //'Hello, World!'
  ReadLn;
end.
```

Обратите внимание, что оператор конкатенации может быть применен не только для типа данных `String`, но и к символьному типу `Char`.

Логические операторы

В табл. 4.2 представлены четыре логических (булевых) оператора, осуществляющих операции логического отрицания, логического "И", логического "ИЛИ" и исключающего "ИЛИ". В результате выполнения любой из логических операций мы можем ожидать только одно значение из двух возможных: истина (`true`) или ложь (`false`).

Таблица 4.2. *Логические операторы Delphi*

Оператор	Операция	Пример	Результат
NOT	Логическое отрицание	`var x: boolean = NOT true;`	false
		`x:= NOT false;`	true
AND	Логическое умножение (конъюнкция, логическое "И") для двух выражений	`x:=true AND true;`	true
		`x:=true and false;`	false
		`x:=false AND true;`	false
		`x:=false AND false;`	false
OR	Выполняет операцию логического "ИЛИ" (сложения) для двух выражений	`x:=true OR true;`	true
		`x:=true OR false;`	false
		`x:=false OR true;`	true
		`x:=false OR false;`	false
XOR	Выполняет операцию исключающего "ИЛИ" для двух выражений	`x:=true XOR true;`	false
		`x:=true XOR false;`	true
		`x:=false XOR true;`	true
		`x:=false XOR false;`	false

Логические операции разрешено проводить с целыми числами. Если вы не забыли порядок перевода чисел из десятичной системы счисления в двоичную и наоборот, то вам наверняка покажется интересным листинг 4.3.

Листинг 4.3. Логические операции с целыми числами

```
var X,Y,Z : byte;
begin
{******* логическое умножение *******}
X:=5;       //в бинарном виде 0101
Y:=3;       //в бинарном виде 0011
```

```
Z:=X AND Y; // 0101 AND 0011 = 0001 {десятичное 1}
WriteLn(X,' AND ',Y,'=',Z);
{******* логическое сложение *******}
X:=1;      //в бинарном виде 0001
Y:=2;      //в бинарном виде 0010
Z:=X OR Y; // 0001 OR 0010 = 0011 {десятичное 3}
WriteLn(X,' OR ',Y,'=',Z);
{******* исключение или *******}
X:=5;       //в бинарном виде 0101
Y:=3;       //в бинарном виде 0011
Z:=X XOR Y; // 0101 XOR 0011 = 0110 {десятичное 6}
WriteLn(X,' XOR ',Y,'=',Z);
{******* отрицание *******}
X:=1;     //в бинарном виде 00000001
Z:=NOT X; // NOT 00000001 = 11111110 {десятичное 254}
WriteLn('NOT ',X,'=',Z);
ReadLn;
end.
```

Операторы поразрядного сдвига

С отдельными битами значения способны работать операторы поразрядного сдвига `SHL` и `SHR`. Первый из операторов осуществляет сдвиг влево (после оператора указывается количество разрядов сдвига), второй — поразрядный сдвиг вправо.

В листинге 4.4 представлен фрагмент кода, демонстрирующий возможности операторов сдвига.

Листинг 4.4. Поразрядный сдвиг влево

```
var i:integer=1;
begin
  while true do
  begin
    i:=i SHL 1; //сдвиг влево на 1 разряд
    WriteLn(i);
    if i>=1024 then break; //выход из цикла
  end;
readln;
end.
```

Если вы запустите программу на выполнение, то получите весьма нетривиальный результат. Оператор сдвига обрабатывает исходное значение, равное 1 (в двоичном представлении `00000001`). Каждый шаг цикла сдвигает единичку на одну позицию влево. В итоге на экране компьютера отобразится следующий столбик значений:

```
2 {что в двоичном представлении соответствует 00000010}
4                                          {00000100}
8                                          {00001000}
```

```
16                                    {00010000}
32                                    {00100000}
64                                    {01000000}
```

и т. д.

По сути, мы с вами реализовали программу, позволяющую возводить значение 2 в заданную степень.

Можете провести эксперимент, приводящий к прямо противоположному результату. Для этого проинициализируйте переменную `i` значением 2 в степени *N* (например, $2^{10} = 1024$), замените оператор `SHL` на `SHR` и перепишите условие выхода из цикла: `if i<1 then break`.

Операторы отношения

Операторы отношения (неравенства) обычно применяются для сравнения двух числовых значений, в результате сравнения возвращаются логические значения `true`/`false` (табл. 4.3). Операторы отношения — желанные гости в условных операторах.

Таблица 4.3. *Операторы отношения*

Оператор	Операция	Пример	Результат
=	Сравнение	10=5	false
<>	Неравенство	10<>5	true
>	Больше чем	10>5	true
<	Меньше чем	10<5	false
>=	Больше или равно	10>=5	true
<=	Меньше или равно	10<=5	false

Примечание

Операторы отношения допустимо использовать не только с числовыми, но и с текстовыми данными.

Операторы множеств

В составе Delphi предусмотрен ряд операторов, позволяющих осуществлять операции над однотипными множествами (табл. 4.4).

Таблица 4.4. *Операторы множеств*

Оператор	Операция	Результат
+	Объединение	Новое множество
-	Разность	
*	Пересечение	

Таблица 4.4 *(окончание)*

Оператор	Операция	Результат
<=	Подмножество	Логическое значение
>=	Надмножество	
=	Равенство	
<>	Неравенство	
in	Членство	

Порядок осуществления операций с множествами демонстрирует листинг 4.5.

Листинг 4.5. Демонстрация операций с множествами

```
type TWeekDay = set of (Mo, Tu, We, Th, Fr, St, Su);

var WD1,WD2,WD3: TWeekDay;
begin
  WD1:=[Mo];         //присваивание,     результат WD1=[Mo]
  WD2:=WD1+[Tu,We]; //объединение,       результат WD2=[Mo, Tu, We]
  WD3:=WD2-WD1-[Tu];//вычитание,         результат WD3=[We]
  WD3:=[];           //пустое множество, результат WD3=[]
  WD1:=[Mo,Su];
                     //пересечение
  WD3:=WD1*[Mo, Tu, We, Th, Fr, St, Su]; //результат WD3=[Mo, Su]
  if WD1<>WD2 then begin//проверка неравенства
                     //...
                   end;
  if Mo in WD1 then begin//проверка членства
                     //...
                    end;
end.
```

Составной оператор *begin..end*

Вне зависимости от назначения разрабатываемого приложения программисту не обойтись без составного оператора `begin..end`. Задача оператора — указание начала и конца программного блока. Характерным примером подобного блока является создаваемый по умолчанию каркас кода консольного приложения, здесь весь исполняемый код заносится вовнутрь `begin..end`.

Выражения, заключенные в рамки составного оператора, могут рассматриваться компилятором как составной оператор, подлежащий выполнению как единое целое. С одним из примеров использования составного оператора вы столкнетесь буквально на следующей странице (см. листинг 4.7).

Условный оператор *if..then..else*

Условные операторы позволяют осуществлять выбор операции во время выполнения программы. В простейшем случае синтаксис команды выглядит следующим образом:

```
if <логическое выражение = true> then <выражение>;
```

Операция подлежит выполнению только при условии, что логическое выражение истинно (возвращает `true`). Предусмотрен еще один, расширенный вариант применения условного оператора:

```
if <логическое выражение = true> then <выражение1> else <выражение2>;
```

В последнем случае мы осуществляем выбор между двумя выражениями. При соблюдении условия выполняется первая операция, в противном случае управление передается второй операции. Оба варианта применения условного оператора `if` представлены на рис. 4.1 в виде фрагментов блок-схем.

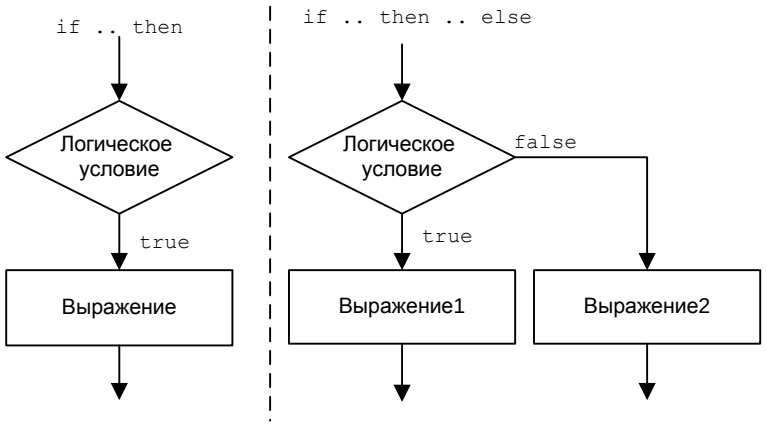


Рис. 4.1. Фрагмент блок-схемы `if..then` и `if..then..else`

Простейший пример работы с условным оператором представлен в листинге 4.6. Если `A` превышает 15, то в переменную `B` передается произведение переменной `A` на `15`, в противном случае результатом окажется значение `A-3`.

Листинг 4.6. Демонстрация оператора `if..then..else`

```
If A>15 then B:=A*15
        else B:=A-2;
```

В листинге 4.7 предложен более сложный фрагмент кода, демонстрирующий порядок работы с условным оператором. В нашем примере мы осуществляем расчет площади квадрата или круга.

Листинг 4.7. Совместная работа `if..then..else` с оператором `begin..end`

```
var S:Char;
    X,Y:Double;
```

```
begin
  WriteLn('Calculation area: "S" — square, "C" — circle');
  ReadLn(S); //пользователь вводит символ "S" или "C"
  if S='S' then
     begin
       WriteLn('The side of square=?');
       ReadLn(X);
       Y:=X*X;
       WriteLn('Area=',Y);
     end else
  if s='C' then
     begin
       WriteLn('Radius=?');
       ReadLn(X);
       Y:=pi*X*X;
       WriteLn('Area=',Y);
     end else WriteLn('Wrong symbol!');
  ReadLn;
end.
```

Выбор фигуры осуществляется пользователем нажатием определенной символьной клавиши, расчет площади квадрата инициирует символ "S", а круга — "C". Если пользователь ввел любой другой символ, то программа уведомляет его об этом надписью "Wrong symbol!" (неверный символ).

> ***Замечание***
>
> По умолчанию консольное приложение отображает вещественные значения в научном формате. Для перехода от научного представления к обычному следует воспользоваться функцией `FloatToStr()`.

Условный оператор может обслуживать достаточно сложные логические выражения, в состав которых входят булевы операторы из табл. 4.3. Один из примеров сложного условия вы обнаружите в листинге 4.8.

Листинг 4.8. Составное логическое условие в операторе `if..then`

```
if (A>15) and ((B=0) or (B=1)) then ...
```

Условный оператор даст разрешение на выполнение только в случае, когда в `A` хранится значение, превышающее 15, и в переменной `B` находится число равное 0 или 1.

Оператор-селектор *case*

Логическим развитием условного оператора `if..then` стал оператор-селектор `case`. Основное преимущество `case` над своим "коллегой" в том, что он обладает большей наглядностью, чем группа операторов `if..then`.

Оператор включает селектор и одну или несколько констант выбора. На роль селектора могут претендовать все типы данных, между которыми существуют отношения порядка (например, целочисленные `byte`, `word`, `integer`), и строковый тип данных. В качестве селектора

допустимо применять не только переменную, но и выражение или функцию, но при соблюдении единственного условия — выражение (функция) должно возвращать значение порядкового типа.

```
case <селектор> of
  <константа1> : <Выражение1>;
  <константа2> : <Выражение2>;
  <константа3> : <Выражение3>;
  else <Выражение4>;
end;
```

Оператор `case` осуществляет проверку на равенство значений селектора и констант оператора. В случае если значение селектора совпадает со значением константы, то выполняется соответствующее константе выражение (листинг 4.9).

Листинг 4.9. Пример обращения к оператору `case`

```
type String866 = type AnsiString(866);
var X,Y,Z:real;
    op:char;
begin
  WriteLn('X=');
  ReadLn(X);
  WriteLn(String866('Выбери операцию [+ или – или / или *]'));
  ReadLn(op);
  WriteLn('Y=');
  ReadLn(Y);
  Case op of
     '+': Z:=X+Y;
     '-': Z:=X-Y;
     '/': if Y<>0 then Z:=X/Y else WriteLn(String866('Ошибка!'));
     '*': Z:=X*Y;
     else WriteLn(op,String866(' – недопустимая операция!'));
  end;
  WriteLn(X,op,Y,'=',Z);
  ReadLn;
end.
```

Предложенный листинг представляет собой реализацию простейшего калькулятора, осуществляющего одну из основных математических операций над двумя значениями. Обратите внимание на то, как мы защитили программу от вероятной ошибки деления на 0: для этого мы воспользовались помощью условного оператора `if..then`, разрешающего операцию деления только при условии, если знаменатель не равен нулю.

Функционал секции `else` внутри селектора `case` аналогичен задачам оператора `else` в `if..then..else`. Если селектору не соответствует ни одна из констант, то будет выполнен оператор, следующий за словом `else`. Если же в конструкции отсутствует ключевое слово `else`, то выполнению подлежит следующая за оператором строка кода.

Примечание

Селектор `case` может использоваться при объявлении записи с вариантным полем.

Оператор перехода *goto*

В 1970-е годы, во времена становления языка Pascal, достаточно распространенным приемом программирования был стиль, основанный на применении оператора перехода `goto`. Оператор перехода позволял осуществить быстрый переход от одного участка кода к другому, отмеченному специальной меткой участку (листинг 4.10).

Листинг 4.10. Использование оператора перехода `goto`

```
label DemoLabel; //объявление метки
var i:integer=0;
begin
DemoLabel: {участок кода отмечен меткой DemoLabel}
    begin
      WriteLn(i);
      inc(i);
      if i<10 then goto DemoLabel; {переход к метке DemoLabel}
    end;
 ReadLn;
end.
```

Сегодня оператор `goto` утратил свою актуальность и практически не применяется, вместо него целесообразно отдавать предпочтение вызову процедур и функций. Единственным оправданием обращения к оператору перехода может стать принудительный выход вниз по коду из нескольких вложенных циклов.

> ***Внимание!***
>
> Оператор `goto` противоречит одному из основных принципов структурного программирования — модули программы должны иметь один вход и один выход.

Оператор *with..do*

Оператор `with..do` значительно упрощает процесс написания кода программы, обеспечивая ускоренный доступ к полям записей и объектов. Каким образом? Допустим, что наша программа использует переменную-запись вида, предложенного в листинге 4.11.

Листинг 4.11. Стандартный способ обращения к полям записи

```
var DemoRecord : record //объявление записи
     I : integer;
     R : real;
     B : boolean;
   end;
begin
  DemoRecord.I:=-41;
  DemoRecord.R:=0.48;
  DemoRecord.B:=True;
```

Как видите, для обращения к полям записи мы вынуждены писать код, повторяющий из строки в строку имя владельца полей — DemoRecord. А если таких полей несколько десятков? Никчемное и рутинное занятие. Разработчики Delphi предложили конструкцию, исключающую необходимость многократных повторов имени объекта (записи, переменной и т. д.):

```
with <объект> do
begin
  <действие с полем 1 объекта>
  . . .
  <действие с полем N объекта>
end;
```

Оператор with..do значительно упрощает труд программиста. Теперь обращение к полям записи происходит без многократного упоминания имени самой записи (листинг 4.12).

Листинг 4.12. Пример работы с оператором with..do

```
with DemoRecord do
begin
  I:=-41;
  R:=0.48;
  B:=true;
end;
```

Встретив конструкцию with..do, компилятор понимает, что далее идет речь только о конкретном объекте (в нашем случае это запись DemoRecord), и больше не требует упоминания его имени.

Организация циклов

При разработке программ зачастую возникает необходимость повторного выполнения ряда одинаковых действий, например вычислений. В подобной ситуации на помощь приходят циклы. Циклы предназначены для многократного выполнения одного или нескольких выражений. В Delphi предусмотрены три разновидности циклов:

- цикл for с параметром;
- цикл while..do с предусловием;
- цикл repeat..until с постусловием.

На рис. 4.2 представлены структурные схемы всех трех циклов.

Цикл с параметром *for..do*

Наиболее прост для понимания цикл с параметром (счетчиком). Он позволяет явным образом задать число выполняемых итераций. Синтаксическая конструкция цикла со счетчиком выглядит следующим образом:

```
for <счетчик цикла>:=<стартовое значение>
to | downto <конечное значение> do <оператор>;
```

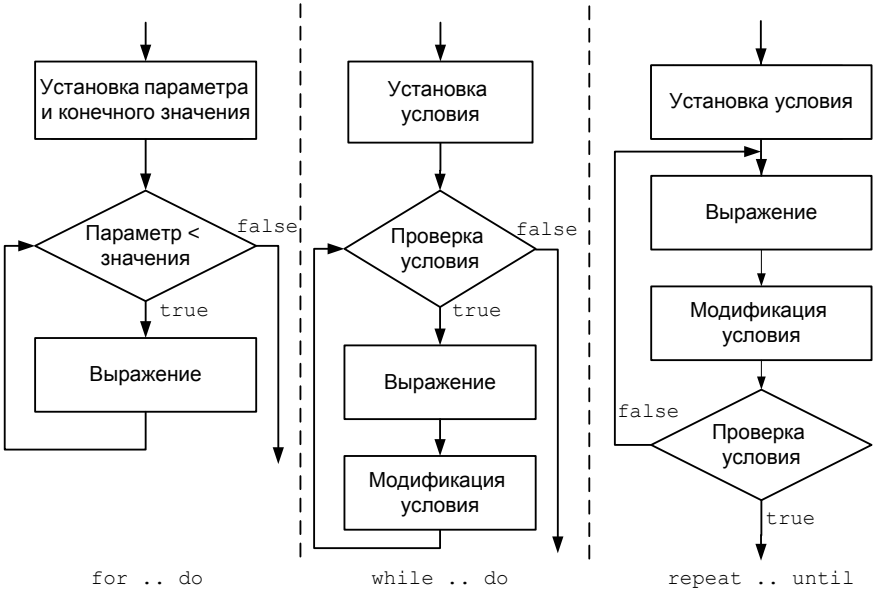


Рис. 4.2. Фрагмент блок-схемы операторов циклов

В качестве счетчика цикла может выступать любая порядковая переменная (ее обычно называют *параметром цикла*). При каждом проходе цикла переменная получает приращение (уменьшение). Цикл продолжается до тех пор, пока значение параметра не достигнет конечного значения. Пример из листинга 4.13 демонстрирует простейший способ применения цикла со счетчиком.

Листинг 4.13. Пример цикла со счетчиком

```
var i:integer;
begin
 for i:=0 to 9 do
     begin
       ...
       WriteLn(i);
     end;
end;
```

В предложенном примере работа цикла прекратится в тот момент, когда параметр `i` превысит значение 9. Таким образом, будет осуществлено 10 итераций.

Отличие ключевого слова `to` от `downto` в том, что при использовании в цикле слова `to` параметр цикла увеличивает свое значение, а во втором — уменьшает. В листинге 4.14 показано, как следует организовывать цикл с убывающим счетчиком.

Листинг 4.14. Цикл с убывающим счетчиком

```
for i:=99 downto 3 do
  begin
    ...
    WriteLn(i);
  end;
```

Внимание!

Ни в коем случае не допускается изменять значения параметра внутри тела цикла `for..do`. Ведь это не более чем счетчик числа итераций. Если же ваша интуиция и логика программы подсказывают, что необходимо найти решение, в котором параметр цикла будет выступать не только в виде банального счетчика, но и в роли управляемого параметра, то на службу вместо цикла `for..do` надо поставить цикл `while..do` или `repeat..until`.

Инструкция *in* в цикле *for..do*

Цикл `for` имеет еще один вариант применения. Если вместо конструкции `for..to..do` задействовать `for..in..do`, то цикл найдет возможность самостоятельно разобраться с шагом счетчика. Если говорить точнее — шаг передается через дополнительный параметр, например множество, строку, коллекцию, запись или массив (листинг 4.15).

Листинг 4.15. Управление параметром цикла с помощью инструкции `in`

```
var x: array[0..7] of byte = (1,2,3,5,8,11,19,45);
    i:Integer;
begin
  for i in x do
    WriteLn(i);
  ReadLn;
end.
```

Результатом выполнения предложенного кода станет вывод на экран строк со значениями 1, 2, 3, 5, 8, 11, 19, 45.

Цикл с предусловием *while..do*

Особенность цикла с предусловием заключается в том, что код, сосредоточенный в теле цикла, будет выполняться до тех пор, пока соблюдается условие, описанное в заголовке цикла. Базовая конструкция цикла выглядит так:

```
while <(логическое выражение = true)> do <оператор>;
```

Условие цикла задается в виде булевого выражения (выражения, возвращающего значение `true`/`false`).

Листинг 4.16. Цикл с предусловием

```
var x: array [0..99] of Integer;
    i:Integer;
begin
  i:=0;
  while i<=99 do
  begin
    x[i]:=i;
    INC(i); //приращение i:=i+1;
  end;
```

Предложенный в листинге 4.16 пример решил задачу заполнения массива рядом целых чисел (от 0 до 99).

Организация цикла while..do требует от программиста большой внимательности — нам нельзя забывать управлять логическим условием цикла. Например, если бы мы с вами забыли в листинге 4.16 ввести команду INC(i), то наш цикл станет выполняться вечно (по крайней мере, до тех пор, пока не сядут аккумуляторы в вашем ноутбуке).

> ***Замечание***
>
> Основное преимущество цикла с предусловием while над своим коллегой циклом for заключается в возможности определения расширенных условий управления циклом. С помощью операторов отношения (<, <=, <>, >=,>, =) и булевых операторов (AND, OR, NOT) мы имеем возможность составлять сложные логические комбинации, что существенно расширяет спектр применения цикла while.

Цикл с постусловием *repeat..until*

Наше знакомство с циклами Delphi продолжит цикл repeat..until.

```
repeat <оператор> until <логическое выражение = false>;
```

Основные отличия цикла с постусловием repeat..until от цикла с предусловием while..do заключаются в следующем. Во-первых, оператор цикла repeat..until вне зависимости от логического выражения будет выполнен хотя бы один раз. Это объясняется местонахождением логического выражения — проверка условия происходит после того, как выполнилось тело цикла. Во-вторых, в отличие от цикла while..do выход из цикла с постусловием осуществляется при истинности логического выражения. Другими словами, цикл repeat..until будет выполняться до тех пор, пока логическое выражение не соблюдается.

Листинг 4.17 демонстрирует порядок применения цикла с постусловием.

Листинг 4.17. Цикл с постусловием

```
var ch:char;
    X,Y : integer;
begin
  repeat
    WriteLn('X=?, Y=?');
    ReadLn(X, Y);
    WriteLn(X, '+', Y, '=', x+y);
    WriteLn('Press "q" to exit');
    ReadLn(ch);
  until (ch='q');
end.
```

Разработанная нами программа суммирует два полученных от пользователя значения. Затем предлагает подтвердить выход из программы нажатием клавиши с символом "q", если будет введен любой другой символ, то цикл подсчета суммы X+Y вновь повторится.

Вложенные циклы

Одним из наиболее распространенных приемов работы с циклами является идея использования вложенных циклов, в этом случае в теле одного цикла выполняется другой. В листинге 4.18 представлен способ обнуления всех элементов двумерного массива A размером

10×10 элементов. Обращение к ячейкам массива производится в рамках двойного цикла `for..do`.

Листинг 4.18. Пример организации вложенного цикла

```
var A : Array[0..9, 0..9] of Integer;
    x, y : byte;
begin
  for x:=0 to 9 do //внешний цикл
    for y:=0 to 9 do A[x, y]:=0; //вложенный цикл
end;
```

Операторы *break* и *continue*

Изучение циклов было бы логически незавершенным без рассмотрения операторов прерывания текущей итерации цикла `break` и внеочередного начала следующей итерации `continue`.

Рассмотрим представленный в листинге 4.19 пример.

Листинг 4.19. Прерывание цикла оператором `break`

```
var Accept  : Boolean;
    Counter : Integer;
begin
  Accept:=true;
  Counter:=0;
  while Accept=true do
    begin
       Accept:=<ЛОГИЧЕСКОЕ ВЫРАЖЕНИЕ>;
       INC(Counter); //Counter:=Counter+1;
       if Counter>10 then break;
    end;
end;
```

Цикл `while..do` будет выполняться до тех пор, пока переменная `Accept` принимает значение `true`. Но количество итераций в любом случае не превысит числа 10, контроль за этим осуществляется с помощью переменной-счетчика `Counter`. При достижении `Counter` значения 11 работа цикла прекращается.

В листинге 4.20 предложен еще один пример. На этот раз нам необходимо подсчитать сумму нечетных чисел, входящих в диапазон от 0 до 99. Для этого воспользуемся оператором `continue` и операцией целочисленного деления `MOD`, вычисляющей остаток от деления.

Листинг 4.20. Переход к очередной итерации с помощью `continue`

```
var X,Sum:Integer;
begin
  Sum:=0;
  for X:=0 to 99 do
```

```
    begin
      if (X MOD 2)=0 then continue; {пропускаем итерации
                                     с четным значением x}
      Sum:=Sum+X; {накапливаем нечетные числа}
    end;
  WriteLn(Sum);
  Readln;
end;
```

Если переменная `X` принимает четное значение, то оператор `continue` принуждает цикл начать новую итерацию (досрочно увеличивает счетчик цикла на единицу). Поэтому суммируются только нечетные значения из диапазона чисел 0—99.

ГЛАВА 5

Процедуры и функции

Хорошей иллюстрацией полезности процедур и функций станет программа, в которой процедуры и функции отсутствуют... Не верите? Допустим, что к нам за помощью обратилась фирма, работающая на рынке строительных услуг. Фирма просит разработать программный продукт, позволяющий рассчитывать площадь прямоугольника, квадрата и прямоугольного треугольника. Благодаря этой программе наш заказчик намерен ускорить процесс подсчета объемов проводимых им работ (например, по покраске помещений).

Сказано — сделано! Листинг 5.1, способный рассчитать площадь трех простейших геометрических фигур, представляет собой если не шедевр программной мысли, то, по крайней мере, вполне работоспособную программу, в которой (как я и обещал заранее) отсутствуют процедуры и функции.

Листинг 5.1. Программа расчета площади фигуры

```
program ch05_01;
{$APPTYPE CONSOLE}
uses  SysUtils;

var A,B,S: real;
    F:byte;
begin
  WriteLn('Выбери фигуру: 1-квадрат; 2-прямоугольник; 3-прямоугольный треугольник');
  ReadLn(F);//выбор пользователя сохраним в переменной F
  WriteLn('Сторона A=?');
  ReadLn(A); //размер стороны A
  if (F=2) OR (F=3) then {если прямоугольник или треугольник,
                          для расчетов нужна сторона B}
    begin
      WriteLn('Сторона B=?');
      ReadLn(B);
    end;

  case F of
     1: S:=A*A;   //площадь квадрата
     2: S:=A*B;   //площадь прямоугольника
     3: S:=A*B/2; //площадь прямоугольного треугольника
```

```
    else WriteLn('Допустимы только цифры 1, 2 или 3');
  end;
  WriteLn('=',S);
  ReadLn;
end.
```

В начале работы программы пользователь выбирает номер фигуры, площадь которой необходимо получить, вводит размеры сторон и получает результат. Но так ли хорош наш код, как может показаться на первый взгляд? Не существует ли способов его усовершенствования? Давайте ненадолго отложим ответ на этот вопрос, чтобы поговорить о процедурах и функциях.

Основное назначение процедур и функций — избавление программиста от необходимости многократного написания одного и того же кода. Процедуры и функции — это не что иное, как программы в миниатюре, поэтому не так давно их называли *подпрограммами*. Структура подпрограммы сильно схожа со структурой полноценной программы на языке Delphi. У подпрограммы также могут быть свои переменные и константы, внутри подпрограммы разрешено использовать операторы и выражения. По большому счету между программой и подпрограммой всего лишь одно отличие — подпрограмма нацелена на решение небольшой подзадачи, а программа мыслит глобально и отвечает за задачу в целом. Как только для достижения глобальной цели программе понадобится решить частную подзадачу, она обратится к отвечающей за эту подзадачу подпрограмме.

Какие условия должны быть соблюдены, чтобы программист принял решение о создании подпрограммы? По большому счету их всего два:

1. Обязательное условие. В листинге программы имеются многократно повторяющиеся программные блоки, решающие одну и ту же частную подзадачу.
2. Желательное условие. Высока вероятность, что способ решения подзадачи может оказаться полезным в других программах.

Если вы видите, что первое или второе условие выполняется, то это верный признак, что код следует вынести в подпрограмму (процедуру или функцию).

> ***Замечание***
>
> Разбиение программы на такие фрагменты упрощает отладку кода и повышает его наглядность.

Процедуры

Процедура представляет собой набор сгруппированных вместе операторов, используемых под одним именем. Процедура состоит из заголовка и тела процедуры. Заголовок начинается ключевым словом `procedure`, затем следуют имя процедуры и при необходимости заключенный в круглые скобки список параметров. Также при необходимости объявление процедуры может завершаться специальными директивами.

Тело процедуры заключается в составной оператор `begin..end`. Ключевому слову `begin` могут предшествовать блоки объявления типов, констант и переменных (`type`, `const` и `var`). Объявленные внутри тела процедуры переменные называются *локальными*. Причина такого названия в том, что жизненный цикл такой переменной начинается с вызовом процедуры и заканчивается в момент ее завершения. Эти переменные не доступны извне процедуры.

```
procedure имя_процедуры ([параметры]); [директивы;]
[локальные объявления]
begin
  <операторы>
end;
```

Для вызова процедуры из программы или другой процедуры достаточно просто указать ее имя и при необходимости в круглых скобках передать параметры процедуры.

Вновь вернемся к листингу 5.1, с которого и стартовало обсуждение темы этой главы. Предлагаю дополнить его процедурой, рисующей разделительную линию из нескольких отрезков. Для этого сразу после строки `uses` опишем процедуру с именем `DashLine` (листинг 5.2).

Листинг 5.2. Процедура "рисования" разделительной линии

```
program ch05_02;

{$APPTYPE CONSOLE}

uses SysUtils;

procedure DashLine; //процедура "рисования" разделительной линии
var b:byte;
begin
  for b:=0 to 69 do Write('-');
  WriteLn('');
end;

var A,B,S:real;
    F:byte;
begin
  WriteLn('Выбери фигуру: 1-квадрат; 2-прямоугольник; 3-прямоугольный треугольник');
  DashLine;   //вызов процедуры DashLine
  ReadLn(F);  //выбор пользователя сохраним в переменной F
  WriteLn('Сторона A=?');
  ReadLn(A);  //размер стороны A
  if (F=2) OR (F=3) then  {если прямоугольник или треугольник,
                           для расчетов нужна сторона B}
    begin
      WriteLn('Сторона B=?');
      ReadLn(B);
    end;

  case F of
     1: S:=A*A;    //площадь квадрата
     2: S:=A*B;    //площадь прямоугольника
     3: S:=A*B/2;  //площадь прямоугольного треугольника
    else WriteLn('Допустимы только цифры 1, 2 или 3');
  end;
```

```
  WriteLn('=',S);
  DashLine; //вызов процедуры DashLine
  ReadLn;
end.
```

Теперь в нашем распоряжении имеется процедура `WriteLn()`, умеющая "разговаривать" на русском языке. Каждый раз, когда нам потребуется использовать символы кириллицы, достаточно вместо стандартной процедуры Delphi `WriteLn()` воспользоваться услугами `WriteLn()`.

> ***ВНИМАНИЕ!***
>
> Все процедуры и функции в программе должны иметь уникальные имена. Исключение составляют так называемые перегружаемые методы, специально отмеченные директивой `overload`.

Функции

У процедур есть коллеги — *функции*. С технической точки зрения между процедурой и функцией разница чисто косметическая. В отличие от процедуры, функция всегда явным образом должна возвращать результат своей работы.

Как и процедура, функция обладает заголовком, областью локальных объявлений и заключенным в составной оператор `begin..end` телом функции.

Объявление заголовка функции начинается с упоминания визитной карточки функции — ключевого слова `function`, за ним следует имя функции, затем в круглых скобках перечисляются параметры. Объявление заголовка завершается указанием типа возвращаемых данных и при необходимости директивами.

```
function имя_функции ([параметры]):возвращаемый_тип_данных; [директивы;]
[локальные объявления]
begin
  <операторы>
end;
```

Еще одна характерная черта функций заключается в особой предопределенной переменной `Result`, автоматически создаваемой Delphi при объявлении функции и доступной только из тела этой функции. Переменная `Result` предназначена для возврата в программу результатов выполнения функции, тип переменной точно такой же, как и объявленный в заголовке тип функции.

> ***ВНИМАНИЕ!***
>
> В каждой из функций Delphi автоматически создается переменная `Result`, имеющая тот же тип, что и возвращаемое функцией значение. При описании кода функции программист должен присвоить этой переменной возвращаемое функцией значение.

Найдется ли работа для функции в нашем проекте расчета площади геометрических фигур? И вновь ответ утвердительный. Изучите листинг 5.2, а точнее, оператор-селектор `case`, в рамках которого мы получаем запрашиваемый пользователем результат. Во всех трех случаях ядро вычислений составляет произведение сторон `A*B`. Мы уже твердо знаем, что наличие повторяющихся фрагментов кода служит верным признаком необходимости созда-

ния очередной подпрограммы, в данном случае функции. Листинг 5.3 отображает код вновь усовершенствованного приложения расчета площади. Функция `AreaRect()` в состоянии помочь нам получать площадь прямоугольника.

Листинг 5.3. Доработанная версия программы расчета площади

```
program ch05_03;

{$APPTYPE CONSOLE}

uses  SysUtils;

procedure DashLine; //процедура "рисования" разделительной линии
var b:byte;
begin
  for b:=0 to 69 do Write('-');
  WriteLn('');
end;

function AreaRect(A,B:Real):Real; //расчет площади прямоугольника
begin
  Result:=A*B;//площадь равна произведению сторон прямоугольника
end;

var A,B,S:real;
    F:byte;
begin
  WriteLn('Выбери фигуру: 1-квадрат; 2-прямоугольник; 3-прямоугольный треугольник');
  DashLine;
  ReadLn(F);
  WriteLn('Сторона A=?');
  ReadLn(A); //размер стороны A
  if (F=2) or (F=3) then {если прямоугольник или треугольник,
                          для расчетов нужна сторона B}
    begin
      WriteLn('Сторона B=?');
      ReadLn(B);
    end;
  case F of
     1: S:=AreaRect(A,A);   //площадь квадрата
     2: S:=AreaRect(A,B);   //площадь прямоугольника
     3: S:=AreaRect(A,B)/2; //площадь прямоугольного треугольника
     else WriteLn('Допустимы только цифры 1, 2 или 3');
  end;
  WriteLn('=',S);
  DashLine;
  ReadLn;
end.
```

При написании кода функции вместо применения предопределенной переменной `Result` допускается указывать имя функции, например, так, как продемонстрировано в листинге 5.4.

Листинг 5.4. Альтернативный способ возвращения значения функцией

```
function AreaRect(A,B:Real):Real;
begin
  AreaRect:=A*B;  //аналог записи Result:=A*B;
end;
```

При проверке работоспособности программы расчета площади фигуры вы наверняка заметили, что при возврате результатов подсчетов площадь фигуры представляется на первый взгляд в необычном формате. Взгляните на рис. 5.1, он иллюстрирует работу приложения при подсчете площади квадрата с размером стороны 10 единиц. Наша программа рапортует, что $10 \times 10 = 1.000000000000E+0002$. Здесь нет никакой ошибки, это *научный формат представления* вещественных чисел — приложение уведомляет пользователя, что получен результат, равный 1×10^2. Однако научная форма представления чисел далеко не всегда подходит для обычного пользователя, скорее всего пользователь предпочтет, чтобы вместо записи 1.000000000000E+0002 наша программа вывела обычное число 100.

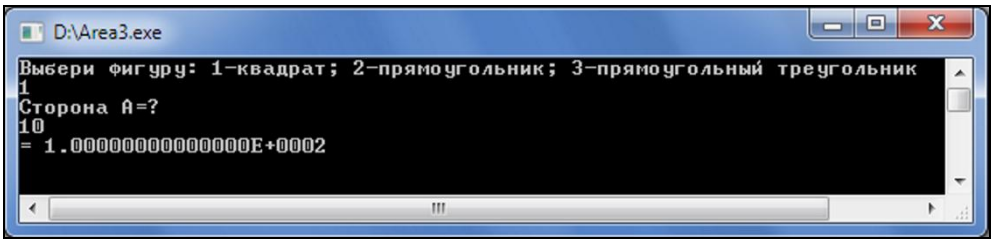


Рис. 5.1. Экранный снимок приложения с научной формой представления результата

В состав языка Delphi входят тысячи разнообразных процедур и функций, которые были созданы разработчиками языка для решения разноплановых задач. По мере прочтения этой книги мы с вами будем знакомиться с наиболее важными из них. Кроме того, дополнительную информацию о стандартных функциях Delphi вы сможете получить из приложений к книге. Сегодня мы воспользуемся услугами одной из стандартных функций, она называется `FloatToStr()`. Благодаря услугам этой функции можно конвертировать действительное число в строку. Во время преобразования научный формат представления чисел самым чудесным образом заменяется общепринятым.

> ***Внимание!***
>
> При преобразовании вещественного числа в строку функция `FloatToStr()` возвращает результат с точностью до четвертого знака после запятой.

Для того чтобы попробовать функцию `FloatToStr()` в действии, найдите в конце листинга 5.3 строку `WriteLn('=',S)` и замените ее новой версией `WriteLn('=',FloatToStr(S))`.

Запустите приложение на выполнение и вновь поставьте перед ним задачу расчета площади квадрата со стороной, равной 10 единицам. Теперь все в порядке (рис. 5.2), полученный результат представлен в *обычном формате*.

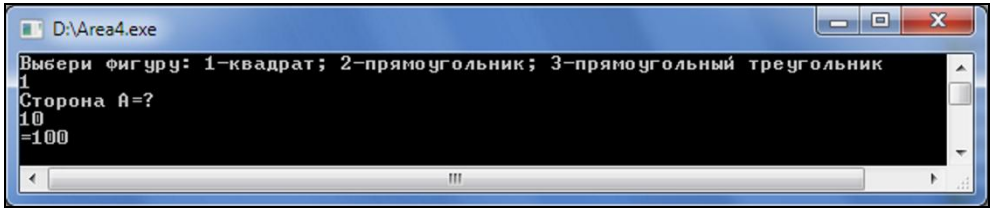


Рис. 5.2. Экранный снимок приложения с обычным представлением результата

Особенности передачи параметров

Механизм обмена данными между программой и подпрограммой основан на использовании параметров процедур и функций Delphi. Параметры, перечисленные в заголовке процедуры (функции), обычно называют *формальными*, а реальные значения, передаваемые в процедуру (функцию) во время ее вызова, называют *аргументами* (или фактическими параметрами).

В процедурах и функциях Delphi может быть задействованы несколько разновидностей параметров:

- параметры-значения;
- параметры-константы `const`;
- выходные параметры `out`;
- параметры-переменные `var`;
- нетипизированные параметры.

С использованием в роли параметров обычных значений мы уже знакомы, мы использовали их при разработке функции для листинга 5.3:

```
function AreaRect(A, B : Real):Real;
```

Такой способ описания параметров имеет наибольшее распространение. Однако стоит знать, что следующим шагом, позволяющим компилятору Delphi еще лучше оптимизировать код приложения, будет применение вместо обычного параметра-значения *параметра-константы*. Единственным условием определения параметра в качестве константы должна быть 100-процентная уверенность в том, что в теле функции этот аргумент не будет претерпевать каких-либо изменений. В листинге 5.5 представлен пример штатной функции Delphi из модуля `SysUtils`.

Листинг 5.5. Функция с параметром-константой

```
function DateToStr(const DateTime: TDateTime): string;
begin
  DateTimeToString(Result, ShortDateFormat, DateTime);
end;
```

Задача функции — преобразование в текстовый вид значения даты/времени. Обратите внимание на то, что в теле функции аргумент `DateTime` не подвергается никаким модификациям, а это значит, что его целесообразно объявить в виде константы.

Замечание

Если параметр объявлен в виде параметра-значения или константы, то при вызове функции помещенное в этот параметр значение копируется в стек, и далее операторы функции работают с копией. В этом случае функция не имеет возможности никаким образом изменить исходное значение. Такой способ передачи параметра называют *передачей по значению*.

Как быть, если результат функции невозможно представить одним-единственным значением? Допустим, возникла необходимость одновременно вычислять не только площадь, но и периметр прямоугольника? Есть несколько вариантов решения этой задачи. Первый вариант находится на поверхности — мы объявляем две специализированные функции. Первая функция вычисляет площадь, а вторая — периметр. Такой подход достаточно прост, поэтому мы его не рассматриваем.

Второй вариант более оригинален. Объявляем запись `TRectInfo` с двумя полями, предназначенными для хранения значения площади и периметра. После объявления записи мы получаем право описать функцию, результатом работы которой также станет тип данных `TRectInfo` (листинг 5.6).

Листинг 5.6. Функция, возвращающая результат в виде записи

```
type
  TRectInfo = record
    Area, Perimeter : REAL;
  end;

function AreaAndPerimeterRect(const A, B : Real): TRectInfo;
begin
  Result.Area:=A*B;
  Result.Perimeter:=2*(A+B);
end;
```

На мой взгляд, это весьма практичное решение нашей задачки — для получения площади и периметра достаточно однократно вызвать функцию `AreaAndPerimeterRect()`.

Благодаря высокой гибкости языка программирования Delphi, на этом возможные варианты решения поставленной задачи не исчерпываются. Третий способ заключается во включении в перечень параметров функции выходного параметра `out` (листинг 5.7).

Листинг 5.7. Функция, возвращающая результат через выходной параметр `out`

```
function AreaAndPerimeterRect(A, B : Real; out Perimeter : Real) : Real;
begin
  Result:= A*B;
  Perimeter:=2*(A+B);
end;

var A, P : Real;
begin
  A:= AreaAndPerimeterRect(5,10, P);
  WriteLn(A);
```

```
  WriteLn(P);
  ReadLn;
end;
```

Площадь прямоугольника мы возвращаем традиционным способом с помощью предопределенной переменной `Result`, а значение периметра присваиваем выходному параметру `Perimeter`. Листинг содержит несколько строк кода, демонстрирующих способ вызова функции с выходным параметром.

Для вызова функции из программы (см. листинг 5.7) мы объявили две переменные вещественного типа. В переменную `A`, предназначенную для хранения площади, результат вычислений помещается уже привычным для нас способом. А периметр фигуры функция возвратит в переменную `P` через выходной параметр.

Наибольшими возможностями обладает *параметр-переменная*. Такой параметр пригодится не только для возврата данных, но и для передачи в функцию или процедуру какого-то входного значения. Листинг 5.8 демонстрирует возможности параметров двойного назначения в рамках процедуры, рассчитывающей площадь и периметр прямоугольника.

Листинг 5.8. Процедура с параметрами-переменными `var`

```
procedure AreaAndPerimeterProc(var A, B : Real);
var x, y : real;
begin
  x:= A*B;     //площадь
  y:=2*(A+B); //периметр
  A:=X;
  B:=Y;
end;

var A,B: real;
begin
  A:=2; B:=3.5; //размеры сторон
  AreaAndPerimeterProc(A, B);
  WriteLn(A);   //площадь
  WriteLn(B);   //периметр
  ReadLn;
end.
```

Сначала параметры-переменные `A` и `B` играют роль входных аргументов — через них в процедуру направляются размеры сторон прямоугольника. Но после выполнения процедуры в параметрах `A` и `B` окажутся площадь и периметр.

> ***Замечание***
>
> Если в вашей процедуре (функции) объявлен выходной параметр или параметр-переменная, то во время обращения к процедуре в стек заносятся адреса ячеек памяти, в которых расположены реальные значения. В этом случае процедура получает возможность изменять исходные значения аргументов. Такой способ передачи параметров называют *передачей по адресу*.

Несмотря на всю свою строгость, при определении параметров процедур и функций язык Delphi допускает некоторые вольности и разрешает программисту опускать определение

типа данных параметров. За примером далеко ходить не стоит. Если взглянуть на упрощенную спецификацию уже знакомой нам процедуры `WriteLn()`, благодаря которой мы до сих пор выводили сообщения в окно консоли

```
procedure WriteLn([ P1; [ ..., PN] ]);
```

то можно увидеть, что ни для одного из параметров функции тип не задан! Дело в том, что эта процедура всеядна и способна "переварить" почти любой тип входных данных, будь это текстовая строка, целое или вещественное число. Немного позднее вы убедитесь, что `WriteLn()` даже способна работать с текстовыми файлами.

К еще одному достоинству языка Delphi стоит отнести возможность передачи в качестве параметра в процедуры и функции массива или даже открытого массива. Посмотрите на пример объявления функции `GetArraySum()` в листинге 5.9.

Листинг 5.9. Открытый массив в роли параметра функции

```
function GetArraySum(A : Array of integer):integer;
var I : Integer;
begin
  Result:=0;
  for I:=Low(A) to High(A) do Result:=Result+A[i];
end;
```

Функция `GetArraySum()` возвратит сумму элементов массива, при этом самостоятельно разберется с размером массива.

> ***ЗАМЕЧАНИЕ***
>
> При описании функции параметры, принимаемые по умолчанию, должны размещаться в конце списка параметров.

При необходимости мы имеем право назначить параметру функции или процедуры значение по умолчанию. Для этого после объявления типа параметра ставим знак равенства и значение аргумента (листинг 5.10).

Листинг 5.10. Параметры со значением по умолчанию

```
function MyFunc(X:integer, Y:integer = 10):integer;
begin
  result :=X+Y;
end;

var Z : integer;
begin
  Z := MyFunc(2); //параметр X=2, параметр Y=10 (значение по умолчанию)
  ...
  Z := MyFunc(5, 20); //параметр X=5, параметр Y=20
end.
```

При вызове функции программисту предоставляется возможность не заполнять второй аргумент функции (оставив его со значением по умолчанию) или передать в него новое значение.

Замечание

Профессионалу надо знать, что параметры процедур и функций, а также локальные переменные передаются в стек. В момент обращения к функции стек заполняется данными, в момент завершения работы — очищается. Основное качество стека — высокая скорость доступа к хранимым в нем данным. По умолчанию размер стека минимален и составляет 16 Кбайт, поэтому, если функция задействует большие по размеру данные (длинные строки, динамические массивы, вариантный тип) — они размещаются в глобальной куче. Для управления размером стека можно воспользоваться директивами компилятора `{$MINSTACKSIZE}` и `{$MAXSTACKSIZE}`.

Директивы для процедур и функций

Для более тонкой настройки поведения процедур и функций Delphi предоставляет в распоряжение программиста набор директив. Сейчас мы познакомимся с наиболее интересными из них: директивой перегрузки `overload`, директивой опережающего объявления `forward` и директивой признака внешнего объявления `external`.

Перегрузка функций: директива *overload*

В этой главе нами была разработана функция `AreaRect()`, позволяющая подсчитать площадь прямоугольника (см. листинг 5.3). При описании `AreaRect()` мы предположили, что для наших расчетов следует воспользоваться типом данных `Real`. Однако не исключено, что в ходе эксплуатации программы выяснится, что иногда надо произвести расчет еще с большей точностью, например, заложенной в тип данных `Extended`. Неужели в этом случае следует заново переписать всю программу, поменяв все типы данных с `Real` на `Extended`? Совсем не обязательно.

Именно для таких случаев в Delphi предусмотрена возможность перегрузки функций, которая активируется обращением к директиве `overload` (листинг 5.11).

Листинг 5.11. Перегрузка функций с помощью директивы `overload`

```
function AreaRect(A,B:Real):Real; overload;
begin
  Result:=A*B;//расчет с точностью Real
end;

function AreaRect(A,B:Extended):Extended; overload;
begin
  Result:=A*B;  //расчет с точностью Extended
end;
```

Механизм перегрузки Delphi позволяет программисту объявлять в одном и том же программном модуле несколько одноименных функций, отличающихся друг от друга только перечнем параметров или типами этих параметров.

Во время выполнения приложения система самостоятельно разберется, какой из экземпляров функции следует вызывать по типу входных параметров. Если мы обратимся к функции `AreaRect()`, передав в нее параметры типа `Real` — будет вызвана первая версия перегружаемой функции, если `Extended` — вторая.

Опережающее объявление: директива *forward*

Допустим, что в нашем проекте существует очень важная процедура MySuperProc. Предполагается, что к этой процедуре предстоит активно обращаться многим другим процедурам и функциям проекта. В такой ситуации обычная практика программирования предполагает следующий порядок объявления функций и процедур: в листинге программы первой должна следовать самая популярная процедура программы MySuperProc, а вслед за ней все остальные, желающие воспользоваться услугами MySuperProc, процедуры и функции проекта (листинг 5.12).

Листинг 5.12. Обычный порядок объявления взаимодействующих процедур

```
procedure MySuperProc; //основная процедура объявлена первой
begin
  //код реализации
end;

procedure Proc1; //вызывающая процедура объявлена второй
begin
  //...
  MySuperProc; //процедура Proc1 вызывает процедуру MySuperProc
end;
```

Такая последовательность объявления процедур имеет простое объяснение — производя синтаксический разбор кода будущего приложения, компилятор считывает листинг сверху вниз. Если вдруг мы нарушим очередность объявления и поместим процедуру Proc1 (из которой производится вызов MySuperProc) хотя бы на несколько строк выше, чем вызываемую процедуру MySuperProc, то мы никогда не увидим свое приложение в действии. Натолкнувшись в теле процедуры Proc1 на строку с обращением к пока еще необъявленной процедуре MySuperProc, компилятор без лишних церемоний выдаст сообщение "Undeclared identifier" и прекратит компиляцию (рис. 5.3).

"Ну и что здесь такого? — спросит читатель. — Надо всегда соблюдать последовательность объявления процедур, и у нас никогда не будет проблем с компиляцией проекта!"

"Нет, будут! И еще какие! — поспешит “обнадежить” такого читателя подготовленный программист. — А что делать, если две процедуры должны вызывать друг друга? Допустим, что процедуре Proc1 по-прежнему необходима поддержка MySuperProc, но при этом процедура MySuperProc в свою очередь намерена обращаться за помощью Proc1".

И действительно, какая должна быть последовательность объявления двух процедур с взаимным вызовом? Ведь невозможно одновременно объявить обе процедуры друг перед другом! Все равно одна из процедур окажется позади, и об нее "споткнется" компилятор.

К счастью, Delphi предоставит нам выход даже из такой запутанной ситуации. В данном случае нам понадобится помощь директивы forward. Благодаря ей можно сделать опережающее объявление процедуры или функций. Другими словами, директива укажет компилятору, что помеченная директивой forward процедура описана несколько позднее. Порядок применения директивы предложен в листинге 5.13.

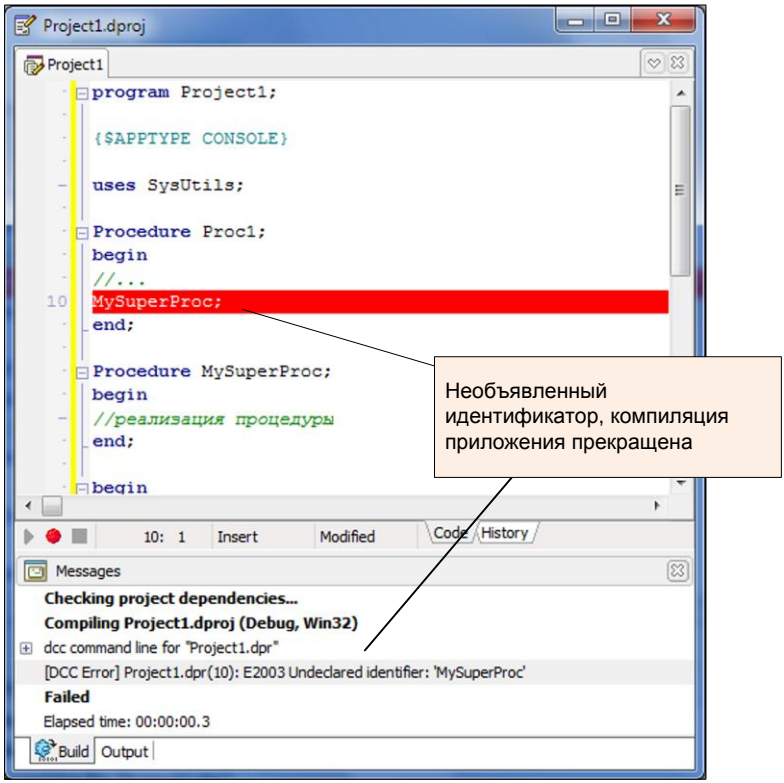


Рис. 5.3. Реакция компилятора на нарушение порядка следования процедур

Листинг 5.13. Опережающее объявление процедур с помощью директивы `forward`

```
{Объявлен только заголовок функции без раздела реализации}
procedure MySuperProc; forward; //опережающее объявление

procedure Proc1;
begin
  ...
  MySuperProc;   {мы намерены воспользоваться услугами
                  процедуры MySuperProc}
end;
...
{полноценное объявление MySuperProc}
procedure MySuperProc;
begin
  //...
  if (логическое условие =true) then Proc1; //вызов процедуры Proc1
end;
```

Теперь, повстречав процедуру, помеченную директивой `forward`, компилятор поймет, что это просто опережающее объявление процедуры `MySuperProc`. И нет смысла поднимать панику, встретив вызов `MySuperProc` из других процедур и функций проекта.

Внешнее объявление: директива *external*

Все современные операционные системы и функционирующее на их базе программное обеспечение построены из большого числа взаимодействующих модулей. Между модулями четко распределены обязанности: одни отвечают за графический вывод, другие — за работу в сети, третьи — за печать. Каждый из модулей зачастую прячет в себе несколько процедур и функций, которыми можно воспользоваться из других приложений, и в частности из приложений, написанных на языке Delphi.

Для уведомления компилятора о том, что мы намерены воспользоваться внешней процедурой или функцией, следует использовать директиву `external`.

Встроенная функция: директива *inline*

Если код, заключенный в тело функции, минимален, то имеет смысл объявлять функцию как *встроенную*. Для этого предназначено ключевое слово `inline`, например

```
function MyFunction(const A: string): string; inline;
```

Компилятор Delphi, встретив встроенную функцию, вместо обращения к функции попытается встроить код функции непосредственно в точку ее вызова.

Применение встроенных функций имеет свои достоинства и недостатки. Положительная сторона заключается в существенном снижении расходов на вызов функции. Ведь теперь не надо искать код функции в памяти и передавать ему управление, не надо сохранять и восстанавливать регистры процессора, одним словом, не надо производить большой перечень трудоемких действий. Отрицательная сторона встраивания функций — возрастание размера машинного кода в исполняемом файле.

Рекурсивная функция

Среди многочисленных функций особое место занимают *рекурсивные функции*. Под рекурсивной понимается функция, способная вызывать саму себя. Наиболее простым примером рекурсивного алгоритма считается алгоритм получения факториала числа. Формула факториала рекуррентна по своей природе

$$n! = \begin{cases} 1 & \text{для } n = 0, \\ n \times (n-1)! & \text{для } n > 0, \end{cases}$$

поэтому рекурсивная функция здесь как нельзя кстати (листинг 5.14).

Листинг 5.14. Рекурсивная функция расчета факториала

```
function factorial(n: byte):UInt64;
begin
  if n>1 then factorial:= n*factorial(n-1)
  else result:=1;
end;
```

На вход функции `factorial()` поступает целое число, факториал от которого мы рассчитываем получить. Стоит отметить, что в предложенной постановке мы сможем рассчитать

факториал любого натурального числа из диапазона от 1 до 65. Значение факториала от 66 уже настолько велико, что не вмещается в тип данных `UInt64`.

Процедурный тип данных

В *главе 2* (см. рис. 2.1) уже упоминалось о существовании процедурного типа данных. Благодаря процедурному типу мы получаем право передавать все права на обращение к процедуре или функции переменной процедурного типа. Объявление процедурного типа данных начинается с ключевого слова **type**, после которого следует имя типа, признак функции или процедуры и (при необходимости) параметры.

```
type тип_функции = function([параметры]):тип_данных;
type тип_процедуры = procedure([параметры]);
```

Листинг 5.15 демонстрирует фрагмент кода, раскрывающий порядок работы с процедурным типом.

Листинг 5.15. Пример процедурного типа данных

```
function AreaRect(A,B:Real):Real;//заголовок функции
...
type TFunction = function(A,B:Real):Real;//процедурный тип

var A,B,S:Real;
    F: Byte;
    P: TFunction; //переменная процедурного типа
begin
  ...
  P:=AreaRect; //присвоение переменной ссылки на функцию

  case F of
     1: S:=P(A,A);   //выясняем площадь квадрата
    ...
```

В листинге 5.15 объявлен процедурный тип данных `TFunction`, полностью повторяющий параметры целевой функции `AreaRect()`. Затем мы вводим переменную `P` типа `TFunction` и несколькими строками ниже присваиваем переменной `P` ссылку на функцию `AreaRect()`. С этого момента для расчета площади прямоугольника разрешается обращаться к процедурной переменной `P`.

> ***ЗАМЕЧАНИЕ***
>
> Процедурный тип данных в качестве параметров допускает применение обобщенных типов (generic types). Более подробно об этом мы поговорим в *главе 8*.

Анонимные функции

Сравнительно недавно язык Delphi стал позволять программисту использовать в своей работе особую разновидность функций, называемых *анонимными* (anonymous)[1]. Своим назва-

[1] Иногда анонимные функции называют лямбда-функциями.

нием функция обязана тому, что при ее объявлении не требуется имени. Обычно анонимная функция описывается непосредственно в месте ее вызова *(контекстный вызов)*, а ссылка на нее передается в отдельную переменную. Эта переменная и обеспечивает вызов функции.

Синтаксис объявления анонимной процедуры и функции представлен в листинге 5.16.

Листинг 5.16. Синтаксическая конструкция

```
//Объявление процедуры
procedure([параметры процедуры])
begin
  {...}
end;

//Объявление функции
function([параметры функции])): [тип возвращаемого значения]
begin
  {...}
end;
```

Пример вызова анонимной процедуры вы обнаружите в листинге 37.3.

> ***ЗАМЕЧАНИЕ***
>
> Анонимные процедуры и функции могут передаваться в качестве параметра в другие процедуры и функции Delphi или возвращаться в качестве результата выполнения функции. Кроме того, объекты Delphi имеют право использовать анонимные методы.

ГЛАВА 6

Файлы и каталоги

Все без исключения до сих пор встречавшиеся в книге типы данных имеют одну общую отрицательную черту — они не в состоянии осуществлять долговременное хранение данных. Будь то простая или строковая переменная, запись или массив, все они располагают байты с информацией в энергозависимой памяти компьютера — в ячейках микросхем ОЗУ (оперативного запоминающего устройства). После отключения питания микросхемы ОЗУ утрачивают возможность хранить данные. Одним словом, вместе с электричеством исчезают все значения из наших переменных и структур. Что делать, если такое положение вещей нас не устраивает? Ответ прост: нам надо научиться сохранять информацию на энергонезависимых устройствах — на жестком диске или во флэш-памяти. Одним словом, пора разобраться с основным способом представления информации на винчестере компьютера — файловым типом данных.

Файловый тип данных (или попросту файл) является основой хранения данных в современном компьютере.

Типизированные файлы

Одна из визитных карточек Delphi — умение языка программирования работать с файлами, хранящими набор однотипных элементов. В качестве таких элементов могут выступать все простые, строковые и большинство структурных типов данных, рассмотренных в *главах 2* и *3* (см. рис. 2.1). Тип подлежащих хранению в файле данных определяется в момент объявления файловой переменной (листинг 6.1).

Листинг 6.1. Объявление файловых переменных для типизированных файлов

```
var fI : file of Integer;    //файл целых чисел Integer
    fD : file of Double;     //файл действительных чисел Double
    fS : file of String[10];//файл строк с размером элемента 10 символов
    fDR: file of TDemoRecord; //файл записей TDemoRecord (см. листинг 3.3)
```

> ***Внимание!***
>
> Файловый тип данных поддерживается только в проектах VCL, созданных исключительно для платформы Windows. Если вы проектируете приложение FireMonkey для iOS и Mac OS X, от использования файлового типа данных следует отказаться.

Вне зависимости от решаемой задачи (будь это создание файла, запись или только чтение данных) работа с файлом всегда начинается с установки ассоциации между файловой переменной и именем файла. Для этих целей предназначена процедура

```
procedure AssignFile(var F: File; FileName: String);
```

Процедура проинициализирует файловую переменную `F`, единственное для этого условие — наличие во втором параметре `FileName` имени целевого файла. При определении имени файла желательно указывать полное имя (путь к каталогу файла, имя и расширение имени файла). Если же мы не упомянем путь к каталогу, то Delphi станет полагать, что файл расположен в той же папке, что и исполняемый файл проектируемого приложения.

> ***ВНИМАНИЕ!***
>
> Нельзя повторно вызвать процедуру `AssignFile()` для файловой переменной, уже связанной с файлом. Файловая переменная освобождается только после выполнения процедуры `CloseFile()`.

Независимо от решаемой программистом задачи, последним этапом работы с файлом должно стать освобождение файловой переменной с одновременным закрытием обрабатываемого файла. Поэтому в качестве напарницы `AssignFile()` всегда должна выступать процедура

```
procedure CloseFile(var F);
```

Во время осуществления операций ввода/вывода могут произойти многочисленные непредвиденные ситуации. Нетерпеливый пользователь может преждевременно выдернуть из гнезда USB накопитель флэш-памяти, на котором размещен файл, может отключиться сетевой диск либо возникнуть другие неприятные происшествия. Хорошее приложение, вне зависимости от мешающих факторов, обязано завершить работу с файлом и вызвать процедуру его закрытия. Для этого программисты помещают процедуры `AssignFile()` и `CloseFile()` в секцию защиты от ошибок `try..finally` (листинг 6.2).

Листинг 6.2. Шаблон кода для работы с файлом

```
const FileName='c:\demo.dat'; //имя файла
var   F:file of integer; //файловая переменная

begin
  AssignFile(F,FileName);
  try //защищенная от ошибок секция кода
  //потенциально опасные операции с файлом
  finally //секция гарантированного завершения кода
    CloseFile(F); //закрываем файл
  end;
end.
```

Конструкция `try..finally` играет очень ответственную роль. При возникновении любой исключительной ситуации при работе с файлом (например, логической ошибки в коде программы) выполнение кода прекращается и управление передается в секцию гарантированного завершения `finally`. В нашем примере именно здесь и расположилась процедура закрытия файла. Таким образом, конструкция `try..finally` сделает все, чтобы при любом стечении обстоятельств процедура `CloseFile()` закрыла файл и освободила файловую переменную.

Замечание

Более подробно порядок работы с конструкциями защиты от ошибок мы рассмотрим в *главе 16*.

После инициализации файловой переменной программная логика работы с файлом может пойти по одному из трех направлений (см. алгоритм на рис. 6.1):

1. Создание нового файла с последующей возможностью записи в него данных.
2. Открытие существующего файла с целью чтения данных.
3. Открытие существующего файла с возможностью как чтения, так и записи.

Для того чтобы конкретизировать, в каком режиме доступа мы намерены открыть файл, стоит настроить системную переменную `FileMode`. По умолчанию переменная содержит значение 2, что соответствует универсальному режиму чтения/записи, режим записи устанавливается значением 1, только чтение — 0.

Для открытия уже существующего файла `F` в режиме, определенном в системной переменной `FileMode`, вызываем процедуру

```
procedure Reset(var F: File [; RecSize: Integer]);
```

Сразу после открытия курсор устанавливается на первом элементе файла. Если файл не типизирован, то следует воспользоваться необязательным параметром `RecSize`, определяющим размер читаемой порции данных. Для типизированного файла этот параметр не задействуется.

Если файл физически отсутствует, то для создания экземпляра файла следует воспользоваться процедурой

```
procedure Rewrite(var F: File [; RecSize: Word ] );
```

В результате на жестком диске компьютера появится новый файл (размером 0 байт), готовый к получению данных.

Внимание!

Будьте осторожны! Если на жестком диске уже существует файл с таким же именем, как и имя, ассоциированное с файловой переменной с помощью `AssignFile()`, то процедура `Rewrite()` сотрет этот файл.

Научившись получать доступ к файлу, перейдем к следующему этапу изучения порядка работы с типизированными файлами и познакомимся с процедурами, предназначенными для чтения и записи данных. Хотя, вполне вероятно, слово "познакомимся" не вполне корректно. Многим из вас процедуры *ввода/вывода* (input/output) покажутся знакомыми, ведь мы ими уже пользовались в некоторых предыдущих примерах этой книги.

```
procedure Read(var F: File; V1; [ ..., VN]);     //чтение из файла
procedure Write([var F: File]; P1; [ ..., PN]); //запись в файл
```

В обеих процедурах в параметр `F` в качестве аргумента направляется файловая переменная. Процедура `Read()` читает элемент данных и направляет его в параметр `V1`, процедура `Write()`, наоборот, заносит в файл элемент данных из `P1`. Особо отметим, что тип данных параметров `P1` и `V1` должен быть идентичен типу файловой переменной.

При каждом обращении к процедуре чтения `Read()` из файла считывается очередная порция данных и внутренний курсор перемещается к очередному элементу файла. В конце концов,

курсор считает последнюю запись из файла и окажется в конце файла. Для того чтобы предотвратить попытки чтения за пределами файла, следует регулярно опрашивать функцию

```
function Eof([var F: File]): Boolean;
```

В момент достижения конца файла функция возвратит `true`.

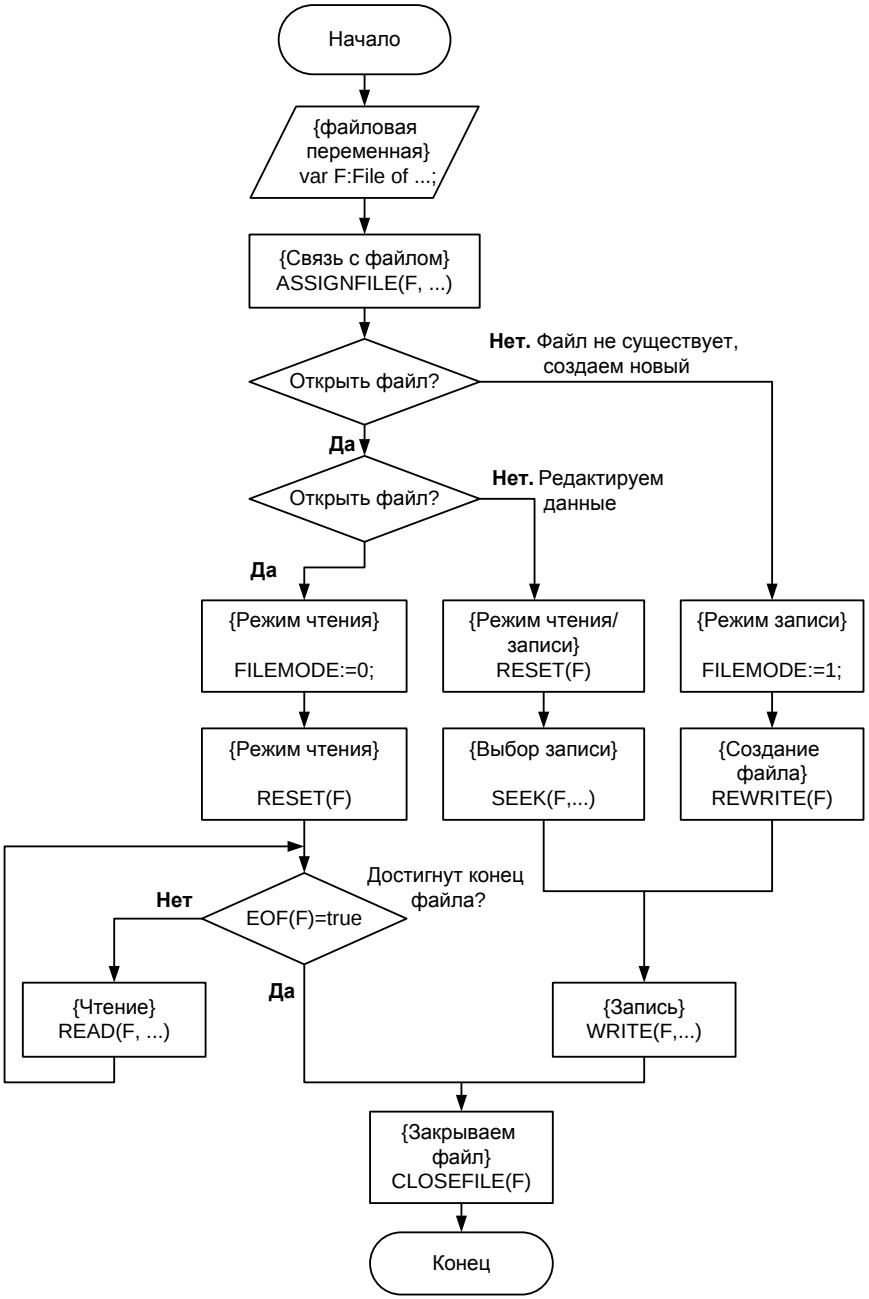


Рис. 6.1. Блок-схема алгоритма работы с типизированным файлом

Для того чтобы выяснить, в каком месте файла расположился курсор, следует воспользоваться функцией

```
function FilePos(var F: File): Integer;
```

Если курсор находится в начале файла (так происходит при вызове метода `Reset`), функция возвращает 0.

Количество элементов в файле подскажет функция

```
function FileSize(var F: File): Integer;
```

Заострю внимание читателя на том, что функция определения размера файла возвращает размер не в байтах, а в элементах, из которых состоит файл. Например, если элементом файла является целое число `Integer`, то для получения физического размера типизированного файла следует умножить результат выполнения функции `FileSize(F)` на размер этой записи в байтах `SizeOf(integer)`.

Для быстрого перемещения по элементам типизированного файла можно воспользоваться процедурой

```
procedure Seek(var F: File; N: Longint);
```

Параметр `N` определяет номер позиции, в которую следует направиться.

Пример работы с типизированным файлом

Для закрепления полученных знаний на практике напишем небольшую программу, демонстрирующую порядок работы с типизированным файлом. Наша задача — разработать прайс-лист для хранения наименования товара и его стоимости.

Задача достаточно сложна, поэтому декомпозируем ее на несколько подзадач:

1. Выбор структуры данных.
2. Проектирование основного приложения.
3. Разработка процедур просмотра, вставки, редактирования и удаления записей из файла.

В листинге 6.3 вы найдете решение первой и второй подзадач.

Листинг 6.3. Структура данных и ядро приложения

```
program ch06_02;

{$APPTYPE CONSOLE}
uses SysUtils;

type TPrice=packed record //структура элемента файла
  Article:string[16];     //поле для хранения названия
  Price:currency;         //поле стоимости товара
end;

const fName='price.dat';  //константа с именем файла
var   Action:Char;        //код выбранной пользователем операции
      F:File of TPrice;   //файловая переменная
      P:TPrice;           //переменная для хранения элемента файла
```

```
{НАЧАЛО РАБОТЫ ПРОГРАММЫ}
begin
  if FileExists(fName)=false then //если файл отсутствует
  begin
     AssignFile(F,fName); //связываемся с файловой переменной
     Rewrite(F);          //создаем пустой файл
     CloseFile(F);        //закрываем файл
  end;

  repeat
    WriteLn('Выбери операцию: 1-просмотр; 2-добавление;
                    3-редактирование; 4-удаление; 5-выход');
    ReadLn(Action); //пользователь выбирает операцию
    case Action of
      '1': {вызов процедуры просмотра файла (листинг 6.4)};
      '2': {вызов процедуры добавления элемента (листинг 6.5)};
      '3': {вызов процедуры редактирования элемента (листинг 6.6)};
      '4': {вызов процедуры удаления элемента (листинг 6.7)};
      '5': {завершаем работу};
      else WriteLn('Неверный код операции!');
    end;
  until Action='5';
end.
```

В качестве структуры, предназначенной для хранения элемента файла, выбрана запись `TPrice` с двумя полями.

Выполнение программы начинается с проверки факта существования файла с данными. Для этого мы обращаемся к функции

```
function FileExists (const FileName: string): Boolean;
```

и передаем в нее имя типизированного файла. Если файл отсутствует, то функция возвратит значение `false`. Это станет сигналом для создания нового пустого файла.

Убедившись в существовании файла с данными, программа выводит строку приглашения, в которой пользователю предлагается выбрать ту или иную операцию. Результат выбора помещается в переменную `Action`, которая в свою очередь обрабатывается в операторе-селекторе `case`. В зависимости от значения в переменной будет вызвана та или иная процедура.

Теперь приступим к решению третьей подзадачи — проектированию процедур ввода/вывода. Начнем с чтения файла и отображения его записей на экране (листинг 6.4).

Листинг 6.4. Процедура вывода на экран записей файла

```
procedure ShowItems; //просмотр файла
var Num:Cardinal;    //номер элемента файла
begin
  AssignFile(F,fName);
  try
    Reset(F);
```

```
    Num:=1;
    while EOF(F)<>true do //цикл чтения
      begin
        Read(F,P); //читаем элемент из файла
        WriteLn(Num,#9,P.Article,#9,FloatToStr(P.Price)); //вывод на экран
        Inc(Num);  //увеличим счетчик строк
      end;
  finally
    CloseFile(F);
  end;
end;
```

Обратите внимание, что чтение файла осуществляется внутри цикла `while..do`. Условием выхода из цикла станет достижение конца файла, об этом нам просигнализирует функция `EOF()`.

Вставку новой записи в конец файла осуществит листинг 6.5.

Листинг 6.5. Вставка новой записи в файл

```
procedure InsertItem;
begin
  WriteLn('Введите название и цену товара');
  ReadLn(P.Article,P.Price); //пользователь ввел значения
  AssignFile(F,fName);
  try
    FileMode:=1;          //режим записи
    Reset(F);
    Seek(F,FileSize(F)); //перемещаемся в конец файла
    Write(F,P);          //вставляем новый элемент
  finally
    CloseFile(F);
  end;
end;
```

Добавление новой записи должно осуществляться в конец прайс-листа, поэтому после открытия файла мы обращаемся к процедуре `Seek()` и перемещаем внутренний курсор в конец файла.

Для того чтобы научить программу редактировать запись, изучите код листинга 6.6.

Листинг 6.6. Редактирование записи

```
procedure EditItem;
var Num:Cardinal;
begin
  WriteLn('Введите номер редактируемой строки');
  ReadLn(Num);
  WriteLn('Введите новое название и цену товара');
  ReadLn(P.Article,P.Price);
```

```
  AssignFile(F,fName);
  try
    FileMode:=1;
    Reset(F);

    if (Num>=0) and (Num<=FileSize(F)) then
      begin
        Seek(F,Num-1); //перемещаемся на нужную запись
        Write(F,P);
      end else WriteLn('Неверный номер!');
  finally
    CloseFile(F);
  end;
end;
```

Идея процедуры проста. Пользователь выбирает номер записи, которую он намерен исправить, и переходит к этой записи с помощью процедуры `Seek()`.

В нашей программе наиболее интересной окажется процедура удаления записи из файла (листинг 6.7).

Листинг 6.7. Удаление записи из файла

```
procedure DeleteItem;  //удаляем элемент
var OldF, NewF : File of TPrice;
    Num, DelNum:Cardinal;
begin
  WriteLn('Введите номер удаляемой строки');
  ReadLn(DelNum);
  AssignFile(OldF,fName); AssignFile(NewF,fName+'~');
  try
    Reset(OldF); Rewrite(NewF);
    Num:=1;
    while EOF(OldF)<>true do //чтение всего файла
      begin
        Read(OldF,P);         //прочитали данные из файла
        if Num<>DelNum then  //пропускаем удаляемый элемент
           Write(NewF,P);    //остальное переносим в новый файл
        Inc(Num);
      end;
  finally
    CloseFile(OldF);
    CloseFile(NewF);
  end;
  Erase(OldF);  //удаляем старый файл
  Rename(NewF, fName); //переименовываем новый файл
end;
```

Для физического удаления записи мы вынуждены пойти на хитрость. Нами создается временный файл с именем, заканчивающимся символом ~. В этот файл переносятся все записи

из исходного файла за исключением записи с номером, указанным пользователем. Затем исходный файл удаляется. Для этого применяется процедура

```
procedure Erase(var F: File);
```

Затем временный файл занимает место исходного. Для этого он переименовывается с помощью процедуры

```
procedure Rename(var F: File; Newname: String);
```

> ***ВНИМАНИЕ!***
>
> Процедуры `Erase()` и `Rename()` могут быть вызваны только после закрытия файла!

Особенности удаления записей из больших файлов

Стоит заметить, что предложенный в листинге 6.7 способ удаления записей хорош только для сравнительно небольших типизированных файлов. Как только размер файла становится внушительным, удаление одной записи станет занимать много времени — ведь ради избавления от одной записи мы вынуждены вновь перезаписать целый файл! А если за один сеанс работы требуется "расправиться" с несколькими элементами файла, то процесс удаления становится просто утомительным.

В подобных случаях программисты идут на хитрость. Один из наиболее распространенных приемов заключается в следующем. Во-первых, изменяется структура элемента файла — в нее добавляется еще одно поле, хранящее признак, что элемент файла нам больше не нужен (листинг 6.8).

Листинг 6.8. Доработка структуры элемента файла

```
type TPrice=packed record //усовершенствованная структура элемента файла
  Article:string[16];     //поле для хранения названия
  Price:currency;         //поле стоимости товара
  Deleted:Boolean; //если true — признак, что запись подлежит удалению
end;
```

Во-вторых, во время сеанса работы с программой вместо физического удаления записи (с вызовом ресурсоемкой процедуры переноса элементов из одного файла в другой) мы просто отмечаем значением `true` записи, подлежащие удалению.

В-третьих, реальное удаление записей из файла осуществляется единственный раз, например перед завершением работы приложения. В этот момент вызывается процедура, схожая с процедурой листинга 6.7, но на этот раз в результирующий файл не переносятся ненужные записи.

Существует еще один прием борьбы с лишними элементами файла. Допустим, что в состоящем из N элементов файле требуется удалить запись M ($M < N$). Если для логики программы не важен порядок следования записей, то имеет смысл поступить следующим образом. Данные из последнего элемента файла N переносятся на место удаляемого элемента M, просто замещая его. Теперь нам предстоит удалить ставший ненужным последний элемент файла N. С помощью процедуры `Seek()` переводим внутренний курсор файла на удаляемый элемент и вызываем процедуру

```
procedure Truncate(var F: File);
```

С точки зрения скорости выполнения процедура очень эффективна. Она просто обрезает файл, удаляя из него все записи, находящиеся после текущего положения курсора.

Текстовые файлы

Для организации хранения исключительно текстовых данных вместо типизированных файлов целесообразно использовать их "коллег" — текстовые файлы. Идея работы с текстовым файлом во многом напоминает порядок работы с типизированным файлом. В этом вы можете убедиться, сравнив алгоритм работы с текстовым файлом (рис. 6.2) с изученным в начале главы алгоритмом (см. рис. 6.1). Однако есть и исключения, на них мы остановимся подробнее.

Работа с текстовым файлом также начинается с объявления файловой переменной, но на этот раз тип файла определяется как `TextFile`:

```
var F:TextFile;
```

Для осуществления операций чтения/записи кроме уже знакомых нам процедур `Read()` и `Write()` в текстовых файлах разрешено использовать процедуры

```
procedure ReadLn(var F: File; [ ..., VN]);
procedure WriteLn([var F: File]; [ P1; [ ..., PN] ]);
```

Они позволяют читать/записывать данные построчно, что весьма удобно при обработке текстовой информации.

Во время чтения текстовых данных хорошую помощь способны оказать функции

```
function SeekEoln([var F: Text]): Boolean;
function SeekEof([var F: Text]): Boolean;
```

Функции позволяют пропускать символы пробелов в текстовом файле, перемещая внутренний курсор на значащие символы. Первая функция дает возможность избавляться от пробелов в отдельной строке, вторая — от полностью пустых строк.

Для демонстрации возможностей функций воспользуемся листингом 6.9.

Листинг 6.9. Чтение текста с подавлением пробелов

```
var f:textfile;
    s:string;
begin
  AssignFile(f,'c:\demo.txt');
  Reset(f);
  while eof(f)<>true do
    begin
      SeekEof(f);  //если есть пустая строка, пропускаем ее
      SeekEoln(f); //если строка начинается с пробелов — пропускаем
      Readln(f,s); //читаем строку в переменную S
      Writeln(s);  //выводим строку на экран
    end;
  CloseFile(f);
  Readln;
end.
```

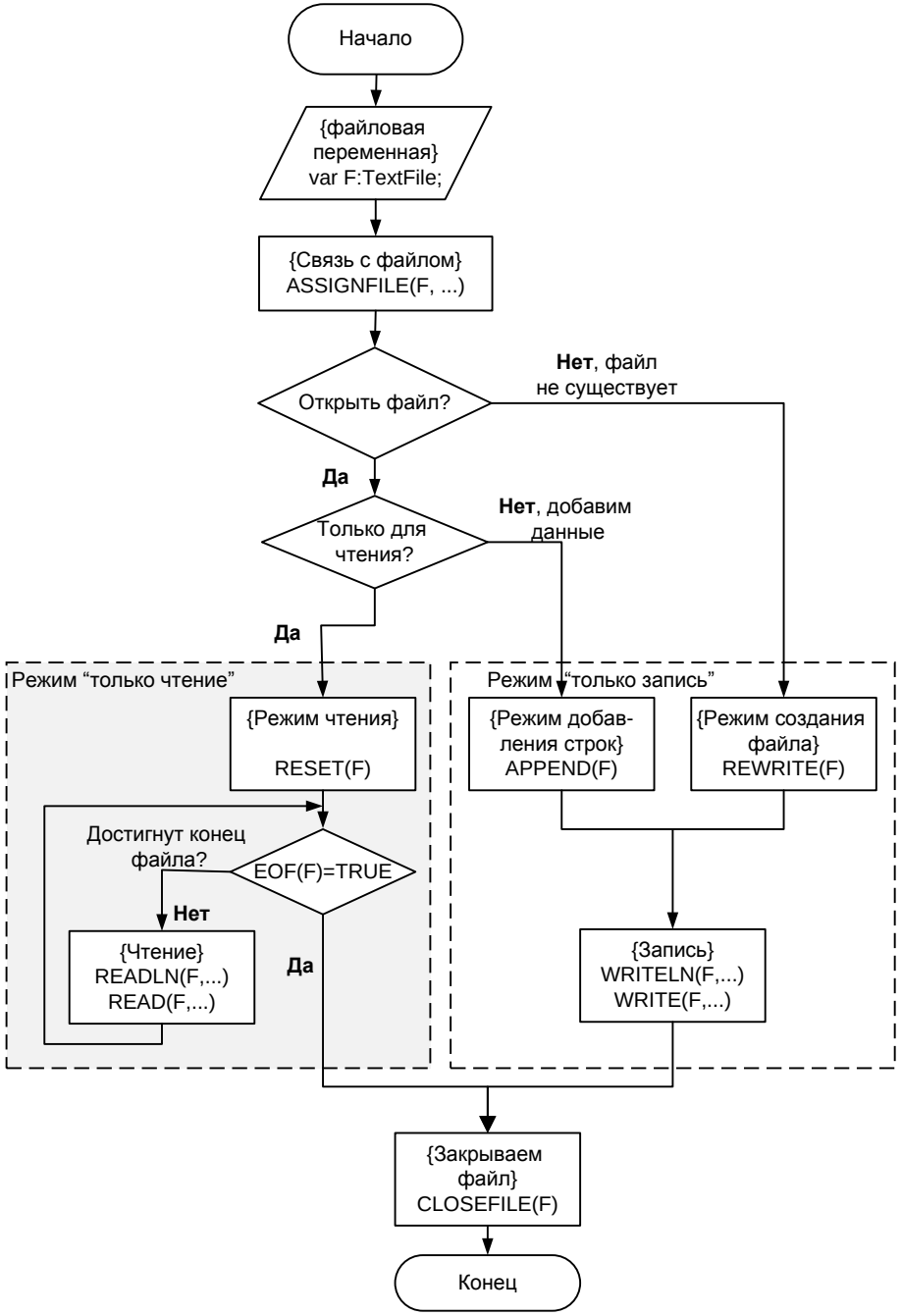


Рис. 6.2. Блок-схема алгоритма работы с текстовым файлом

Прежде чем запустить программу, создайте (например, с помощью программы Блокнот) текстовый файл, включающий пустые строки вначале файла и строки, начинающиеся с пробелов. Сохраните файл под именем c:\demo.txt и запустите код на выполнение. Результат выполнения программы иллюстрирует рис. 6.3. Обратите внимание, что в окне консольного приложения исчезли пустые строки и пробелы вначале строк.

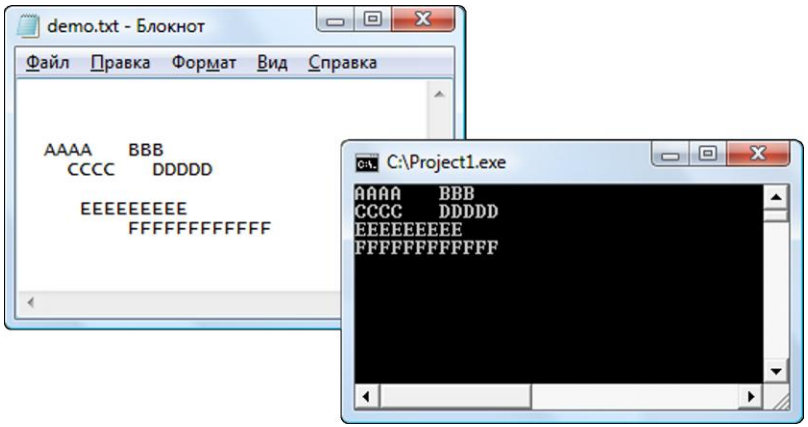


Рис. 6.3. Демонстрация подавления пробелов при чтении текста

При работе с текстовым файлом (см. рис. 6.2) мы можем читать текст, создавать новый файл для записи в него текста и открывать файл с данными для добавления в него новых строк. Режим добавления строк активируется вызовом процедуры:

```
procedure Append(var F: Text);
```

В этом случае доступ к файлу предоставляется только с возможностью записи.

> ***Внимание!***
>
> В модуле `IOUtils` объявлена интеллектуальная запись `TFile`, значительно упрощающая работу с текстовым файлом *(см. приложение 7)*.

Двоичные файлы

Подавляющее большинство хранящихся на жестких дисках вашего компьютера файлов не попадает под классификационные признаки типизированного или текстового файла. Эти файлы обладают существенно более сложной структурой и содержат наборы самых разнообразных данных. При работе с файлами неизвестного формата следует применять приемы работы с *двоичными (бинарными) файлами*.

Ключевое отличие состава базовых операций ввода/вывода, применяемых для двоичных файлов, заключается в осуществлении операций чтения и записи. При работе с бинарными данными чтение и запись целесообразно производить блоками с помощью функций

```
function BlockRead(var F: File; var Buf; Count: Integer;
                   [var Result: Integer]): Integer;
function BlockWrite(var F: File; const Buf; Count: Integer;
                    [var Result: Integer]): Integer;
```

С первым параметром `F` все ясно — это файловая переменная. Второй параметр `Buf` служит буфером, в который записываются прочитанные данные или, наоборот, из которого считы-

вается информация для сохранения в файле. Параметр `Count` определяет, сколько блоков данных планируется считать из файла за одну операцию чтения. Обычно ему присваивают значение, равное единице. Последний параметр `Result` необязательный. Из него (во время операций чтения) мы можем узнать количество записей, реально считанных из файла. Соответственно, во время осуществления записи в переменной окажется количество записей, внесенных в файл.

Объявление файловой переменной для работы с бинарным файлом также несколько отличается от переменной, обслуживающей типизированный или текстовый файл. Теперь достаточно сказать, что мы работаем с обычным файлом `var f:file`.

Во всем остальном работа с двоичным файлом не должна вызвать проблем. Например, листинг 6.10 демонстрирует порядок чтения данных из произвольного файла.

Листинг 6.10. Чтение данных из бинарного файла, как из произвольного

```
procedure ReadBinFile(fName:string);
var F:file;
    Buf:array[0..4095] of byte;
begin
  AssignFile(F,fName);
  try
    Reset(F);
    while EOF(F)=false do
    begin
      BlockRead(F,Buf,1);
      //прочие операции
    end;
  finally
    CloseFile(F);
  end;
end;
```

ЗАМЕЧАНИЕ

В Delphi реализован целый комплекс классов, специализирующихся на обслуживании разнообразных файлов, в их числе `TFileStream`, `TStringList`, `TIniFile` и многие другие классы.

Управление дисками, каталогами и файлами

Одна из самых сложных задач, с которой рано или поздно сталкивается разработчик программного обеспечения, связана с управлением файлами, каталогами и дисками компьютера. Основная трудность этой области программирования связана с высокой вероятностью возникновения различного рода исключительных ситуаций (вызванных некорректным совместным использованием файлов и папок, ограничением свободного пространства, некорректным доступом к ресурсу, ошибками операций чтения и записи и т. п.). Далее мы рассмотрим наиболее распространенные приемы работы с файловой системой Windows и познакомимся с ключевыми функциями и процедурами.

ЗАМЕЧАНИЕ

Основные функции управления файлами, каталогами и дисками сосредоточены в модулях `SysUtils`, `FileCtrl`, `IOUtils` и конечно в модуле `Windows`, который позволяет использовать функции из набора Windows API.

Работа с дисками компьютера

Говоря о дисках компьютера, первое, что нас может заинтересовать, — это состав логических дисков на вашем компьютере. Предусмотрено несколько способов сбора этих сведений. Наиболее простой среди них заключается в задействовании возможностей интеллектуальной записи `TDirectory` (модуль `IOUtils`). Среди многочисленных методов записи имеется функция класса

```
class function GetLogicalDrives: TStringDynArray;
```

создающая динамический массив строк, заполненный именами дисков в формате: `<БуквaДиска>:\`. Меньше десяти строк кода (листинг 6.11) позволят передать названия дисков в комбинированный список.

Листинг 6.11. Получение перечня логических дисков с помощью `TDirectory`

```
uses ioutils,types;

procedure TForm1.FormCreate(Sender: TObject);
var Disks:TStringDynArray;
    i:integer;
begin
  Disks:=TDirectory.GetLogicalDrives;
  for i:=0 to High(Disks) do ComboBox1.AddItem(Disks[i],nil);
end;
```

Еще один весьма полезный метод, на этот раз из обоймы Windows API, расскажет нам о типе диска, подключенного к системе:

```
function GetDriveType (Drive : PChar) : Integer;
```

Метод требует указания всего одного параметра — указателя на строку с именем диска — и возвращает результат в соответствии с табл. 6.1. С помощью `GetDriveType()` также легко получить перечень дисков, пример вы найдете в *главе 21* (см. листинг 21.10).

***Таблица 6.1.** Расшифровка результата выполнения `GetDriveType()`*

Константа	Описание
`DRIVE_UNKNOWN` (0)	Невозможно определить тип диска
`DRIVE_NO_ROOT_DIR` (1)	Ошибочное имя, например, когда диск с этим именем не смонтирован
`DRIVE_REMOVABLE`	Съемный диск
`DRIVE_FIXED`	Жесткий диск
`DRIVE_REMOTE`	Сетевой диск
`DRIVE_CDROM`	Привод оптических дисков
`DRIVE_RAMDISK`	Диск в оперативной памяти

Полезные сведения о размерах диска и свободном пространстве возвращают функции

```
function DiskSize(Drive: Byte): Int64;
function DiskFree(Drive: Byte): Int64;
```

Параметр `Drive` требует передачи номера накопителя. Например: 0 — текущий, 1 — A:, 2 — B:, 3 — C: и т. д. Если метод не в состоянии выяснить размер диска, то он возвратит –1.

Оба вышерассмотренных метода втайне от нас пользуются услугами функции Windows API:

```
function GetDiskFreeSpaceEx(Drive : PChar; FreeBytesAvailable,
         TotalNumberOfBytes, TotalNumberOfFreeBytes : INT64):Boolean;
```

Преимущество этого метода в том, что он всего за один вызов делает то, с чем `DiskFree()` и `DiskSize()` справляются только вдвоем. А именно, он возвращает число свободных байтов (`FreeBytesAvailable`), доступных в данный момент пользователю, и общее число байтов на диске (`TotalNumberOfBytes`). При этом имя диска передается в очеловеченном виде (например `'C:\'`), а не числовым значением. Если метод выполнился удачно, то он возвратит `true`. Четвертый параметр функции не используется, вместо него передавайте неопределенный указатель — `nil`.

Сведения о диске компьютера пополнит еще одна функция из пакета Windows:

```
function GetVolumeInformation (
  Drive : PChar;            //указатель на строку с именем диска
  VolumeNameBuffer : PChar; //буфер, в который занесется имя тома
  VolumeNameSize : DWord;   //размер буфера с именем тома
  VolumeSerialNumber : DWord;    //указатель на серийный номер тома
  MaximumComponentLength : DWord;//максимальное число символов в имени
  FileSystemFlags : DWord;      //указатель на флаги файловой системы
  FileSystemNameBuffer : PChar; //буфер с названием файловой системы
  FileSystemNameSize : DWord    //размер буфера с названием
) : Integer;
```

В первом аргументе требуется указать имя исследуемого диска (не забыв про двоеточие и слэш). Во все остальные параметры требуется подставить соответствующие массивы и переменные (листинг 6.12). При успешном завершении работы функция вернет `true`.

Листинг 6.12. Получение информации о логическом томе

```
var   VolumeName, FileSystemName: array [0..MAX_PATH-1] of Char;
      SerNum, MaxLen, NotUsed, VolFlags : DWORD;
begin
  GetVolumeInformation(PChar('C:\'),VolumeName,
                       DWord(Sizeof(VolumeName)),
                       @SerNum,
                       MaxLen,
                       VolFlags,
                       FileSystemNameBuffer,
                       DWord(SizeOf(FileSystemName)));
  //дальнейшая обработка
end;
```

В примере осталось нераскрытым содержание параметра — FileSystemFlags. Это флаг (или комбинация флагов), выявляющий некоторые особенности файловой системы, например, такие, как поддержка регистра в названиях файлов, возможность сжатия тома и т. п. Более полную информацию вы можете получить из справочной системы.

В заключительной части разговора о дисках познакомимся еще с одной функцией Windows API. Она предназначена для смены метки тома.

```
function SetVolumeLabel(Drive, VolumeName : PChar):Boolean;
```

Название тома передавайте через параметр VolumeName.

Организация поиска каталогов и файлов

Еще со времен первой версии Delphi в языке были реализованы три специализированные функции, осуществляющие поиск файлов и каталогов, в соответствии с критериями, определяемыми программистом. Все три функции должны работать в тесном союзе друг с другом и вызываться только в определенной последовательности. Первая из функций предназначена для инициализации поиска и нахождения первого файла, соответствующего нашим требованиям:

```
function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer;
```

В первом параметре Path функции указываем путь и имя каталога, в котором следует организовать поиск, кроме того, в Path передаем маску поиска. Например, путь и маска 'C:\Mydir*.txt' зададут поиск в каталоге 'C:\Mydir' всех файлов с расширением txt. Второй параметр Attr определяет атрибуты искомых файлов (табл. 6.2).

Таблица 6.2. *Константы атрибутов файла*

Атрибут	Значение	Описание
faReadOnly	$00000001	Только для чтения
faHidden	$00000002	Скрытый
faSysFile	$00000004	Системный
faVolumeID	$00000008	Диск
faDirectory	$00000010	Каталог
faArchive	$00000020	Архив
faAnyFile	$0000003F	Любой

Если файл успешно найден, то функция вернет нулевое значение, а данные о найденном файле окажутся в переменной-записи F типа TSearchRec (листинг 6.13).

Листинг 6.13. Состав полей записи TSearchRec

```
type TSearchRec = record
  private
    function GetTimeStamp: TDateTime;
  public
    Time: Integer;           //дата-время создания файла
    Size: Integer;           //размер в байтах
```

```
    Attr: Integer;           //атрибуты файла
    Name: TFileName;         //имя файла
    ExcludeAttr: Integer;    //не документировано
    FindHandle: THandle;     //дескриптор файла
    FindData: TWin32FindData;//структура с дополнительной информацией
    property TimeStamp: TDateTime read GetTimeStamp;//метка времени
end;
```

Логическим продолжением функции `FindFirst()` выступает

```
function FindNext(var F: TSearchRec): Integer;
```

Задача функции заключается в поиске оставшихся файлов и папок. Так же как и в случае со своей предшественницей, при нахождении файла функция возвращает нулевое значение и записывает данные в переменную `F`.

По завершению поиска процедура

```
procedure FindClose(var F: TSearchRec);
```

в обязательном порядке должна освободить задействованные ресурсы. Пример, демонстрирующий порядок работы с функциями, вы обнаружите в листингах 6.16 и 22.4.

В современных версиях Delphi предусмотрены и альтернативные способы сбора сведений о каталогах и файлах. У объявленной в модуле `IOUtils` записи `TDirectory` *(см. приложение 7)* имеется ряд методов, существенно упрощающих процесс поиска. Мы рассмотрим лишь три, наиболее востребованных. В их числе перегружаемая функция `GetDirectories()`, способная построить список подкаталогов, принадлежащих каталогу `Path`.

```
class function GetDirectories(const Path, SearchPattern: string;
        const SearchOption: TSearchOption): TStringDynArray;
```

Перегружаемая функция

```
class function GetFiles(const Path, SearchPattern: string;
        const SearchOption: TSearchOption): TStringDynArray;
```

нацелена на поиск файлов.

Универсальная функция

```
class function GetFileSystemEntries(const Path,
        SearchPattern: string): TStringDynArray;
```

соберет сведения как о каталогах, так и о файлах. Все функции обладают идентичным перечнем параметров: `Path` указывает путь к родительскому каталогу; `SearchPattern` назначает маску поиска; `SearchOption` определяет опции поиска (`soTopDirectoryOnly`, `soAllDirectories`). В результате выполнения собранные сведения помещаются в динамический массив строк (листинг 6.14).

Листинг 6.14. Сбор имен каталогов и файлов, принадлежащих диску C:

```
var SDA:TStringDynArray;
    i:integer;
begin
  SDA:=TDirectory.GetFileSystemEntries('C:\','*.*');
  for i:=0 to High(SDA) do ComboBox1.AddItem(SDA[i],nil);
end;
```

Для поиска конкретного файла стоит воспользоваться услугами функции

```
function FileSearch(const Name, DirList: string): string;
```

В аргумент `Name` направляют имя искомого файла, а в параметр `DirList` передают перечень путей (разделенных точкой с запятой), в которых следует осуществить поиск. Если в перечне путей не окажется правильного пути, то метод вернет пустую строку.

Проверка существования файла и каталога

Для того чтобы сразу отсечь достаточно большой объем ошибок, перед обращением к файлу или каталогу всегда проверяйте факт их существования:

```
function FileExists(const FileName: string): Boolean;
function DirectoryExists(const Directory: string): Boolean;
```

Функции возвращают `true`, если файл (каталог) действительно существует.

Расположение системных каталогов

Существенную помощь в выяснении места расположения системных папок нам окажут функции Windows API. В первую очередь отмечу информационные методы, рассказывающие нам о системных каталогах:

```
function GetWindowsDirectory(Buffer : PChar; Size : Integer):Integer;
function GetSystemDirectory (Buffer : PChar; Size : Integer):Integer;
function GetTempPath(Size : Integer; Buffer : PChar):Integer;
```

Функции передадут в буфер `Buffer` пути к папке Windows (листинг 6.15), системному каталогу и каталогу временных файлов. Если функции выполнились без ошибок, то они возвратят количество символов в пути, в противном случае вы получите нулевое значение.

Листинг 6.15. Расположение каталога Windows

```
function GetWinDir : string;
var  Buf : array [0..MAX_PATH-1] of Char;
begin
  GetWindowsDirectory(Buf,SizeOf(Buf));
  Result:=StrPas(Buf);
end;
```

Дополнительные сведения о размещении каталогов сможет предоставить специализирующая на обслуживании имен файлов и папок запись `TPath` (модуль `IOUtils`). В арсенале `TPath` имеется метод

```
class function GetHomePath: string;
```

готовый поделиться сведениями о папке Roaming с данными приложений.

Упомянем важное понятие *"текущий каталог"* — это каталог, в котором производятся текущие операции ввода/вывода. Для работы с текущим каталогом предназначены функции

```
function GetCurrentDir: string;
function SetCurrentDir (const Dir: string): Boolean;
```

позволяющие выяснить, какой из каталогов является текущим, и назначить новый текущий каталог.

Создание, удаление, копирование и перемещение

Создание пустого каталога можно доверить функциям

```
function CreateDir(const Dir: string): Boolean;
function ForceDirectories(Dir: string): Boolean;
```

Первая из упомянутых функций создаст новый каталог внутри текущего каталога. Вторая функция способна на более серьезные подвиги — она за один вызов в состоянии сформировать цепочку каталогов.

Нет ничего проще, чем просто удалить файл. Для этого достаточно знать имя стираемого файла и название функции

```
function DeleteFile(const FileName: string): Boolean;
```

Если по какой-то причине удалить файл невозможно, то функция возвратит `false`.

С удалением каталога дела обстоят немного сложнее. Если каталог пуст, то для "расправы" над ним следует воспользоваться функцией

```
function RemoveDir(const Dir: string): Boolean;
```

Но если подлежащий удалению каталог содержит вложенные каталоги и файлы, то придется потрудиться. Вниманию тех, кто не ищет простых путей, предложу листинг 6.16. Он содержит пример рекурсивной функции, вызывающей саму себя до тех пор, пока подлежащий удалению каталог не будет полностью очищен, затем каталог удаляется.

Листинг 6.16. Функция полной очистки и удаления каталога

```
function ClearAndRemoveDir(Dir: TFileName): Boolean;
var SR: TSearchRec;
begin
  Result := true;
  Dir := IncludeTrailingBackslash(Dir);

  if FindFirst(Dir + '*.*', faAnyFile, SR)=0 then
    repeat
      if (SR.Name = '.') or (SR.Name = '..') then Continue;

      if (SR.Attr AND faDirectory <>0 ) then
        begin //Рекурсивный вызов удаления каталога
          Result := ClearAndRemoveDir(Dir + SR.Name);
          if Result=false then
            begin
              raise Exception.Create('Полное удаление невозможно!');
              exit;
            end;
        end else
        begin//Иначе удаляем файл
          Result := SysUtils.DeleteFile(Dir + SR.Name);
```

```
          if Result=false then
            begin
              raise Exception.Create('Полное удаление невозможно!');
              exit;
            end;
        end;
    until FindNext(SR)<>0;
  FindClose(SR);
  Result:=RemoveDir(Dir); //обычное удаление уже пустого каталога
end;
```

Для "лентяев", которым предложенная в листинге 6.16 функция показалась слишком громоздкой, у меня (а точнее, у Delphi) также есть подарок. Объявленная в модуле `IOUtils` запись `TDirectory` обладает методом

```
class procedure Delete(const Path: string;  const Recursive: Boolean);
```

самостоятельно решающим задачу удаления папки с вложенными элементами. Для активизации рекурсии надо не забыть установить в состояние `true` параметр `Recursive`.

Для копирования файлов используйте функцию Windows API

```
function CopyFile(ExistingFileName, NewFileName : PChar;
                  FailIfExists : Boolean): Boolean;
```

Здесь: `ExistingFileName` хранит указатель на строку с полным именем копируемого файла; `NewFileName` содержит указатель на строку с полным именем нового файла; параметр `FailIfExists` определяет поведение метода при совпадении имен (если параметр равен `true` и файл с таким именем уже существует, то копирование прерывается). В случае успеха функция возвращает ненулевое значение.

Для перемещения или переименования файлов и каталогов используйте функцию

```
function MoveFile(ExistingFileName, NewFileName : PChar) : Boolean;
```

Параметры функции идентичны первым двум аргументам метода `CopyFile()`. В случае успеха функция возвращает ненулевое значение.

Размер файла

В начале главы мы уже упоминали функцию `FileSize()`, способную получать размер типизированного файла. Однако применение `FileSize()` не всегда оправдано, т. к. в качестве единицы измерения принят не байт, а запись файла. Поэтому для выяснения размера файла в байтах целесообразно использовать функцию из состава Windows API:

```
function GetFileSizeEx(hFile : HANDLE; pFileSize : Pointer) : Boolean;
```

Функция снабжена двумя аргументами: `hFile` — дескриптор файла и `hFileSize` — указатель на переменную типа `INT64` (в которую будет внесен полученный размер). Простейшим примером использования функции может стать листинг 6.17.

Листинг 6.17. Получение размера файла в байтах

```
function SizeOfFile(const FileName : TFileName):INT64;
var  F : HANDLE;
```

```
begin
  F:=FileOpen(FileName,fmOpenRead);
  Result:=GetFileSize(F, @Size);
  FileClose(F);
end;
```

Дата и время создания файла и каталога

Самый простой способ для знакомства с возрастом файла заключается в использовании функции

```
function FileAge(const FileName: string): Integer;
```

Как видите, для этого достаточно передать путь и название файла в параметр с именем `FileName`. Еще один метод, выполняющий аналогичную задачу, называется

```
function FileGetDate (Handle: Integer): Integer;
```

Однако здесь вместо имени требуется указатель на файл. Другими словами, последний должен быть открыт функцией `FileOpen()`. Оба метода возвращают дату не в привычном для нас формате `TDateTime`, а в виде структуры даты/времени, принятой в Windows. Так что для приведения результата к принятому в Delphi виду применяйте функцию

```
function FileDateToDateTime(FileDate: Integer): TDateTime;
```

Существует и функция, решающая обратную задачу:

```
function DateTimeToFileDate(DateTime: TDateTime): Integer;
```

Пример получения даты создания файла предложен в листинге 6.18.

Листинг 6.18. Дата/время создания файла

```
Function DateTimeOfFile(const FileName : TFileName):TDateTime;
var  Age : INTEGER;
begin
  Age:=FileAge(FileName);
  Result:=FileDateToDateTime(Age);
end;
```

Для того чтобы назначить файлу новые время и дату, понадобится функция

```
function FileSetDate(Handle: Integer; Age: Integer): Integer;
```

Порядок работы с функцией демонстрирует листинг 6.19.

Листинг 6.19. Изменение даты/времени создания файла

```
function ChangeDateTimeOfFile(const FileName : TFileName;
                              NewDateTime:TDateTime):TDateTime;
var  F,FDate : INTEGER;
begin
  FDate:=DateTimeToFileDate(NewDateTime);
  F:=FileOpen(FileName, fmOpenReadWrite);
  FileSetDate(F, FDate);
  FileClose(F);
end;
```

ЗАМЕЧАНИЕ

Вновь напомню о существовании записей `TFile` и `TDirectory` *(см. приложение 7)*, в арсенале которых имеются методы, позволяющие не только читать и редактировать дату/время создания файла (каталога), но и дату/время последнего редактирования файла (каталога), а также дату/время последнего доступа к файлу (каталогу).

Атрибуты файла и каталога

В языке Delphi имеется несколько "специалистов" по работе с атрибутами файла. За чтение атрибутов и установку атрибутов соответственно отвечают

```
function FileGetAttr(const FileName: string): Integer;
function FileSetAttr(const FileName: string; Attr: Integer): Integer;
```

В обеих функциях `FileName` — имя файла. В методе установки атрибутов есть второй параметр — `Attr`, с помощью которого и назначаются атрибуты. С существующими типами атрибутов мы уже встречались (см. табл. 6.2).

Контроль атрибутов файла весьма полезен при проведении с файлами потенциально опасных операций. Ознакомьтесь с листингом программы, демонстрирующей простейший способ защиты системных и скрытых файлов от удаления. Представленная в листинге 6.20 функция перед удалением файла проверяет его атрибуты и не допускает удаление скрытых или системных файлов.

Листинг 6.20. Контроль удаления файла

```
function DeleteSimpleFile(FileName : TFileName) : Boolean;
var   Attr : INTEGER;
begin
  Attr:=FileGetAttr(FileName);
  if (Attr and faHidden = 0) or (Attr and faSysFile = 0) then
    begin
      DeleteFile(FileName);
      Result:=True;
    end else Result:=False;
end;
```

Наиболее частая операция, связанная с установкой (удалением) атрибута "только для чтения", унифицирована и представлена в лице метода:

```
function FileSetReadOnly(const FileName: string; ReadOnly: Boolean): Boolean;
```

Здесь `FileName` — имя файла, `ReadOnly` — состояние атрибута.

ЗАМЕЧАНИЕ

Дополнительную информацию об организации работы с файлами и каталогами вы найдете в *приложениях 6* и *7*.

ГЛАВА 7

Концепция ООП

Своим появлением на свет подавляющее большинство современных языков программирования (и в том числе Delphi) обязано весьма неординарной идее — парадигме объектно-ориентированного программирования (ООП).

Корни ООП уходят в далекий 1967 год. Именно тогда в Норвегии Кристен Нюгор (Kristen Nygaard) и Уле-Йохан Даль (Ole-Johan Dahl) придумали очень необычный по тем временам язык Simula. Язык создавался для моделирования процессов реального мира, поэтому модули языка строились не на процедурах, а на объектах и классах — краеугольных камнях современного ООП. Нельзя сказать, что в далекие 1960-е годы появление Simula было воспринято как сенсация, скорее наоборот — язык был подвергнут жесткой критике и практически не нашел поклонников. Однако, как это очень часто бывает, понимание глубины заложенных в Simula идей пришло несколько позже[1].

Основные понятия ООП

Во многом ответ на вопрос: "Что такое ООП?" дает сам термин "объектно-ориентированное программирование". ООП — это программирование, основным действующим лицом которого выступает объект. Объект — это очень широкое понятие. В окружающем нас мире объект — это то, к чему можно прикоснуться руками: это клавиша, сама клавиатура, настольная лампа, спящая под ней моя кошка и проезжающий за окном автомобиль. У объектов имеются характеристики, взглянув на которые мы можем судить об их состоянии. Лампа может быть включена или выключена, клавиша нажата или отпущена, автомобиль может двигаться по дороге со скоростью 60 км/ч, а может, как кошка, мирно дремать, но на этот раз не под лампой, а на стоянке или в гараже.

Объекты, с которым привыкли иметь дело программисты, сильно отличаются от объектов реального мира. Но впрочем, в обеспечении абсолютной идентичности и нет смысла. Я полагаю, что никто из читателей и не рассчитывал повернуть ключ зажигания в автомобиле, "собранном" на языке Delphi, и уехать на нем на прогулку. Программные объекты призваны лишь моделировать физические объекты, а для этого вместо шестеренок, винтиков и болтиков они должны обладать полями, свойствами, методами и событиями (рис. 7.1).

Каждый из объектов обладает некоторыми характеристиками, рассматривая и сравнивая которые мы можем судить о сходстве или различиях между ними. Так, несмотря на всю

[1] За выдающийся вклад в развитие концепции ООП оба ученых в 2001 году были удостоены премии Тьюринга.

похожесть друг на друга кнопок клавиатуры, мы отличаем их по сопоставленному символу алфавита. Все присущие объекту характеристики в терминах ООП называют *полями* (fields), по сути, это принадлежащие объекту переменные или структуры определенного типа, в которые записываются значения, отражающие состояние объекта.

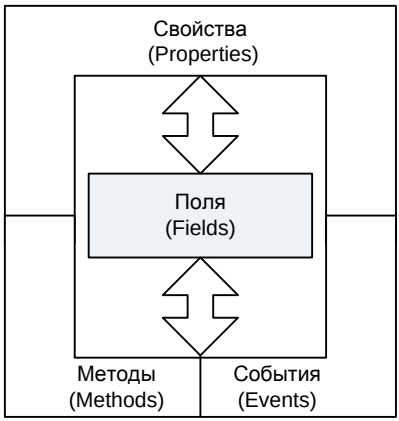


Рис. 7.1. Объект Delphi

Напрямую доступ к своим полям объект никогда не предоставляет. Вместо этого у него имеются посредники: функции (которые в объектно-ориентированных технологиях обычно называются *методами* (methods)) и *свойства* (properties). Настольной лампой управляют два ключевых метода: включить и выключить. У автомобиля этих методов значительно больше: он заводится, набирает скорость, изменяет направление движения, замедляет движение. В первую очередь методы и свойства воздействуют на поля объекта, так практически все методы автомобиля сосредоточены вокруг полей, описывающих его скорость и направление движения. Если методы объекта возвращают какое-нибудь значение, то они реализуются в виде функции, если нет — то метод представляется процедурой.

У наиболее совершенных объектов имеется особая разновидность методов — *события* (events). Благодаря событиям объект приобретает возможность реагировать на окружающий мир и заставлять мир обращать внимание на объект. В программах Delphi наиболее распространенное событие — щелчок (по кнопке, по элементу меню, по гиперссылке и т. п.).

Объект не может быть соткан из воздуха, в объектно-ориентированных языках в качестве предтечи объекта выступает *класс*. Класс — это не что иное, как чертеж проектируемого объекта, но не нарисованный карандашом, а описанный на алгоритмическом языке. В чертеж включается перечень полей будущего объекта и управляющие ими методы и свойства.

Признанный гуру в области объектно-ориентированного проектирования Гради Буч (Grady Booch) в своих работах выделяет четыре обязательных элемента, без которых идея объектно-ориентированного подхода оказывается недееспособной. К обязательным ингредиентам концепции ООП относятся: абстрагирование, инкапсуляция, модульность и наследование.

Абстрагирование

Все наши приложения описывают реальные физические объекты и/или процессы, в которых участвуют эти объекты. Возьмем даже программы-игрушки, например стратегии. В них моделируется поведение полководца, гоняющего свою электронную армию по экрану монитора с целью подчинить себе весь мир. В хороших стратегиях учитываются боевые качества юнитов, толщина брони, дальнобойность и калибр артиллерии, характеристики местности и многие другие факторы, оказывающие прямое или косвенное влияние на результат боя. Чем больше мелочей смог описать разработчик, тем правдоподобнее впечатление. Но, тем не менее, это не реальность, а всего лишь ее абстрактное представление, удобное для игроков. Приведу другой пример: рассмотрим такой объект, как автомобиль. Это весьма сложная конструкция, включающая тысячи деталей, работающих (или нет) по сложным

математическим, физическим, химическим и другим законам. Но с точки зрения начинающего водителя любой автомобиль можно абстрагировать как транспортное средство, предназначенное для перемещения грузов из точки *А* в точку *Б*. Чтобы закончить абстракцию, добавим к перечисленному марку, цвет и номерной знак над бампером. И все! Среднестатистическому водителю нет никакого дела до того, как от двигателя крутящий момент добирается до колес его механического коня.

> ***Замечание***
>
> Абстракция выделяет ключевые характеристики некоторого объекта, отличающие его от всех других видов объектов, и отбрасывает незначительные детали. Таким образом, абстракция определяет концептуальные границы объекта с точки зрения программиста или пользователя.

Инкапсуляция

Следующим за абстрагированием столпом, на котором держится концепция ООП, считается инкапсуляция. Основное назначение инкапсуляции — скрытие от пользователя особенностей реализации класса. Проводя параллель с окружающим нас миром, скажу, что точно так же от нас спрятаны внутренности телевизора. Снаружи только необходимые органы управления, используя которые мы можем найти интересующий нас канал, изменить громкость, яркость. При этом телезрителя мало интересует частота гетеродина, работа декодера PAL или уровень напряжения в блоке дежурного питания, а для многих названные мною термины сродни тарабарщине. Если что-то сломалось, то мы вызываем мастера. Он специалист по инкапсуляции и бесстрашно снимает заднюю стенку телевизора...

Уже интуитивно понятно, что благодаря инкапсуляции повышается надежность функционирования программного объекта, т. к. в отличие от четырех-пяти винтов в задней стенке телевизора неопытный программист не сможет выкрутить винты из объекта, исповедующего традиции ООП. Но помимо повышения общей надежности объекта, инкапсуляция обеспечивает программную логику его работы, а это еще более важное звено в идее объектно-ориентированного программирования. Классическим примером использования инкапсуляции является способ предоставления доступа к полям объекта через его свойства и методы.

Модульность

Модульность — это способность системы строиться на основе совокупности взаимодействующих самостоятельных элементов. Подавляющее большинство окружающих нас вещей состоит из таких элементов. Например, системный блок компьютера содержит модули процессора, памяти, видеокарты, которые в свою очередь также состоят из модулей (диодов, транзисторов, резисторов и т. д.).

Поддержание модульности в программировании сродни искусству. В классическом структурном программировании нужны определенные способности, чтобы грамотно разнести используемые совместно подпрограммы по модулям, в объектно-ориентированном программировании требуется умело разделить классы и объекты. Деление на модули значительно упрощает процесс разработки программного обеспечения: мы собираем программу из заранее подготовленных кубиков. Наивысший пилотаж — умение программиста разработать программу, в которой можно заменять один "кубик" другим без перекомпиляции исходного кода.

Наследование

Еще одним ингредиентом объектно-ориентированного программирования считается наследование. Это совсем простое понятие. Его смысл в том, что при описании нового класса допустимо брать за основу другой, родительский класс. Таким образом достигается преемственность поколений — дочерний класс наследует все методы и свойства своего предка. Ограничений на длину цепочки наследования нет, в Delphi объявлены классы, имеющие в родословной больше десятка предков. Таким образом, самый последний в иерархии наследования класс, если так можно выразиться, самый опытный: он впитал все лучшее от всех своих "дедушек" и "прадедушек".

Вне зависимости от функционального назначения объекта в роли его самого "древнего" предка выступает основа иерархии библиотеки визуальных компонентов Delphi — класс `TObject`. Кроме того, `TObject` — это единственный класс, не имеющий предка, — он создан с чистого листа.

В результате объединения усилий инкапсуляции, модульности и иерархичности появляется едва ли не самый сильный из козырей объектно-ориентированного программирования — *полиморфизм*. Дословная расшифровка этого термина обозначает обладание многими формами. Идея полиморфизма заключается в праве экземпляра некоторого класса представлять все классы из его иерархической цепочки. Более того, благодаря полиморфизму мы получаем право скрывать или переопределять поведение унаследованных методов. Поэтому различные по содержанию процедуры и функции всевозможных объектов могут использовать одно и то же имя. Вызов же метода будет приводить к выполнению кода, соответствующего конкретному экземпляру объекта.

Класс Delphi

Основной структурной единицей всей концепции ООП выступает класс. *Класс* — это программное описание будущего объекта, включающее в себя определение полей, методов и свойств. Объявление класса начинается с ключевого слова `type`, включает имя и признак класса `class` (листинг 7.1).

Листинг 7.1. Синтаксическая конструкция объявления класса

```
type
   Имя_класса = class [abstract | sealed] (класс-предок)
   //поля и методы
end;
```

При объявлении класса можно воспользоваться одним из ключевых слов: `abstract` и `sealed`. Класс, отмеченный словом `abstract`, называют *абстрактным*. Основная особенность абстрактного класса заключается в том, что непосредственно на его основе невозможно создать самостоятельный объект или, как говорят программисты, *экземпляр класса*. Назначение абстрактного класса заключается в другом — он определяет концепцию построения всех своих наследников. При желании абстрактный класс можно сравнить с наброском рисунка, который впоследствии художник возьмет в качестве основы своей картины.

> ***ВНИМАНИЕ!***
>
> *Абстрактный класс* предназначен для определения родовых черт своих наследников. Это своего рода замысел будущего объекта, поэтому объявленные в таком классе методы и

свойства не нуждаются в реализации. Характерным примером абстрактного класса Delphi можно считать класс `TComponent` — прообраз всех компонентов, которые вы видите на палитре компонентов.

Еще одно ключевое слово `sealed` появилось в Delphi сравнительно недавно, оно запрещает классу иметь наследников. Лишать класс возможности "заводить" потомство логично только в случае, когда вы хотите защитить свое творение (например, компонент) от других разработчиков, которые хотели бы незаконно переработать ваше детище.

По умолчанию новый класс создается на основе базового для всей иерархии классов *визуальной библиотеки компонентов* (Visual Components Library, VCL) класса `TObject`. Но никто не запрещает нам, немного "порывшись" в дереве классов VCL, найти для своего класса наиболее подходящего предка.

Проектирование класса

Допустим, что перед нами поставлена задача — разработать класс, моделирующий автомобиль. На первых порах мы не станем углубляться в детали и обсуждать предпочтительную для нас коробку передач или особенности обвода кузова. Будем проще, в качестве прообраза будущего класса предлагаю взять листинг 7.2.

Листинг 7.2. Объявление класса `TAutomobile`

```
type TAutomobile= class(TObject)
  fSpeed:single;      //поле скорости движения

  function SetSpeed(Value:single):single;  //установить скорость движения
end;
```

Предлагаемый прототип автомобиля снабжен полем `fSpeed`, отвечающим за хранение данных о текущей скорости движения. За управление полем несет ответственность метод `SetSpeed()`, он позволит нам увеличить или уменьшить скорость автомобиля.

> ***ЗАМЕЧАНИЕ***
>
> По существующему соглашению имена полей классов всегда должны начинаться с префикса "f" (сокращение от англ. *field*).

> ***ЗАМЕЧАНИЕ***
>
> Если разрабатываемый класс основан на `TObject`, то при объявлении класса допускается не конкретизировать имя предка **`type`** `TAutomobile=`**`class`**.

Каждый знает, что автомобиль без двигателя — не более чем груда металлолома. Поэтому создадим класс двигателя `TEngine` (листинг 7.3) и интегрируем его в автомобиль.

Листинг 7.3. Объявление класса `TEngine`

```
type TEngine= class
  fEnabled :boolean; //двигатель включен — true, выключен — false

  procedure SetEnabled(Value:Boolean); //запуск или останов двигателя
  function  GetEngineState:boolean;    //состояние двигателя
end;
```

```
function TEngine.GetEngineState: boolean;//состояние двигателя
begin
  Result:= fEnabled; //если true — двигатель включен
end;

procedure TEngine.SetEnabled(Value: Boolean);
begin
  fEnabled:=Value;
end;
```

В нашем примере в состав двигателя вошло поле `fEnabled`, которое определяет состояние двигателя (`true` — включен, `false` — отключен). Метод `SetEnabled()` отвечает за запуск и остановку двигателя. Функция `GetEngineState()` информирует о текущем состоянии двигателя.

В отличие от трудоемкой операции установки настоящего двигателя в реальную машину, вставка его алгоритмической модели займет не более минуты (листинг 7.4). Для того чтобы двигатель оказался "под капотом" класса `TAutomobile`, объявляется поле `fEngine:TEngine` (если вас устроит такое сравнение — поле `fEngine` является аналогом двигательного отсека в корпусе реального автомобиля).

Листинг 7.4. Инкапсуляция двигателя `TEngine` в автомобиль `TAutomobile`

```
type TAutomobile= class(TObject)
  fSpeed:single;       //поле скорости движения
  fEngine:TEngine;     //поле двигателя

  function SetSpeed(Value:single):single;  //установить скорость движения
end;
```

Для того чтобы продемонстрировать пример взаимодействия автомобиля и его двигателя, в состав методов автомобиля добавим функцию `SetSpeed()`, она возьмет на себя ответственность за управление скоростью движения автомобиля (листинг 7.5).

Листинг 7.5. Реализация класса `TAutomobile`

```
function TAutomobile.SetSpeed(Value:single):single;
begin
  Result:=0;
  if fEngine.GetEngineState=true then {если двигатель заведен}
    begin
      if (Value>=0) and (Value<=240) then {проверим корректность значения}
        begin
          fSpeed:=Value;
          Result:=fSpeed; //скорость успешно изменена
        end
     else Raise Exception.Create('Допустимый диапазон 0...240 км/ч');
   end else
     begin
       fSpeed:=0;
```

```
      Raise Exception.Create('Необходимо запустить двигатель!');
    end;
end;
```

Метод `SetSpeed()` не просто передает значение скорости в поле `fSpeed`. В коде функции осуществляются две проверки. Во-первых, функция не допускает установки значения скорости при выключенном двигателе. Во-вторых, осуществляется проверка вхождения нового значения скорости в пределы разумных ограничений от 0 до 240 км/ч.

Управление жизненным циклом объекта

Любой класс VCL автоматически вооружается парой специализированных методов, несущих прямую ответственность за жизненный цикл объекта. Это:

- *конструктор* (`constructor`), который отвечает за создание из класса нового объекта (экземпляра класса). По существующей договоренности конструктор описывается в рамках метода `Create()`;
- *деструктор* (`destructor`), в обязанности которого входит уничтожение объекта с одновременным освобождением отведенных под обслуживание объекта ресурсов. Функционал деструктора обычно закладывается в процедуру `Destroy()`.

Описание класса автомобиля, включающее конструктор и деструктор, выглядит примерно так, как представлено в листинге 7.6.

Листинг 7.6. Объявление конструктора и деструктора класса `TAutomobile`

```
type TAutomobile= class(TObject)
  ...
  constructor Create;  //конструктор объекта
  destructor Destroy;  //деструктор объекта
end;

constructor TAutomobile.Create; //создание автомобиля
begin
  fEngine:=TEngine.Create;    //создаем двигатель
  fEngine.SetEnabled(False); //в исходном состоянии двигатель выключен
  fSpeed:=0;                  //скорость автомобиля 0 км/ч
end;

destructor TAutomobile.Destroy; //уничтожение автомобиля
begin
  fSpeed:=0;         //автомобиль должен быть остановлен
  fEngine.Destroy;  //уничтожаем двигатель
end;
```

В коде конструктора мы создаем двигатель автомобиля и присваиваем ссылку на него в поле `fEngine`. Затем мы устанавливаем параметры автомобиля в состояние по умолчанию: двигатель выключен, скорость равна 0. Во время вызова деструктора мы решаем обратную задачу — останавливаем автомобиль и уничтожаем двигатель.

Если программист по каким-то причинам отказался от описания конструктора и деструктора разрабатываемого класса, то создание и уничтожение объекта будет осуществлено за счет конструктора и деструктора родительского класса. В нашем примере мы не стали создавать индивидуальные методы `Create()` и `Destroy()` для двигателя `TEngine`. Это означает, что управление существованием двигателя будет осуществляться за счет унаследованных им методов класса-предка (в нашем случае `TObject`).

> ***Примечание***
>
> Конструктор `Create()` относится к группе так называемых *методов класса* (`class functions`). Это особая категория методов, которые могут вызываться не от имени объекта, а от имени его класса.

Практический аспект применения конструктора и деструктора рассмотрен в листинге 7.7.

Листинг 7.7. Создание и уничтожение объекта в коде программы

```
var A : TAutomobile;   //объектная переменная
begin
  A:=TAutomobile.Create; //создаем объект
  ...                    //операции с объектом
  A.Destroy;             //разрушаем объект
end;
```

Программист объявляет переменную объектного типа, конструирует этот объект, проводит с созданным объектом запланированные операции и затем уничтожает его. Зачастую обязанности по созданию и уничтожению объектов берет на себя Delphi, так среда программирования в состоянии взять на себя полную ответственность за создание формы и принадлежащих ей компонентов при разработке приложения средней сложности. Но если программист намерен создавать объекты самостоятельно, то не стоит забывать ключевое правило объектов: все созданное нашими руками по окончании работы должно быть уничтожено нами же.

Опережающее объявление класса

Разрабатывая даже столь простую имитацию транспортного средства, нам все равно хочется достичь схожего поведения класса `TAutomobile` и реального автомобиля. Именно из-за этого наиболее интересной частью кода стал метод `SetSpeed()`, управляющий скоростью машины. Благодаря программной логике, заложенной в упомянутый метод (см. листинг 7.5), мы не позволим набрать скорость автомобилю, если у него не включен двигатель. Предлагаю поставить перед собой еще одну задачу — при выключении двигателя скорость автомобиля должна упасть до 0.

Если вы ненадолго вернетесь к листингу 7.3, в котором представлены методы класса `TEngine`, то вы сразу вспомните, что за остановку двигателя отвечает процедура `SetEnabled()` с аргументом `false`. На первый взгляд может показаться, что мы находимся в одном шаге от решения поставленной задачи — надо лишь дополнить метод `SetEnabled()` строкой кода

```
if Value=false then fAutomobile.fSpeed:=0;
```

Однако на практике это невозможно. Причина проста — класс `TEngine` не имеет ни малейшего представления даже о существовании класса `TAutomobile`, не говоря уже о его полях и методах. Поэтому упоминание поля автомобиля `fSpeed` в методе двигателя приведет к од-

ному — Delphi скажет, что идентификатор не объявлен (undeclared identifier) и откажется компилировать проект.

Какой выход из создавшегося положения? Ответ таков: надо сделать так, чтобы двигатель знал, в какую машину он установлен. Для этого я введу в состав класса `TEngine` еще одно поле — `fAutomobile:TAutomobile`. Однако в нашем листинге класс `TAutomobile` объявлен после класса `TEngine`, поэтому двигатель по-прежнему не понимает, что такое `TAutomobile`. Для устранения этой проблемы воспользуемся еще одной хитростью Delphi — *опережающим объявлением класса*.

Суть идеи опережающего объявления заключается в том, что мы заявим о намерении описать класс `TAutomobile` практически в первых строках листинга, а реальное описание оставим на прежнем месте (после описания двигателя). Благодаря такой уловке класс `TEngine` перестанет отвергать поле `fAutomobile`.

Листинг 7.8 содержит усовершенствованное объявление классов `TEngine` и `TAutomobile`.

Листинг 7.8. Улучшенный способ объявления классов `TEngine` и `TAutomobile`

```
{начало объявлений, одно ключевое слово type для TEngine и TAutomobile}
type
  TAutomobile=class; //опережающее объявление класса TAutomobile

{полное объявление класса TEngine}
  TEngine=class
    fEnabled :boolean;
    fAutomobile :TAutomobile; //автомобиль, в который установлен двигатель
    //другие поля и методы класса
  end;

{полное объявление класса TAutomobile}
  TAutomobile=class(TObject)
    fSpeed:single;   //скорость движения
    //другие поля и методы класса
  end;
```

Обязательно обратите внимание на небольшое изменение в порядке объявления классов. Раньше (см. листинги 7.2 и 7.3) описание каждого из классов начиналось с отдельного ключевого слова `type`, в новой версии кода для обеспечения видимости опережающего объявления класса `TAutomobile` мы ограничились одним словом `type`.

Еще одна синтаксическая особенность опережающего объявления в том, что заявление о существовании класса заканчивается не ключевым словом `end`, как того требует синтаксис (см. листинг 7.1). Вместо этого *точка с запятой* ставится *сразу* после инструкции `class`.

Добившись того, что класс `TEngine` узнал о существовании класса `TAutomobile`, усовершенствуем конструктор автомобиля (листинг 7.9).

Листинг 7.9. Исправление конструктора класса `TAutomobile`

```
constructor TAutomobile.Create;
begin
  fEngine:=TEngine.Create;    //создаем двигатель
```

```
  fEngine.fAutomobile:=self;  //двигатель получил ссылку на автомобиль
  fEngine.SetEnabled(False);  //исходное состояние — двигатель выключен
end;
```

Теперь, в момент создания автомобиля, ссылка на экземпляр класса `TAutomobile` будет передана в соответствующее поле экземпляра класса `TEngine`. Благодаря ссылке двигатель получает возможность обращаться к полям и методам автомобиля.

Нам осталось сделать последний штрих — дополнить метод включения/отключения двигателя `SetEnabled()` командой на остановку автомобиля (листинг 7.10).

Листинг 7.10. Остановка автомобиля в момент выключения двигателя

```
procedure TEngine.SetEnabled(Value: Boolean);
begin
  fEnabled:=Value;
  if Value=false then
    if Assigned(fAutomobile) then {если есть связанный объект}
      fAutomobile.fSpeed:=0;
end;
```

Ограничение видимости членов класса

При описании класса программист имеет право определять степень доступности (видимости) его полей, свойств и методов. Это один из способов защиты наиболее критичных элементов класса от несанкционированного вмешательства сторонних разработчиков. Область видимости поля (метода) класса зависит от того, в какой из четырех возможных секций оно объявлено: `private`, `protected`, `public` и `published` (листинг 7.11).

Листинг 7.11. Секции видимости членов класса

```
type
  TAutomobile = class
  private
    ... { секция частных объявлений }
  protected
    ... { секция защищенных объявлений }
  public
    ... { секция общих объявлений }
  published
    ... { секция опубликованных объявлений }
end;
```

Поля и методы, доступ к которым нежелателен, обычно размещаются в секциях `private` и `protected`. Наиболее защищена секция `private`. К размещенным в ней полям и методам возможно обращение только из того же программного модуля, в котором описан этот класс. Секция `protected` несколько приоткрывает завесу секретности — находящаяся в ней информация без каких-либо ограничений доступна для классов-потомков. Секция `public` предоставляет объявленные в ней поля и методы для общего пользования всем желающим. Секция `published` самая доброжелательная. Например, объявленными в ней данными умеет

пользоваться Инспектор объектов. Это возможно благодаря тому, что для этой секции класса генерируется *информация о времени выполнения* (Run Time Type Information, RTTI). Поэтому в секции `published` обычно объявляют все свойства и обработчики событий объекта.

> ***Примечание***
>
> В Delphi предусмотрен еще более строгий способ ограничения видимости членов класса. Он обеспечивается путем введения секций `strict private` и `strict protected`. Все члены класса, объявленные в границах секции `strict private`, будут доступны лишь классу-владельцу секции. Объявления в рамках `strict protected` сохранят видимость и в классах потомках при условии, что они описаны в том же модуле, где и родительский класс.

Свойства объекта

Как правило, объект не должен предоставлять прямой доступ к своим полям. Это требование инкапсуляции, благодаря которому поддерживается целостность объекта. Поэтому при проектировании класса одновременно с полями объявляют особых посредников, называемых *свойствами* (properties).

В первую очередь в обязанность свойств входит проверка корректности передаваемых в поля данных, для этого они могут задействовать всю мощь языка Delphi. В самом простейшем случае синтаксическая конструкция объявления свойства выглядит следующим образом:

```
property Имя_свойства: Тип_свойства read (способ_чтения)
                                    write (способ_записи);
```

За инструкциями `Read` и `Write` могут следовать названия процедур, соответственно отвечающих за чтение данных из поля и запись данных в поле.

Возьмем в качестве примера класс двигателя `TEngine`, он обладает парой полей `fEnabled` и `fAutomobile`, которые следует обязательно защитить. Листинг 7.12 демонстрирует один из возможных сценариев усовершенствования класса.

Листинг 7.12. Объявление свойств

```
type TEngine=class
  private
    fEnabled :boolean; //двигатель включен — true, выключен — false
    fAutomobile:TAutomobile; //ссылка на автомобиль
  public
    procedure SetEnabled(Value:Boolean); //запуск или останов двигателя
    function  GetEngineState:boolean;    //состояние двигателя
  published
    property Enabled:boolean read fEnabled write SetEnabled;
    property Automobile:TAutomobile read fAutomobile;
end;
```

Все поля класса "спрятаны" в секцию `private`, что существенно усложнит обращение к полям извне. Наоборот, методы и свойства, к которым мы намерены разрешить полный доступ, размещены в секции публичных объявлений. Теперь для включения/отключения двигателя надо воспользоваться свойством `Enabled`. Еще большую степень защиты мы обеспечили полю `fAutomobile`, хранящему ссылку на автомобиль, в который установлен двигатель — свойство допускает только чтение информации из поля.

Особенности объявления методов

На первых страницах главы, разъясняя понятие метода, мы говорили, что это процедура или функция, управляющая поведением объекта. До сих пор такое упрощение было оправданным и не мешало процессу изучения классов. Но перед тем как мы начнем разговор о наследовании и полиморфизме, стоит расширить свои знания о методах.

Хотя по своей сути метод является процедурой или функцией, в объектно-ориентированном программировании в него вкладывается значительно более глубокий смысл. Концепция ООП предполагает возможность класса наследовать методы своего предшественника, а при необходимости и переопределять их поведение. Для обеспечения этой, очень сложной задачи в Delphi методы разделяются на три категории:

- статические (директива `static`);
- динамические (директива `virtual`);
- виртуальные (директива `dynamic`).

По умолчанию любой вновь описанный метод становится *статическим*. Такое название метод заслужил из-за того, что во время компиляции статическому методу выдается постоянный адрес, такой способ называется *статическим связыванием* (static binding). У метода с постоянным адресом есть две особенности: высокая скорость вызова (что не может не радовать) и абсолютная невозможность изменить поведение этого метода в дочерних классах (что является отрицательной стороной статического связывания). Одним словом, объявляя статический метод, мы должны понимать, что наносим удар по полиморфизму (наследник класса утратит возможность переопределить этот метод).

В отличие от своего статического собрата *виртуальные* и *динамические* методы исповедуют идеи *позднего связывания* (late binding). Это означает, что адреса таких методов не "высечены в граните", а могут изменяться. Так адреса виртуальных методов класса заносятся в таблицу виртуальных методов этого класса, а адреса динамических методов — в специальный список методов этого класса. Благодаря технологии позднего связывания доступ к виртуальным и динамическим методам осуществляется несколько медленнее, но зато они допускают переопределение своего поведения у классов-потомков.

> ***Внимание!***
>
> Для того чтобы поведение метода могло быть переопределено в дочернем классе, он должен объявляться с директивой `virtual` или `dynamic`.

Поля класса и методы класса

Одно из правил объектно-ориентированного программирования предполагает, что обращение к полям и методам, описанным в классе, невозможно до тех пор, пока из класса не будет создан объект — экземпляр класса. Это вполне логичное требование, ведь никому в реальном мире и в голову не придет прокатиться на чертеже (читай — классе) автомобиля. Сначала следует превратить чертеж в реальный автомобиль и только потом давить на педаль газа. С точки зрения программы Delphi это означает, что начало работы со свойствами и методами объекта должно предваряться вызовом его конструктора `Create()`, в противном случае отладчик среды разработки отправит нам неприятное сообщение, содержащее фразу "Access Violation" ("Нарушение доступа"), и прекратит выполнение программы.

Однако из правил существуют исключения. В состав класса могут входить особые поля и методы, которые остаются доступными даже в отсутствие экземпляра объекта. Такие поля и

методы называют соответственно *полями класса* (class fields) и *методами класса* (class methods).

По своей сути поле класса представляет собой обычную переменную, для ее объявления следует воспользоваться ключевыми словами `class var` или просто `var`.

Метод класса представляет собой процедуру или функцию с первым словом в заголовке `class`, например `class procedure` или `class function`.

Благодаря полю класса и методу класса мы можем внести очередное усовершенствование класса двигателя `TEngine` (листинг 7.13).

Листинг 7.13. Объявление поля класса и метода класса `TEngine`

```
TEngine=class
  private
    fEnabled :boolean;
    fAutomobile:TAutomobile;
  public
    class var fMax_HorsePower : word; //переменная класса
    class function GetMax_HorsePower : word; //метод класса
...
end;
{реализация функции класса}
class function TEngine.GetMax_HorsePower: word;
begin
  Result:=fMax_HorsePower;
end;
```

Как и утверждалось ранее, для обращения к полям и методам класса совсем не обязательно создавать объект, вполне допустимо воспользоваться строками кода из листинга 7.14.

Листинг 7.14. Обращение к полю класса и методу класса `TEngine`

```
var X:Word;
begin
  with TEngine do
    begin
      fMax_HorsePower:=250;
      X:=GetMax_HorsePower;
    end;
end;
```

Иерархия наследования

Вновь возвратимся к очень важному ингредиенту ООП — наследованию. Мы уже знаем, что в основе концепции наследования лежит возможность строительства нового класса на базе уже существующего родительского класса. Благодаря наследованию дочерний класс впитывает в себя все родовые черты своего предка. Ограничений на длину цепочки наследования нет, в визуальной библиотеке компонентов Delphi объявлены классы, имеющие в

родословной больше десятка предков. Таким образом, самый последний в древе наследования класс вбирает в себя все лучшее от всех своих предшественников.

Для того чтобы понять какие уникальные возможности предоставляет программисту наследование, перейдем к практической стороне вопроса. В нашем распоряжении уже имеется класс `TAutomobile`, олицетворяющий все наши знания в области ООП. Попробуем развить первоначальный успех и создать программный класс, способный смоделировать пассажирский автобус — класс `TAutobus`.

Автобус — это всего лишь разновидность автомобиля. У него, как и у автомобиля, есть двигатель, автобус также может набирать и снижать скорость движения. Одним словом, за основу нашего нового класса можно взять уже существующий `TAutomobile` и дополнить его всего одной возможностью — перевозить пассажиров (листинг 7.15).

Листинг 7.15. Создание дочернего класса `TAutobus`

```
type TAutobus=class(TAutomobile)
  private
    fPassenger:byte; //поле — число пассажиров
    procedure SetPassenger(Value:byte); //управление числом пассажиров
  public
    property Passenger:byte read fPassenger write SetPassenger;
end;

procedure TAutobus.SetPassenger(Value: byte);
begin
  if (Value>=0) and (Value<=30) {ограничение числа пассажиров}
    then  fPassenger:=Value
  else Raise Exception.Create('Число пассажиров должно быть от 0 до 30!');
end;
```

Полиморфизм

Концепция наследования порождает еще одну полезную черту классов — полиморфизм. Идея полиморфизма тесно связана с наследованием и заключается в праве экземпляра некоторого класса представлять все классы из его иерархической цепочки. Более того, благодаря полиморфизму мы получаем право скрывать (инструкция `override`) или переопределять (инструкция `inherited`) поведение унаследованных методов. Поэтому различные по содержанию процедуры и функции всевозможных объектов могут использовать одно и то же имя. Вызов же метода будет приводить к выполнению кода соответствующего конкретному экземпляру объекта.

Листинг 7.16 поможет нам разобраться с основными принципами полиморфизма, он представляет собой модуль с цепочкой классов-самолетов.

Листинг 7.16. Цепочка классов `TAirframe`, `TAircraft` и `TJetAircraft`

```
unit Polymorph;

interface
uses SysUtils, Dialogs;
```

```
type
  TAirframe = class //абстрактный класс планер
    procedure Fly; virtual; abstract;
  end;

  TAircraft = class(TAirframe)  //самолет
    procedure Fly; override;    //перезапись метода
  end;

  TJetAircraft = class(TAircraft) //реактивный самолет
    procedure Fly; override;      //перезапись метода
  end;

implementation

  procedure TAircraft.Fly;
  begin
    ShowMessage('Aircraft Fly');
  end;

  procedure TJetAircraft.Fly;
  begin
    {позднее этот участок кода будет доработан}
    ShowMessage('JetAircraft Fly');
  end;
end.
```

Первый из классов абстрактный. Это означает, что из него нельзя создать самостоятельный объект, но в этом и нет необходимости, задача планера `TAirframe` заключается в определении родовой черты всех своих потомков. Это, ни много ни мало, отвечающий за полет метод `Fly()`, который должен быть обязательно объявлен и заполнен программной логикой у всех наследников `TAirframe` — летательных аппаратов, классов `TAircraft` и `TJetAircraft`.

Для того чтобы классы `TAircraft` и `TJetAircraft` получили право пользоваться одноименным методом `Fly()`, после объявления его заголовка использована специальная директива `override`, аннулирующая метод класса-предка.

> ***ВНИМАНИЕ!***
>
> Директива `override` переопределяет метод предка в новом классе. Причем метод имеет такое же имя и практически такое же описание, как у родителя, однако его поведение у всей цепочки наследников полностью изменяется.

В результате в момент обращения к процедуре `Fly()` класса `TAircraft` выводится окно с текстовым сообщением "Aircraft Fly", процедуре `Fly()` класса `TJetAircraft` — "JetAircraft Fly". Одним словом, каждый из самолетов "летает" по-своему. Для того чтобы в этом убедиться, можно воспользоваться листингом 7.17.

Листинг 7.17. Вызов метода `Fly()` экземпляра класса `TJetAircraft`

```
with TJetAircraft.Create do
begin
  Fly;
  Destroy;
end;
```

В ответ на вызов фрагмента кода (листинг 7.17) мы получим сообщение "JetAircraft Fly".

А теперь настал час провести эксперимент с полиморфизмом, но для этого придется внести незначительное исправление в реализацию метода `Fly()` класса реактивного самолета `TJetAircraft`. Поэтому мы добавим в программный модуль всего одну строку с инструкцией `inherited` (листинг 7.18).

Листинг 7.18. Усовершенствование метода `Fly()` класса `TJetAircraft`

```
unit Polymorph;
...
  procedure TJetAircraft.Fly;
  begin
    inherited;
    ShowMessage('JetAircraft Fly');
  end;
end.
```

Инструкция `inherited` уведомляет Delphi о том, что при вызове метода `Fly()` класс `TJetAircraft` намерен воспользоваться услугами одноименного метода `Fly()` своего класса-предка `TAircraft`.

> ***ВНИМАНИЕ!***
>
> Всю мощь полиморфного наследования активирует инструкция `inherited`. Она требует, чтобы в распоряжение класса-потомка был передан соответствующий метод, унаследованный у класса-предка.

Если вы после внесения исправлений вновь вызовите предложенный в листинге 7.17 код, то на экране отобразится не одно, а целых два окна сообщений:

- сообщение "Aircraft Fly" (вызванное методом `Fly()` класса-предка `TAircraft`);
- сообщение "JetAircraft Fly" (принадлежащее экземпляру класса `TjetAircraft`).

> ***ПРИМЕЧАНИЕ***
>
> В ситуации, когда родительский класс намерен скрыть от своих потомков свой виртуальный или динамический метод, следует воспользоваться директивой `reintroduce`.

Операторы класса

В объектно-ориентированном языке Delphi имеются два очень сильных оператора — `is` и `as`, которые мы в дальнейшем станем называть *операторами класса*. Первый из них предназначен для проверки принадлежности объекта к определенному классу.

```
If (MyObject is TAircraft)=True then MyObject.Fly;
```

Ключевая особенность оператора `is` состоит в том, что он проверяет не только непосредственного предка объекта, а всю иерархическую цепочку наследования объекта в целом. И если `TAircraft` будет предком объекта `MyObject` даже в десятом колене, то конструкция вернет `true`.

Оператор as называется *оператором приведения типа*. Это очень грозное и одновременно опасное оружие, и поэтому использование оператора должно быть максимально продуманным.

```
with (MyObject as TJetAircraft) do ...
```

Оператор приведения типа позволяет явным образом указывать Delphi, что объект MyObject должен действовать как экземпляр класса TJetAircraft. И горе той программе, которая с помощью оператора as попробует заставить объект выполнить несвойственные ему действия. Такая попытка заканчивается вызовом исключительной ситуации.

> ***ЗАМЕЧАНИЕ***
>
> При желании можно воспользоваться сокращенной формой приведения типа, отказавшись от явного упоминания оператора as — with TJetAircraft(MyObject) do.

Аннотация класса

В последних версиях Delphi появилась возможность добавления к любому существующему классу (или к отдельному члену класса) дополнительной информации — своего рода аннотации поля, метода, свойства или класса в целом.

Метаинформация ни в коем случае не изменяет структуру класса — это просто дополнительная информация, которая (начиная с Delphi 2010) во время компиляции проекта добавляется в особую секцию исполняемого файла. По мере необходимости, во время выполнения программы средствами RTTI (Runtime Type Information) аннотация класса извлекается из бинарного кода исполняемого файла.

Создание и подключение атрибутов с аннотацией

В терминах языка Delphi XE2 добавляемые к классу (или элементу класса) метаданные называются *атрибутами*. Хотя по своей сути атрибуты являются простыми переменными, в которые будет заноситься та или иная информация, в объектно-ориентированном языке Delphi атрибуты создаются на основе специального класса TCustomAttribute (который, в свою очередь, является надстройкой над опорным классом библиотеки компонентов TObject).

Работа с атрибутами начинается с небольшого подготовительного этапа, на котором производится описание нового класса хранилища атрибутов (потомка TCustomAttribute). В секции публичных объявлений public класса определяются поля класса, именно они и станут хранить метаданные (листинг 7.19).

Листинг 7.19. Объявление класса-контейнера атрибутов

```
type
  TDemoAttribute = class(TCustomAttribute)
  public
    fCopyright,fOtherInfo: String; //2 поля для текстовых метаданных
    fDateTime:TDateTime; //поле в формате "дата/время"
    constructor Create(const AValue: String);
  end;
```

```
{конструктор класса}
constructor TDemoAttribute.Create(const AValue: String);
begin
  fCopyright:='(c) 2011, D. Osipov'; //разработчик
  fDateTime:=Now; //атрибут с датой/временем создания объекта
  fOtherInfo := AValue; //дополнительные данные
end;
```

В нашем примере мы определяем два атрибута: поля `fCopyright` и `fOtherInfo` строкового типа и атрибут `fDateTime`, предназначенный для хранения значения даты/времени. Для того чтобы заполнить атрибуты метаданными, задействуем конструктор класса. В момент создания экземпляра класса в поле `fCopyright` заносятся сведения о разработчике класса, в поле `fDateTime` сохраняется метка со временем создания экземпляра класса, в поле `fOtherInfo` поступит внешнее текстовое значение из параметра конструктора `AValue`.

Порядок подключения атрибутов к классу (в нашем случае разработанному в первой половине главы классу автомобиля) раскрыт в листинге 7.20.

Листинг 7.20. Подключение атрибутов к классу

```
type [TDemoAttribute('Class annotation')]  //подключение атрибутов
      TAuromobile=class
      //поля и методы класса
end;
```

Обратите внимание на синтаксис подключения атрибутов `TDemoAttribute` к классу `TAuromobile`. Имя класса-хранилища атрибутов `TDemoAttribute` заключается в квадратные скобки и располагается перед названием класса, к которому будет присоединена аннотация. Два атрибута заполняются автоматически в момент вызова конструктора, а атрибут `fOtherInfo` получит внешние данные, в примере из листинга 7.20 это будет текст "Class annotation".

Атрибуты могут подключаться не только к классу в целом, но и к любому члену класса. Так листинг 7.21 демонстрирует способ подключения атрибутов к методу класса.

Листинг 7.21. Подключение атрибутов к методу класса

```
type TAutomobile=class
  //...
  [TSecondDemoAttribute]
  Function SetSpeed(Value:single):single;  //установить скорость движения
end;
```

> ***Примечание***
>
> Delphi позволяет подключить аннотацию не только к классу (членам класса), но и к другим структурам, в частности к записям и простым типам данных.

Извлечение аннотации

Для того чтобы добраться до атрибутов класса и прочитать хранящуюся в них информацию во время выполнения приложения, необходима помощь RTTI. Мы не станем углубляться в

тонкости работы с информацией о типах выполнения. Отметим лишь то, что, начиная с Delphi 2010, RTTI основательно переработан, и в состав библиотек языка программирования включен модуль RTTI.pas, помощью которого мы воспользуемся в листинге 7.22.

Листинг 7.22. Получение информации RTTI об атрибутах класса

```
uses ..., RTTI; {необходимо подключить модуль RTTI}

procedure GetClassAttributes(const AClass: TClass);
var rContext: TRttiContext;
    rType: TRttiType;
    rAttr: TCustomAttribute;
begin
  rContext := TRttiContext.Create; {создаем контекст RTTI}
  try
    rType := rContext.GetType(AClass); {получаем информацию о типах RTTI}
    WriteLn(rType.Name); {выводим в консоль название класса}

   for rAttr in rType.GetAttributes() do {просмотр атрибутов}
     if rAttr is TDemoAttribute then {выбираем только наши атрибуты}
       begin
         WriteLn(TDemoAttribute(rAttr).fCopyright);
         WriteLn(DateTimeToStr(TDemoAttribute(rAttr).fDateTime));
         WriteLn(TDemoAttribute(rAttr).fOtherInfo);
       end;

  finally
    rContext.Free;  {освобождаем контекст}
  end;
end;
```

Предложенная процедура предназначена для работы в консольном приложении, ее единственный входной параметр служит для получения ссылки на класс, атрибуты которого мы намерены просмотреть. Таким образом, для того чтобы изучить подключенную к классу `TAutomobile` метаинформацию, достаточно написать одну строку кода:

```
GetClassAttributes(TAuromobile);
```

> ***Замечание***
>
> Благодаря механизму подключения атрибутов с метаданными к классам, программы Delphi приобрели возможность поддерживать новую парадигму *аспектно-ориентированного* (aspect-oriented) программирования (АОП). Базовая идея АОП заключается в возможности выделения функциональности классов в отдельные сущности, что, в частности, позволит разрабатывать программы на основе идеи контрактного программирования, упрощает трассировку программ, повышает уровень безопасности программного обеспечения.

ГЛАВА 8

Шаблоны

Еще в Delphi 2009 в составе языка появилась поддержка *шаблонов* или, говоря точнее, *обобщенных типов* (generics types) данных[1]. Прежде обобщения существовали преимущественно только в средах программирования ориентированных на технологию .NET и в языке C++, так что для классической Delphi это новинка.

Своим появлением на свет шаблоны обязаны желанием разработчиков развить возможности ООП. Именно в языках программирования, основанных на ООП, код приобрел истинную универсальность и стал позволять программистам повторно использовать первоначально разработанный алгоритм для решения новых задач. Создав исходный класс для применения в какой-то прикладной задаче, мы имеем возможность перенести свой класс (или его потомков) в другой проект. Однако обычные классы, пусть даже усиленные за счет наследования, инкапсуляции и полиморфизма далеко не универсальны, ведь их методы и свойства нацелены на обслуживание строго определенных типов данных. Концепция обобщенного типа данных как раз и предназначена для устранения этого недостатка — она позволяет программисту абстрагироваться от типа данных. Благодаря шаблонам разработчик получает возможность создавать алгоритмы, не указывая типы данных, с которыми он работает. Услуги обобщенного типа данных окажутся весьма кстати при описании схожих алгоритмов, занимающихся обработкой разноплановых данных. Например, это позволяет реализовать классические алгоритмы сортировки массивов самых разнообразных типов данных в рамках одного и того же метода класса.

> ***Замечание***
>
> Благодаря шаблонам мы получаем удобную возможность отделения алгоритма от конкретных типов данных, с которыми он работает.

Обобщенный тип данных в полях записей

Основные приемы работы с обобщенным типом данных наиболее просто продемонстрировать с помощью записи Delphi. Еще в первых версиях языка Pascal появилась запись — структурный тип, специализирующийся на предоставлении доступа к некоторому перечню разнотипных полей. В современных версиях Delphi программисту представилась уникальная возможность включать в состав записи не только строго типизированные поля, но и

[1] В некоторой литературе вместо термина "обобщенный тип" применяют термин "родовой тип".

поля-обобщения, т. е. поля, тип данных которых заранее не определен. Описание шаблона записи с полями обобщенного типа выглядит следующим образом:

```
type имя_типа_записи <имя_обобщенного_типа[,...]> = record
  имя_поля: имя_обобщенного_типа;
  [Другие поля]
end;
```

Наличие угловых скобок уведомляет нас, что запись намерена работать с обобщенным типом данных. Внутри угловых скобок перечисляются *параметры типов* (type parameters). С одной стороны, параметр типа — это имя переменной, которое мы задействуем внутри исходного кода класса в тех местах, где используется соответствующий тип данных. С другой стороны, параметр типа — это псевдоним типа данных, которым воспользуется программист при обращении к классу извне.

> ***Примечание***
>
> В синтаксических конструкциях языка Delphi угловые скобки <...> являются визитной карточкой обобщенного типа данных.

Отметим, что при определении имени обобщенного типа нет необходимости вспоминать о существовании предопределенных типов данных Delphi. Имя обобщенного типа — это всего лишь псевдоним какого-то заранее неизвестного типа данных, во время выполнения программы им может оказаться целое или вещественное число, строковый или символьный тип либо какой-то другой тип данных.

Листинг 8.1 демонстрирует порядок объявления шаблона записи с полями обобщенного типа. Перечень имен, используемых в записи обобщенных типов данных, приводится внутри угловых скобок, сразу за названием типа записи.

Листинг 8.1. Объявление шаблона записи с полями обобщенного типа

```
type TGenericRec<T,R>= record
  A    : char; //в любом случае это символьное поле
  B,C  : T;    //поля обобщенного типа T
  D    : R;    //поле обобщенного типа R
end;

var GRec1:TGenericRec<integer, double>; //объявление переменной GRec1
    GRec2:TGenericRec<boolean, string>; //объявление переменной GRec2
begin
  GRec1.B:=15;    //поле B хранит целое значение
  GRec1.D:=0.675; //поле D хранит вещественное значение
  //...
  GRec2.B:=true; //поле B хранит булев тип данных
  GRec2.D:='Hello, World!'; //в поле D находится строка с текстом
 //...
end.
```

Во время инициализации переменных имена обобщенных типов заменяются названиями реальных типов данных. В нашем примере при объявлении `GRec1` (см. листинг 8.1) вместо обобщения `T` мы задействуем целочисленный тип `integer`, вместо `R` — вещественный тип

double. При объявлении GRec2 мы меняем правила игры: теперь обобщенный тип T замещается типом boolean, R — string.

Обобщения в процедурах и функциях

Если абстрагироваться от типа данных, окажется, что логика очень многих алгоритмов абсолютно одинакова. Например, алгоритм быстрой сортировки идентичен как для массива целых, так и для массива вещественных чисел. В подобных ситуациях благодаря созданию обобщенной функции (функции с параметром обобщенного типа) мы сможем определить смысл алгоритма, не отвлекаясь на обслуживаемый тип данных. Поэтому обобщенный тип данных может оказаться весьма полезным при разработке шаблонов процедурных типов данных, способных воспринимать параметры любых типов.

По аналогии с объявлением обычной процедуры или функции, при создании шаблона процедуры или функции сначала указывают ключевое слово type, а затем имя типа. Сразу за именем следует синтаксический элемент, характерный для обобщений — пара угловых скобок <...> с названием обобщенного типа данных.

```
type тип_функции <имя_обобщенного_типа[,...]> =
        function([параметр_обобщенного_типа [,...]]):имя_обобщенного_типа;
type тип_процедуры <имя_обобщенного_типа [,...]>  =
        procedure([параметр_обобщенного_типа [,...]]
```

Листинг 8.2 демонстрирует порядок создания шаблона функции. В примере объявлена функция класса GetMaxValue() с двумя параметрами, тип которых не определен. Задачей функции является сравнение двух значений и возврат наибольшего из них.

Листинг 8.2. Объявление шаблона процедурного типа

```
uses System.SysUtils, System.Generics.Defaults;

type TGetMax = class
    class function GetMaxValue<T>(X, Y: T):T;
end;

class function TGetMax.GetMaxValue<T>(X, Y: T): T;
var Comparer: IComparer<T>;
begin
 {к сожалению, нельзя написать такой код:
  if X>Y then Result:=X else Result:=Y;
  ведь мы не знаем тип данных сравниваемых параметров}
  Comparer:=TComparer<T>.Default;
  if Comparer.Compare(X,Y)>0 then Result:=X
                             else Result:=Y;
end;

var A:integer;
    B:extended;
    C:ansichar;
begin
  A:=TGetMax.GetMaxValue<integer>(10,20); //сравниваем целые числа
  WriteLn(A);
```

```
  B:=TGetMax.GetMaxValue<extended>(1/3,2/3); //вещественные числа
  WriteLn(B);

  C:=TGetMax.GetMaxValue<ansichar>('A','B');//сравниваем символы
  WriteLn(C);

  ReadLn;
end.
```

Реализация обобщенной функции требует ряда комментариев. К сожалению, сравнивая данные неизвестного типа, мы не имеем возможности воспользоваться обычными операторами сравнения (например: **if** `X<Y` **then** ...). Вместо этого мы вынуждены обратиться за услугами к интерфейсу `IComparer` (он описан в модуле `System.Generics.Defaults`). Этот интерфейс (кстати, также основанный на идее шаблонов) является профессионалом в области сравнения двух величин, для этого у него есть метод `Compare()`. Все остальное дело техники — в листинге 8.2 мы сравнили значения трех различных типов данных.

Обобщенные типы данных в шаблонах классов

Синтаксическая конструкция, объявления шаблона класса, способного работать с обобщенным типом данных, не многим отличается от унаследованной еще со времен языка Object Pascal конструкции описания обычного класса. Здесь также присутствуют ключевые слова `type` и `class`, и класс-шаблон, так же как и его обычный "коллега", содержит поля и методы. Однако внимательный читатель обнаружит отличительную черту шаблона — пару угловых скобок <...> сразу после имени класса:

```
type имя_класса<параметр_обобщенного_типа [, ...]> = class
  //поля и методы
end;
```

Простейший пример использования обобщенного типа при объявлении шаблона класса предложен в листинге 8.3.

Листинг 8.3. Объявление и реализация шаблона класса

```
unit generics_unit;

interface
type TGenericClass<T>=class    //объявление шаблона класса
    fT:T;                      //поле обобщенного типа
    procedure SetT(Value:T);   //метод записи в поле
    function GetT:T;           //метод чтения значения из поля
end;

implementation

function TGenericClass<T>.GetT: T;
begin
  Result:=fT; //узнаем содержимое поля
end;
```

```
procedure TGenericClass<T>.SetT(Value: T);
begin
  fT:=Value; //записываем данные в поле
end;
end.
```

Для объявления параметра типа мы воспользовались символом `T` и заключили его в угловые скобки. Класс содержит единственное поле `fT` универсального типа `T` и два метода, позволяющих записывать и считывать значение из поля.

Разобравшись с объявлением и реализацией шаблона класса, способного работать обобщенным типом данных, попробуем воспользоваться его услугами. Листинг 8.4 демонстрирует порядок обращения к экземпляру класса `TGenericClass` с указанием того, что он необходим нам в роли хранилища целого числа.

Листинг 8.4. Применение шаблона класса

```
var gObject: TGenericClass<integer>;
    i:integer;
begin
  gObject:= TGenericClass<integer>.Create;

  gObject.SetT(99); //записываем значение 99
  i:=gObject.GetT;  //считываем значение из класса

  gObject.Destroy;
end;
```

Ключевой элемент листинга 8.4 — это его первая строка. В ней указывается, какой тип данных должен будет задействован после создания экземпляра класса `TGenericClass`. Название типа записывается в угловых скобках после имени класса, в нашем случае это целое число `integer`. С таким же успехом мы можем создать объект любого типа, за небольшим исключением. В параметр типа не стоит передавать статистические массивы, строки старого образца (`short string`) и записи, содержащие разнотипные поля.

Наследование шаблона класса

Любой, построенный на основе концепции обобщенного типа данных, шаблон класса вправе выступать в качестве предка для своей цепочки классов-наследников. Объявляя дочерний класс, программист принимает решение о статусе параметра обобщенного типа. Предусмотрены два варианта наследования (листинг 8.5):

- наследование с сохранением обобщенного типа данных, другими словами, класс-наследник остается шаблоном;
- наследование с явной конкретизацией типа данных, который следует использовать при работе дочернего класса.

Листинг 8.5. Наследование шаблона класса

```
type TGenericParent<T>=class {родительский класс с открытым параметром T}
  protected
     fGP:T;
```

```
  public
     procedure SetGP(Value:T);
     function  GetGP:T;
end;
    {1 вариант наследования — объявляется новый класс-шаблон}
    TGenericChild<T>=class(TGenericParent<T>)
      //поля и методы класса TGenericChild1
    end;
    {2 вариант — объявляется обычный класс}
    TNotGenericChild<T>=class(TGenericParent<integer>)
      //поля и методы класса TNotGenericChild
    end;
```

В нашем примере описываются два дочерних класса. Первый из них, класс `TGenericChild`, допускает, что при обращении к свойствам и методам дочернего и родительского классов могут использоваться любые типы данных, поэтому `TGenericChild` сохраняет все черты шаблона.

Напротив, класс `TNotGenericChild` перестал быть шаблоном, он явным образом указывает, что класс-предок в состоянии воспринимать только целочисленные данные.

При описании дочернего класса-шаблона разрешается вводить дополнительные параметры, способные работать с обобщенным типом данных, например:

```
TGenericChild2<T, C>=class(TGenericParent<T>)
```

Таким образом, в дочернем классе появился дополнительный параметр обобщенного типа.

Перегрузка методов с параметром обобщенного типа

Наличие у метода класса параметра обобщенного типа не служит препятствием при перегрузке одноименных методов. Листинг 8.6 демонстрирует порядок объявления и перегрузки обычного метода и шаблона метода с параметром обобщенного типа.

Листинг 8.6. Примеры объявления перегружаемых методов

```
type TDemoClass = class
     procedure DemoMethod(X: double); overload;      //обычный метод
     procedure DemoMethod<T>(X: T);   overload;      //шаблон метода
end;
```

В распоряжении демонстрационного класса имеются два метода `DemoMethod()`. Параметр первого метода задан явным образом в виде действительного типа `double`, а параметр второго представлен в виде обобщенного типа `T`.

Листинг 8.7 показывает порядок вызова перегружаемых методов.

Листинг 8.7. Вызов перегружаемых методов

```
var SC:TDemoClass;
begin
  SC:=TDemoClass.Create;
```

```
  SC.DemoMethod(3.1426);       //обращение к DemoMethod(X: double)
  SC.DemoMethod<Integer>(500); //обращение к шаблону DemoMethod<T>(X: T)
//...
```

Обратите внимание на то, что при вызове шаблона метода с обобщенным параметром следует определиться с типом обрабатываемых данных, указав его в угловых скобках после названия метода.

> ***Замечание***
>
> Основные обобщенные типы данных VCL Delphi объявлены в модуле `Generics.Collections`. Здесь вы обнаружите шаблоны для таких структур данных, как: списки `TList<T>`, очереди `TQueue<T>`, стеки `TStack<T>`, словари `TDictionary<TKey,TValue>` и т. д.

Шаблон массива, класс *TArray*

В библиотеке VCL языка Delphi вы найдете немало классов, работающих с данными обобщенного типа. Давайте познакомимся с одним из наиболее показательных классов — шаблоном массива `TArray`. Класс-шаблон `TArray` описан в модуле `Generics.Collections` и предназначен для оказания помощи программистам при решении задач упорядочивания элементов массива по возрастанию значений и поиску данных в массиве.

Фрагмент объявления класса `TArray` предложен в листинге 8.8.

Листинг 8.8. Объявление класса `TArray` в модуле `Generics.Collections`

```
type TArray = class
  private
    class procedure QuickSort<T>(var Values: array of T;
                    const Comparer: IComparer<T>; L, R: Integer);
  public
    class procedure Sort<T>(var Values: array of T); overload;
    //другие версии процедуры сортировки Sort<T>

    class function BinarySearch<T>(const Values: array of T;
          const Item: T; out FoundIndex: Integer;
          const Comparer: IComparer<T>;Index,
                Count: Integer): Boolean; overload;
    //другие версии процедуры бинарного поиска BinarySearch<T>
end;
```

Обратите внимание на то, что класс не предназначен для хранения данных, для этого у него нет ни одного поля. Зато в распоряжении `TArray` имеются две группы одноименных методов-шаблонов. Методы `Sort<T>()` позволяют сортировать данные в массивах любого простого типа, а методы `BinarySearch<T>()` найдут в упорядоченном массиве индекс ячейки с заданным элементом. Все процедуры и функции объявлены в качестве методов класса (напомню, что об этом можно судить по слову `class` вначале объявления метода), а это означает, что методы подлежат вызову без создания экземпляра класса.

Воспользуемся услугами класса `TArray` для упорядочивания значений в трех разнотипных массивах (листинг 8.9). Для этого задействуем его метод `Sort<T>()`.

Листинг 8.9. Сортировка массивов с помощью шаблона класса `TArray`

```
program TArrayDemo;

{$APPTYPE CONSOLE}
uses System.SysUtils, System.Math, Generics.Collections;

const Size=10; //размер массивов

var A:Array[0..Size] of Integer; //объявляем массивы
    B:Array[0..Size] of Real;
    C:Array[0..Size] of AnsiChar;
    i:Word;
begin
  Randomize; //инициализируем генератор псевдослучайных чисел
  for i:=0 to Size do {заполним массивы случайными значениями}
  begin
    A[i]:=Random(100);                    //генерируем целое число
    B[i]:=Random(100)/RandomRange(1,100); //генерируем действительное число
    C[i]:=AnsiChar(RandomRange(33,255));  //генерируем символ
  end;

  TArray.Sort<integer>(A);   //упорядочим значения в массивах
  TArray.Sort<Real>(B);
  TArray.Sort<AnsiChar>(C);

  for i := 0 to Size do //проверим результат сортировки
  begin
    Write('A[',i,']=',#9,A[i],#9);
    Write('B[',i,']=',#9,B[i],#9);
    Write('C[',i,']=',#9,C[i],#9);
    WriteLn;
  end;
  ReadLn;
end.
```

Замечание

При разработке класса `TArray` программисты Embarcadero использовали одни из наиболее эффективных алгоритмов сортировки и поиска. Так упорядочивание элементов осуществляется с помощью метода, получившего название *быстрой сортировки* (quick sort), а поиск реализован методом *бинарного поиска* (binary search). Оба метода относятся к классу сложности алгоритмов с оценкой $N \cdot Log(N)$, это визитная карточка наиболее производительных алгоритмов.

ЧАСТЬ II

Платформа VCL

Глава 9. Опорные классы VCL
Глава 10. Массивы указателей, наборы строк и коллекции
Глава 11. Классы потоков данных
Глава 12. Визуальные элементы управления и класс *TControl*
Глава 13. Оконные элементы управления и класс *TWinControl*
Глава 14. Приложение VCL
Глава 15. Форма, фрейм и модуль данных
Глава 16. Исключительные ситуации
Глава 17. Компоненты отображения и редактирования текста
Глава 18. Кнопки и компоненты выбора значений
Глава 19. Меню приложения
Глава 20. Управление приложением с помощью команд
Глава 21. Списки
Глава 22. Сетки
Глава 23. Иерархические данные и компонент *TTreeView*
Глава 24. Панели-контейнеры
Глава 25. Инструментальные планки
Глава 26. Наборы закладок и блокноты
Глава 27. Работа с датой и временем
Глава 28. Диалоговые окна
Глава 29. Технология естественного ввода
Глава 30. Управление графическим выводом
Глава 31. Холст *TCanvas*
Глава 32. Растровая и векторная графика
Глава 33. Сложные графические задачи
Глава 34. Управление печатью

ГЛАВА 9

Опорные классы VCL

У истоков успеха Delphi в конкурентной борьбе с другими средами разработки программного обеспечения лежит хорошо продуманная *библиотека визуальных компонентов* (Visual Components Library, VCL). Библиотека VCL представляет собой набор программных модулей, содержащих описание огромного количества разноплановых классов. В свою очередь, классы VCL позволяют создавать программный продукт любой степени сложности практически для любой сферы применения. В перечне: обычные приложения для Windows; приложения для баз данных; проекты DataSnap; Web-приложения; службы; серверы и клиенты COM и многое, многое другое.

У VCL много достоинств, среди них: полная поддержка концепции ООП; многообразие сфер применения; простота использования; высокая надежность кода; быстрота разработки приложений; легкость изучения; открытость (исходные коды библиотеки доступны программисту). Важно то, что библиотека VCL постоянно совершенствуется высокопрофессиональными программистами Embarcadero, и при этом поддерживается обратная совместимость проектов. Другими словами, все разработанные на базе более ранних версиях Delphi проекты легко переносятся на новые платформы.

Задача этой главы — дать читателю общее представление о концепции построения библиотеки и о ключевых классах VCL. В результате вы станете владеть методологией, которая упростит изучение любого класса VCL и, что очень важно, позволит на профессиональном уровне применять компоненты Delphi в своих проектах.

> ***Замечание***
>
> В Delphi XE2 появилась еще одна платформа, позволяющая разрабатывать программное обеспечение как для операционной системы Windows, так и для Mac OS X и iOS. Новая платформа называется FireMonkey (FMX).

На рис. 9.1 представлен фрагмент иерархии наследования классов Delphi. Здесь вы обнаружите наиболее важные классы VCL. Почему поставлен акцент на слово "важные"? Потому, что именно эти классы определяют основные родовые черты своих многочисленных потомков. Изучив опорные классы VCL, мы получим представление о ключевых особенностях всех классов библиотеки.

Класс *TObject*

На вершине древа наследования расположен класс `TObject`. Это в своем роде небожитель, выше него нет никого. Класс `TObject` не без гордости взирает с Олимпа на свое многочисленное потомство, ведь в каждом из потомков класса есть его родовые черты.

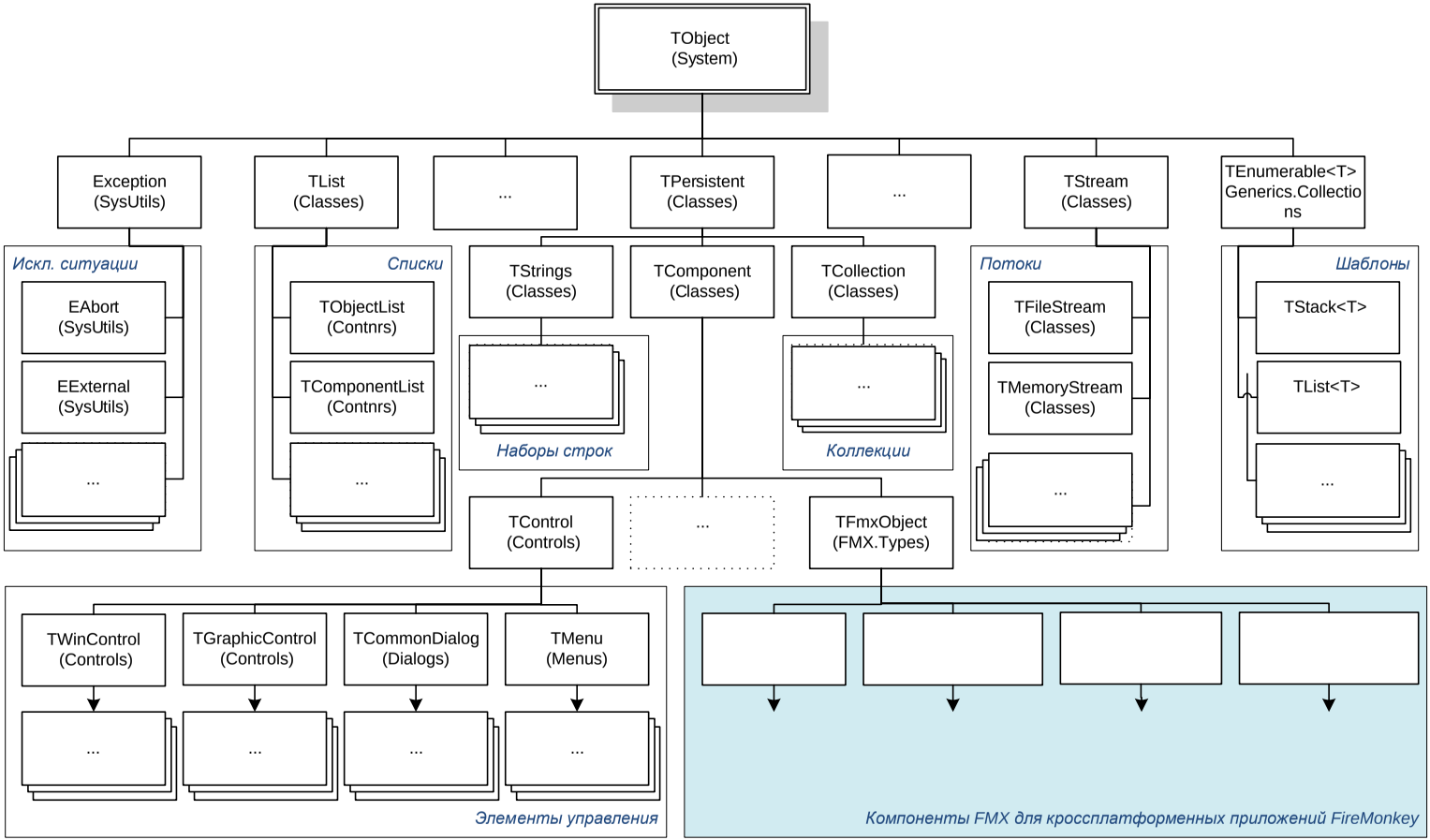


Рис. 9.1. Фрагмент иерархии VCL

В ранних версиях Delphi класс `TObject` (как, впрочем, и подавляющее большинство классов с вершины пирамиды VCL) был сделан абстрактным. Это означало, что на основе `TObject` невозможно было создать самостоятельный объект (экземпляр класса). В современных версиях Delphi основополагающий класс VCL больше не абстрактен и обладает правом превратиться в объект.

> ***Замечание***
>
> *Абстрактный класс* никогда не сможет стать физическим объектом (как кнопка, строка ввода или панель). В абстрактном классе, как правило, только определяются заголовки необходимых методов, а программный код их реализации отсутствует. На первый взгляд такой подход может показаться несколько странным. Казалось бы, какая сложность описать в классе реализацию метода и тем самым избавить от этой необходимости потомков? Но при более глубоком рассмотрении оказывается, что лобовое решение — не лучшее. Абстрактный класс — это в своем роде договор о намерениях. Таким образом, с одной стороны, он принуждает своих потомков описывать объявленные в нем методы. Так обеспечивается единая концепция в построении всех дочерних классов. С другой стороны, абстрактный класс ни в коем случае не ограничивает своих наследников в поиске путей реализации. Благодаря этому дочерние классы приобретают физическую независимость от предка.

К задачам, решаемым классом `TObject`, в первую очередь стоит отнести:

- создание, поддержку и уничтожение объекта; распределение, инициализацию и освобождение памяти, необходимой для этого объекта;
- возврат всей информации об экземпляре класса, в том числе данных RTTI об опубликованных свойствах и методах объекта;
- поддержку взаимодействия объекта с внешней средой с помощью сообщений и интерфейсов.

> ***Внимание!***
>
> В заключительной части книги мы познакомимся с новейшим "оружием" Delphi XE2 — технологией разработки кроссплатформенных приложений FireMonkey (FMX). Основу библиотеки классов FMX составляет класс `TFmxObject`.

Управление жизненным циклом объекта

Исследование вклада класса `TObject` в классы VCL начнем с конструктора и деструктора. Это наиболее важные методы для любого из класса VCL — они управляют существованием объекта.

```
constructor Create; //создание объекта
destructor Destroy; virtual; //разрушение объекта
```

Пока мы видим наиболее простое объявление конструктора и деструктора, в ряде более поздних классов методы усложнятся и дополнятся параметрами. Совсем недавно *(см. главу 7)* мы уже пользовались услугами конструктора и деструктора, один из простейших примеров создания и уничтожения объекта вы обнаружите, возвратившись к листингам 7.6 и 7.7. А сейчас мы рассмотрим более сложный способ работы с конструктором.

Если вы имеете хотя бы небольшой опыт работы с Delphi, то вам наверняка приходилось размещать на форме проекта различные компоненты (среди них кнопки, строки ввода, метки). Это очень удобно, для создания элемента управления достаточно взять компонент с палитры и положить его на форму. А вы пробовали создавать элементы управления и размещать их на форме, не обращаясь к палитре компонентов? Если нет, то следующий пример окажется для вас интересным.

Выбрав пункт меню **File | New | VCL Forms Application**, создайте новое приложение на основе формы. Подключите к строке `uses` проекта модуль `StdCtrls` (в нем расположен код стандартных элементов управления). В перечне событий формы `Form1` (на вкладке **Events** Инспектора объектов) находим событие `OnShow`. Двойной щелчок левой кнопкой мыши в строке напротив названия события заставит Delphi создать код обработчика события, вызываемого в момент показа формы на экране. Вам осталось лишь повторить несколько строк кода, приведенных в листинге 9.1.

Листинг 9.1. Демонстрация возможностей конструктора

```
uses ..., StdCtrls;
...
procedure TForm1.FormShow(Sender: TObject);
var Btn:TButton; //переменная класса кнопки TButton
begin
  Btn:=TButton.Create(Form1); //вызов конструктора класса TButton
  Btn.Parent:=Form1;          //размещаем кнопку на форме
end;
```

После запуска приложения в левом верхнем углу формы появится кнопка. Она возникла не из воздуха, — это следствие вызова конструктора `Create()` для экземпляра класса `TButton`.

Обязательно проведите небольшой эксперимент — замените в коде листинга 9.1 класс `TButton` на `TEdit` (или `TMemo`, `TLabel`, `TComboBox`). Это придется сделать в двух местах кода события `OnShow` формы: в секции локальных переменных и в строке вызова конструктора. Вновь запустите проект на выполнение. В результате на свет появится элемент управления именно того класса, который вы указали. Как видите, унаследованный от `TObject` конструктор выполняет свою задачу во всей иерархии классов.

Проведем еще один опыт. На этот раз он посвящен исследованию деструктора класса — методу `Destroy()`. Для опыта нам понадобится помощь нового проекта Delphi. Разместим две кнопки `TButton` (вы их найдете на странице **Standard** палитры компонентов) на форме. Научим одну из кнопок уничтожать своего "коллегу". Для этого дважды щелкнем левой кнопкой мыши по кнопке `Button1` и в появившемся в окне редактора кода обработчике события `OnClick` напишем всего одну строку кода (листинг 9.2).

Листинг 9.2. Вызов деструктора объекта

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button2.Destroy; //уничтожение экземпляра кнопки Button2
end;
```

Запустив проект, щелкните по кнопке `Button1`. Это действие вызовет деструктор объекта `Button2`, и элемент управления исчезнет с поверхности формы. Но на этом эксперимент не завершен. Вновь щелкнем по кнопке `Button1`. В ответ разгневанный отладчик Delphi выведет на экран примерно такое сообщение, как на рис. 9.2.

Причина возмущения отладчика в том, что мы потребовали от Delphi невозможного — повторно уничтожить уже несуществующий объект. Для того чтобы впредь избежать подоб-

ных исключительных ситуаций, в наших программах вместо `Destroy()` для уничтожения экземпляра объекта следует использовать метод

```
procedure Free;
```

Это более осторожный метод. Перед тем как вызвать деструктор объекта, он убедится в факте существования этого объекта. Для проверки возможностей метода замените в листинге 9.2 обращение к деструктору `Destroy` на `Free` и вновь повторите опыт. На этот раз повторный щелчок по кнопке не приведет к исключительной ситуации.

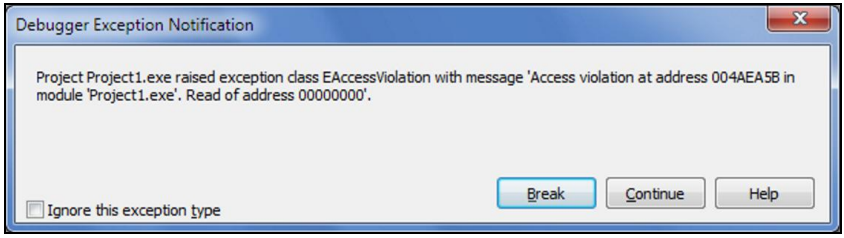


Рис. 9.2. Сообщение об ошибке доступа `EAccessViolation` к несуществующему объекту

Информирование о классе

В составе класса `TObject` имеется ряд полнофункциональных обычных методов и *методов класса* (class function), позволяющих получить подробную метаинформацию о классе. Ключевые методы `TObject` представлены в табл. 9.1.

ЗАМЕЧАНИЕ

Методы класса разрешено вызывать без создания экземпляра самого класса.

***Таблица 9.1.** Получение сведений о классе*

Метод	Описание
`class function ClassName: string;`	Функции возвратят текстовую строку с названием класса
`function ToString: string; virtual;`	
`class function UnitName: string;`	Имя модуля, в котором описан класс
`function ClassType: TClass; inline;`	Возвратит описание класса
`class function InstanceSize: Longint; inline;`	Размер в байтах, необходимый для создания экземпляра класса
`class function ClassNameIs(const Name: string): Boolean;`	Проверка, является ли объект экземпляром класса `Name`
`class function InheritsFrom(AClass: TClass): Boolean;`	Проверка, является ли объект потомком класса `AClass`
`class function ClassParent: TClass;`	Возвратит непосредственного предка объекта
`class function MethodName(Address: Pointer): string;`	Возвращает имя метода по указателю. Метод должен быть опубликован в секции `published`
`class function MethodAddress(const Name: string): Pointer;`	Возвращает указатель на метод по его имени. Метод должен быть опубликован в секции `published`

***Таблица 9.1** (окончание)*

Метод	Описание
`function FieldAddress(const Name: string): Pointer; overload;`	Возвращает указатель на поле с именем `Name`. Поле должно быть опубликовано в секции `published`

Все перечисленные в табл. 9.1 методы являются мощным оружием, позволяющим подготовленному программисту создавать очень эффективный код. Вместе с тем в неумелых руках они могут представлять и некоторую опасность, служа источником ошибок.

Создайте новый проект и разместите на его форме компонент-список (класс `TListBox`) и три кнопки (`TButton`). Дважды щелкнув левой кнопкой мыши по кнопке `Button1`, вызовите редактор кода с обработчиком события щелчка и напишите в нем всего одну строку кода (листинг 9.3).

Листинг 9.3. Вывод сообщения в момент возникновения события `Button1Click`

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Щелчок по кнопке'); //команда на вывод сообщения
end;
```

Щелчок по кнопке приведет к показу на экране компьютера сообщения.

На кнопку `Button2` возложим более существенную задачу. Щелчок по кнопке должен собрать сведения обо всех опубликованных у класса `TForm1` методах. Для этого я перебираю диапазон предполагаемых адресов методов (листинг 9.4).

Листинг 9.4. Сбор сведений об опубликованных методах класса `TForm1`

```
procedure TForm1.Button2Click(Sender: TObject);
var i : Integer;
    s : String;
    p : Pointer;
begin
  ListBox1.Items.Clear; //очищаем список
  i:=4000000;           //адрес начала диапазона поиска
  while i<9000000 do    //предположительное окончание диапазона поиска
  begin
    p:=Ptr(i);
    s:=TForm1.MethodName(p);
    if s<>'' then ListBox1.Items.Add(s); //заносим имя метода в список
    INC(i,4);            //в 32-разрядной ОС шаг поиска равен 4 байтам
  end;
end;
```

Во время перебора адресов с помощью метода `MethodName()` мы пытаемся получить имя метода, если попытка заканчивается успехом, то имя метода заносится в список `ListBox1`.

Нам осталось описать реакцию приложения на щелчок по кнопке `Button3`. Наша цель — научить программу вызывать метод формы, выбирая его имя из списка `ListBox1` (листинг 9.5).

Листинг 9.5. Вызов метода через указатель

```
procedure TForm1.Button3Click(Sender: TObject);
var s    : string;
    Proc : procedure(Sender: TObject);//прототип метода
begin
  if ListBox1.ItemIndex<>-1 then
  begin
    s:=ListBox1.Items[ListBox1.ItemIndex];
    @Proc:=TForm1.MethodAddress(s);
    Proc(nil);
  end;
end;
```

Имя метода передается в строковую переменную `s`. С помощью функции `MethodAddress()` получаем указатель на метод и вызываем его. Однако здесь есть подводные камни. В нашем примере мы безболезненно сможем обратиться только к методу `Button1Click`, благодаря которому на экран выводится текстовое сообщение. Попытка вызвать другие, объявленные в `TForm1`, методы обречена на неудачу. Причина этого в том, что полученные указатели предназначены только для работы с методами классов, т. е. с методами, для выполнения которых нет необходимости обращаться к полям, свойствам, методам конкретных объектов. По щелчку по кнопке `Button1` мы обращаемся к самодостаточной процедуре `ShowMessage()`, которая никак не зависит от формы `Form1` и расположенных на ней компонентов. Поэтому вызов проходит успешно. В коде методов `Button2Click` и `Button3Click` мы пользуемся услугами объекта `ListBox1`, поэтому косвенный вызов этих методов приводит к ошибке.

Класс *TPersistent*

Одну из наиболее значимых ветвей классов в древе VCL начинает ближайший сподвижник `TObject` — класс `TPersistent`. Основное предназначение класса — научить своих потомков работать с памятью, что позволит им загружать данные из области в памяти и выгружать данные в область. Именно благодаря `TPersistent` компоненты и форма проекта приобрели возможность сохранять свое описание в специальный файл с расширением dfm или xfm.

Как и в случае с `TObject`, основу рассматриваемого класса составляют только методы. Среди них наибольшего внимания заслуживают виртуальные методы присвоения данных:

```
procedure Assign(Source: TPersistent); virtual;
procedure AssignTo(Dest: TPersistent); virtual;
```

Перечисленные процедуры позволяют разделить одну и ту же область памяти между несколькими объектами — потомками `TPersistent`. Метод `Assign()` позволяет одному объекту подключиться к ресурсу другого, метод `AssignTo()` — передать ресурс от одного компонента к другому.

Для демонстрации этой полезной возможности создайте новый проект и разместите на его главной форме два компонента `TImage` (они предназначены для хранения и отображения векторной и растровой графики) и кнопку `TButton`. Воспользовавшись свойством `Picture` компонента `Image1`, загрузите в контейнер любой файл с изображением.

Замечание

После установки Embarcadero RAD Studio на компьютер файлы с рисунками вы обнаружите в каталоге C:\Program Files\Embarcadero\RAD Studio\9.0\Images\, если вы работаете в 32-разрядной Windows, или в каталоге c:\Program Files (x86)\Embarcadero\RAD Studio\9.0\Images\ при работе в 64-разрядной ОС.

Осталось написать одну строку кода в обработчике события `OnClick` кнопки `Button1`.

Листинг 9.6. Разделение ресурса между двумя компонентами

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image2.Picture.Assign(Image1.Picture);
end;
```

Щелчок по кнопке приведет к тому, что во втором компоненте `Image2` отобразится та же самая картинка, что и в `Image1`. Еще раз подчеркну, что метод `Assign()` не создал еще одну копию картинки, а разделил одну и ту же область памяти между двумя потомками класса `TPersistent`.

Надо понимать, что методы `Assign()` и `AssignTo()` позволяют разделять ресурс только между совместимыми компонентами, способными прочитать этот ресурс. Встретив даже такую, абсолютно некорректную строку кода

```
Button1.Assign(Image1.Picture);
```

среда проектирования не станет сопротивляться. Но полученный исполняемый код окажется неработоспособным, т. к. мы попытались передать кнопке неприемлемую область памяти.

Научив несколько объектов работать с общей областью памяти, класс `TPersistent` позаботился еще об одной важной детали — при необходимости любой из совместно работающих объектов имеет право уйти в автономный режим работы, при этом не нарушая состояния своих коллег. Допустим, что у нас имеются компонент-список `ListBox1`, хранящий какой-то перечень текстовых строк, и многострочный текстовый редактор `Memo1:TMemo`. Это совместимые компоненты (они умеют работать с набором строк `TStrings`). Поэтому строка кода

```
Memo1.Lines.Assign(ListBox1.Items);
```

окажется вполне корректной. После ее вызова оба компонента разделят область памяти и станут отображать одинаковые текстовые строки. Но как только какой-то из компонентов захочет изменить свой текст (например, будет вызвана команда, добавляющая еще одну строку в редактор `Memo1`), для него будет моментально выделена отдельная область памяти, в которую будут перенесены все изначальные строки и новый текст.

Основа компонента, класс *TComponent*

Проникнем еще глубже в иерархию классов VCL и рассмотрим третий по счету в цепочке наследования (см. рис. 9.1) класс `TComponent`. Наш очередной знакомый построен на основе только что изученного класса `TPersistent`. Кроме того, с целью поддержки технологии COM *(см. главу 42)* в объявлении класса нашлось место для двух интерфейсов.

```
TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
```

Обсудим вклад `TComponent` в библиотеку VCL. Во-первых, от него берут начало все компоненты VCL, которые вы найдете на палитре компонентов Delphi. Во-вторых, в логику клас-

са заложено понятие "частной собственности". Другими словами, в `TComponent` предусмотрено право владения, когда один объект выступает владельцем другого объекта. В-третьих, класс расширяет возможности по хранению и обслуживанию данных. В-четвертых, в отличие от своих предков (оперирующих только методами) класс `TComponent` обладает свойствами.

Самое главное свойство, подаренное `TComponent` своим наследникам, — имя.

```
property Name: TComponentName; // type TComponentName: string;
```

Каждый компонент, задействованный в проекте Delphi, обязан иметь уникальное имя. Имя присваивается автоматически в момент переноса элемента управления с палитры компонентов на поверхность рабочей формы проекта. Механизм выбора имени достаточно прост: берется имя класса (допустим, `TButton`), убирается первый символ "T" и к окончанию имени добавляется порядковый номер элемента управления этого типа на форме. Так первая кнопка, размещенная на форме, получит название `Button1`, вторая — `Button2` и т. д.

У каждого компонента опубликовано свойство, никогда не используемое системой:

```
property Tag: Longint;
```

Это свойство часто применяется программистами для дополнительной идентификации объекта или хранения ассоциированных с объектом целочисленных значений.

Владение компонентом

При разработке проекта на основе формы (класс `TForm`) мы располагаем на поверхности формы кнопки, строки ввода, метки и другие компоненты, необходимые для работы приложения. Форма (являющаяся дальним потомком класса `TControl`) становится *владельцем* (owner) этих компонентов. В свою очередь, каждый из компонентов знает, кому он принадлежит, для этого следует опросить свойство

```
property Owner: TComponent;
```

Для того чтобы любой компонент прямо в момент своего рождения приобретал владельца на уровне класса `TComponent`, несколько усовершенствован конструктор

```
constructor Create(AOwner: TComponent);
```

В объявлении появился параметр `AOwner`, благодаря ему устанавливается связь компонента с его владельцем в момент вызова конструктора. Если вы вернетесь к первому примеру из этой главы (см. листинг 9.1), то увидите, что при создании кнопки мы указали, что ее владельцем становится форма `Form1`.

> ***Внимание!***
>
> Уничтожение объекта-владельца влечет за собой уничтожение всех принадлежащих ему объектов.

Владелец компонентов имеет представление о числе принадлежащих ему компонентов и в состоянии обратиться к каждому из них. Для этого предназначены соответствующие свойства:

```
property ComponentCount: Integer; //число компонентов
property Components[Index: Integer]: TComponent; //список компонентов
```

Умение работать со списком подчиненных компонентов не раз пригодится нам при проектировании своих приложений. Создайте новый проект и разместите на его форме несколько

любых элементов управления, которые вы найдете на палитре компонентов Delphi, в их числе редактор значений — компонент `ValueListEditor1: TValueListEditor` и обычная кнопка `Button1:TButton`. Листинг 9.7 содержит код обработчика события `OnClick()` по кнопке `Button1`.

Листинг 9.7. Получение списка подчиненных компонентов

```
procedure TForm1.Button1Click(Sender: TObject);
var i:integer;
begin
  ValueListEditor1.Strings.Clear;
  for i:=0 to form1.ComponentCount-1 do
    ValueListEditor1.Strings.Add(
            form1.Components[i].ClassName+'='+form1.Components[i].Name);
end;
```

В рамках цикла осуществляется перебор всех принадлежащих форме `Form1` компонентов с выводом их названия класса и имени в редактор значений `ValueListEditor1`.

Каждый компонент знает свой индекс в списке компонентов своего владельца:

```
property ComponentIndex: Integer;
```

Ряд методов позволяет манипулировать списком компонентов, для которых `TControl` выступает владельцем. Удаление компонента из списка обеспечивается процедурой

```
procedure RemoveComponent(AComponent: TComponent);
```

Полная очистка списка с одновременным уничтожением компонентов:

```
procedure DestroyComponents;
```

Вставка в конец списка нового компонента:

```
procedure InsertComponent(AComponent: TComponent);
```

Поиск компонента-потомка в списке:

```
function FindComponent(const AName: string): TComponent;
```

ГЛАВА 10

Массивы указателей, наборы строк и коллекции

Очень часто перед разработчиком программного обеспечения возникает необходимость обеспечить хранение и обработку некоторого перечня разнотипных данных или даже объектов, с возможностью быстрого доступа к любому из обслуживаемых элементов. Хороший тому пример был представлен в *главе 9*, в ней мы обсуждали возможность владения одним компонентом (допустим, панелью) некоторого перечня других компонентов. В таком случае головной компонент должен иметь возможность обратиться к любому из своих подчиненных.

Как решить подобную задачу? Нам известно, что наиболее быстрый доступ к своим элементам предоставляет массив. Однако обычный массив специализируется на обслуживании одинаковых по размеру элементов, поэтому сохранить перечень разномастных компонентов в ячейках классического массива невозможно. Для того чтобы выйти из сложившегося положения, разработчики VCL пошли на хитрость и создали массив, каждый элемент которого предназначен не для объекта, а для хранения указателя на объект или какие-то иные данные (рис. 10.1). Благодаря массиву была обеспечена высокая скорость доступа к ячейкам, а благодаря всеядности указателя приобретена возможность ссылаться на любые данные и объекты.

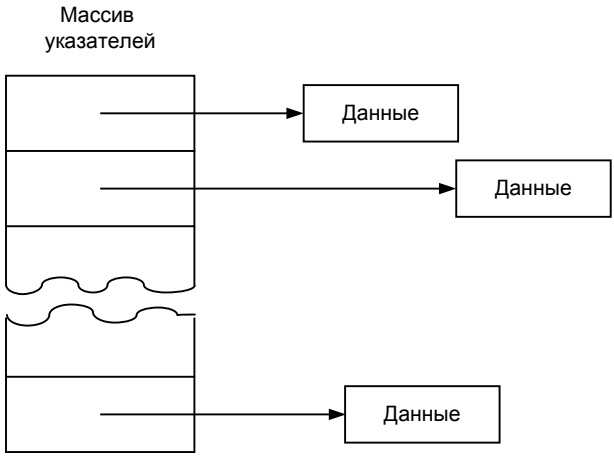


Рис. 10.1. Представление массива указателей в памяти

Массив указателей, класс *TList*

В VCL имеется множество разноплановых классов, способных работать с массивом указателей. Мы с ними еще не раз встретимся на страницах этой книги. А сейчас познакомимся с ключевым специалистом по хранению перечня разнотипных данных, описанным в модуле `Classes` классом `TList`.

Именно `TList` выступает в роли точки опоры для широкого перечня классов. Надо подчеркнуть, что речь не обязательно ведется о наследовании, экземпляр `TList` очень часто просто инкапсулируется в другой класс в качестве внутреннего объекта-поля. Именно так поступили уже знакомый нам родоначальник компонентов `TComponent` и ряд других представителей VCL (рис. 10.2).

Замечание

В модуле `Generics.Collections` объявлен одноименный класс `TList`, решающий схожие со своим коллегой из модуля `Classes` задачи. Самое существенное различие между "однофамильцами" состоит в том, что класс из модуля `Generics.Collections` является универсальным классом-шаблоном, ориентированным на работу с обобщенным типом данных, а герой этой главы — классический пример массива указателей.

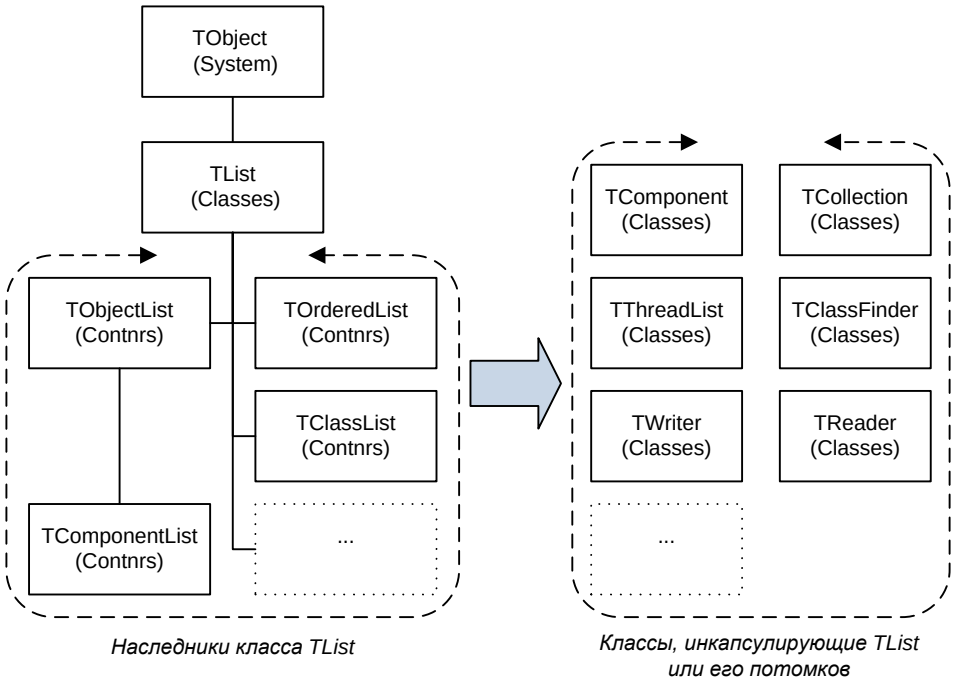


Рис. 10.2. Место класса `TList` в библиотеке VCL

Класс `TList` аскетичен, в нем сконцентрирован минимально необходимый набор методов, позволяющих:

- добавлять в массив и удалять из него ссылки на данные и объекты;
- осуществлять реорганизацию объектов внутри массива;

- осуществлять поиск ассоциированных с массивом объектов;
- упорядочивать объекты в массиве.

Размер массива указателей должен быть установлен программистом сразу после создания экземпляра класса `TList`. Для этой цели предназначено свойство

```
property Capacity: Integer;
```

О числе элементов массива проинформирует свойство

```
property Count: Integer;
```

Число элементов `Count` не должно превышать размера массива `Capacity`.

Доступ к массиву ссылок на объекты или данные предоставляет свойство

```
property Items[Index: Integer]: Pointer;
```

Вы можете получить указатель на объект, включенный в список по его порядковому номеру. Для доступа к первому объекту переменной `Index` присвойте значение `0`, ко второму — `1`, индекс последнего — `Count-1`. В массиве ссылок не исключено нахождение пустых элементов. В этом случае указатель вернет неопределенный указатель `nil`.

Свойство `List` предназначено для организации непосредственного доступа к массиву указателей.

```
property List: PPointerList; //только для чтения
```

Основные методы класса `TList` представлены в табл. 10.1.

Таблица 10.1. *Методы класса* `TList`

Метод	Описание
`function Add(Item: Pointer): Integer;`	Добавляет в конец списка указатель на новый элемент `Item`
`procedure Clear; dynamic;`	Очищает массив, удаляя из него указатели `Count=Capacity=0`
`procedure Delete(Index: Integer);`	Удаляет указатель из ячейки с номером `Index`. Оставшиеся элементы массива смещаются на свободное место. Свойство `Count` уменьшается на 1
`procedure Exchange(Index1, Index2: Integer);`	Меняет местами элементы массива с порядковыми номерами `Index1` и `Index2`
`function Expand: TList;`	Если `Count=Capacity`, увеличивает размер массива. Если размер списка более 8 элементов, метод расширит список на 16 элементов, от 5 до 8 — на 8, менее 5 — на 4 элемента
`function ExtractItem(Item: Pointer; Direction: TDirection): Pointer;`	Изымает из массива указатель `Item`. Ячейка массива освобождается и остальные элементы сдвигаются на 1 позицию вверх. Параметр `Direction` уточняет направление поиска указателя: с начала массива (`FromBeginning`) или с конца (`FromEnd`)
`function Extract(Item: Pointer): Pointer; inline;`	
`function First: Pointer;`	Вернет указатель первого элемента в массиве
`function IndexOfItem(Item: Pointer; Direction: TDirection): Integer;`	Находит в массиве индекс ячейки, в которой хранится указатель `Item`. Поиск осуществляется линейно, направление уточняет параметр `Direction`
`function IndexOf(Item: Pointer): Integer;`	

Таблица 10.1 *(окончание)*

Метод	Описание
`procedure Insert(Index: Integer; Item: Pointer);`	Вставит в массив элемент `Item`. Место определяется параметром `Index`
`function Last: Pointer;`	Вернет указатель последнего элемента в списке `Items[Count-1]`
`procedure Move(CurIndex, NewIndex: Integer);`	Переместит элемент массива из позиции `CurIndex` в позицию `NewIndex`
`procedure Pack;`	Упаковка массива элементов. Из массива удаляются все указатели `nil`, остальные элементы смещаются к началу списка
`function RemoveItem(Item: Pointer; Direction: TDirection): Integer;`	Находит и удаляет из массива указатель, равный `Item`, и возвращает индекс удаляемого элемента. Направление поиска уточняет параметр `Direction`
`function Remove(Item: Pointer): Integer;`	
`procedure Sort(Compare: TListSortCompare);` `type TListSortCompare = function (Item1, Item2: Pointer): Integer;`	Сортирует элементы массива

ЗАМЕЧАНИЕ

Все отвечающие за удаление ячеек массива методы класса `TList` не разрушают данные в памяти компьютера, а просто удаляют указатели на них. Это замечание справедливо и для деструктора `TList`: деструктор просто уничтожает массив с указателями, не затрагивая данные и объекты.

В листинге 10.1 предложен простейший пример работы с массивом указателей.

Листинг 10.1. Пример работы с массивом указателей `TList`

```
var List:TList;
    i:integer;
    x:^integer;
begin
  List:=TList.Create;
  List.Capacity:=10;
  {заполним массив}
  for i := 0 to List.Capacity-1 do
  begin
    New(x);       //новая динамическая переменная целого типа
    x^:=i;        //присваиваем x значение счетчика
    List.Add(x); //передаем ссылку на переменную в элемент массива
  end;
  {собираем данные об указателях и их значениях}
  for i := 0 to List.Count-1 do
    Memo1.Lines.add(Format('%p->%d',
                    [List.Items[i], INTEGER(List.Items[i]^)]));
  List.Clear;
  List.Destroy;
end;
```

В нашем примере мы создаем экземпляр массива указателей и распределяем память, достаточную для хранения 10 элементов массива. Затем, вызывая процедуру `New()`, создаем 10 динамических переменных типа `integer` и передаем в них значение счетчика `i`. На следующем этапе указатель на динамическую переменную направляется в соответствующую ячейку массива. Содержимое массива указателей просматривается во втором цикле, здесь хранящиеся в массиве указателей данные выводятся в компонент `Memo1:TMemo`.

Контейнер объектов, класс *TObjectList*

Прямой потомок класса `TList` контейнер объектов `TObjectList` описан в модуле `Contnrs`. Класс обладает более узкой специализацией: он отвечает за хранение и обслуживание объектов. Главное отличие класса от `TList` в том, что он способен выступать владельцем своих объектов. Для этого достаточно установить в `true` свойство

```
property OwnsObjects: Boolean;
```

Вступив в права владения объектами, контейнер станет относиться к ним значительно строже, чем его предшественник `TList`. Теперь вызов метода `Clear()` не просто очистит список, а уничтожит находящиеся в нем объекты. То же самое произойдет и при вызове деструктора контейнера.

С правами владения связаны обе версии конструкторов класса:

```
constructor Create; overload;
constructor Create(AOwnsObjects: Boolean); overload;
```

Разница между вариациями конструкторов в уточнении прав на содержащиеся в контейнере объекты. Экземпляр списка `TObjectList`, созданный первым вариантом конструктора, автоматически становится владельцем всех находящихся на хранении объектов. Второй вариант метода `Create()` допускает, что список не станет вступать в права владения своими элементами, для этого в параметр `AOwnsObjects` передается значение `false`.

Для поиска в списке объекта класса `AClass` предназначен метод

```
function FindInstanceOf(AClass: TClass; AExact: Boolean = True;
                        AStartAt: Integer = 0): Integer;
```

Если в параметр `AExact` передано значение `false`, то в поиск будут вовлечены все потомки класса `AClass`, иначе необходимо точное совпадение класса. Метод начнет просмотр списка с позиции `AStartAt` и остановится на самом первом объекте, соответствующем критериям поиска. Его индекс и будет возвращен этой функцией.

> ***ЗАМЕЧАНИЕ***
>
> К особенностям класса следует отнести способность реагировать на операции вставки, изменения или удаления его элементов. В этот момент автоматически вызывается процедура уведомления `Notify()`.

Контейнер компонентов, класс *TComponentList*

Изучение массивов указателей мы завершим упоминанием класса `TComponentList`. Это прямой наследник только что изученного списка объектов `TObjectList` с еще более узкой специализацией — он нацелен на обслуживание исключительно потомков класса `TComponent`.

Тому подтверждение его основное свойство

```
property Items[Index: Integer]: TComponent;
```

предоставляющее доступ к хранилищу компонентов.

Все унаследованные от предков свойства и методы подкорректированы для работы с экземплярами класса `TComponent`. Во всем остальном список компонентов повторяет своих славных предков и не должен вызвать затруднений в работе.

Наборы строк, класс *TStrings*

Родоначальник наборов строк, класс `TStrings`, описан в модуле `Classes` и является наследником рассмотренного в *главе 9* класса `TPersistent`. Класс предназначен для организации работы с текстовыми строками и связанными с ними объектами. Класс играет ключевую роль в таких компонентах, как `TListBox`, `TComboBox`, `TMemo` и во многих других элементах управления, специализирующихся на обработке многострочного текста (рис. 10.3). Как правило, в этих элементах набор строк доступен при посредничестве свойств с названиями `Items` или `Lines`.

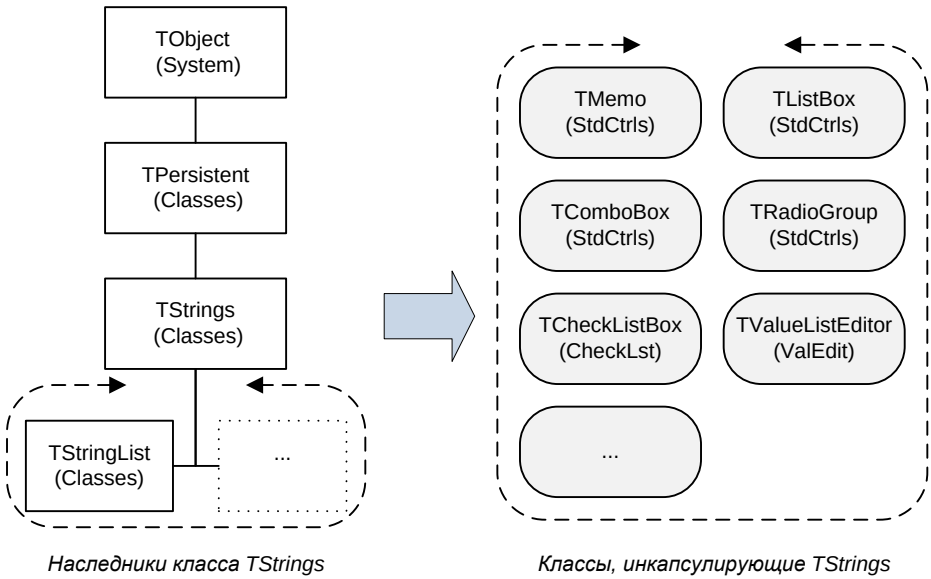


Рис. 10.3. Место класса `TStrings` в библиотеке VCL

Теперь познакомимся с перечнем основных задач, за решение которых отвечает класс `TStrings`:

- создание нового набора строк;
- загрузка и сохранение набора строк;
- манипуляции со строками в наборе;
- связывание внешнего объекта со строкой набора;
- операции со связанными объектами.

В первую очередь разберемся с кодировкой хранимых в наборе строк символов. За ее назначение несет ответственность свойство

```
property DefaultEncoding: TEncoding;
```

В момент создания экземпляра класса устанавливается кодировка по умолчанию. Порядок установки пользовательской кодировки предложен в листинге 10.2.

Листинг 10.2. Изменение кодировки списка

```
var  OL:TStrings;
begin
  OL:=TStrings.Create;
  OL.DefaultEncoding:=TEncoding.UTF8;
  //работа со списком
  OL.Free;
end;
```

Список способен одновременно хранить не только текстовые строки, но и ассоциированные с ними объекты. Для получения доступа к строкам и объектам предназначены свойства

```
property Strings[Index: Integer]: String;
property Objects[Index: Integer]: TObject;
```

Сведения о числе строк в наборе предоставляются свойством:

```
property Count : Integer; //только для чтения
```

Редактирование списка и управление данными

В классе `TStrings` объявлен богатый набор методов, предназначенных для вставки/удаления строк в набор и управления ассоциированными со строками данными (табл. 10.2).

Таблица 10.2. *Управление данными в классе* `TStrings`

Метод	Описание
`function Add(const S: string): Integer;`	Добавляет в конец набора строку `S` и возвращает ее порядковый номер
`function AddObject(const S: string; AObject: TObject): Integer;`	Добавляет в конец набора строку `S` и ассоциирует с данной строкой объект. Возвращает порядковый номер
`procedure AddStrings(Strings: TStrings); overload;`	Три перегружаемых версии метода позволяют добавить в конец набора другой набор `Strings`. Последний из представленных методов также передает в набор массив объектов
`procedure AddStrings( const Strings: TArray<string>); overload;`	
`procedure AddStrings( const Strings: TArray<string>; const Objects: TArray<TObject>); overload;`	
`Procedure Append(const S: string);`	Аналогичен методу `Add()` за исключением того, что не возвращает номер добавленной строки
`procedure Clear;`	Очищает набор от всех строк

***Таблица 10.2** (окончание)*

Метод	Описание
`procedure Delete(Index: Integer);`	Удаляет из набора строку с порядковым номером `Index`
`function Equals(Strings: TStrings): Boolean;`	Сравнивает строки текущего объекта со строками `Strings`. Возвратит `true` в случае идентичности наборов строк
`procedure Exchange(Index1, Index2: Integer);`	Меняет местами пару "строка + объект" с номерами `Index1` и `Index2`
`procedure Insert(Index: Integer; const S: string);`	Вставляет в набор строку `S` в место, определенное параметром `Index`
`procedure InsertObject(Index: Integer; const S: string; AObject: TObject);`	Вставляет в набор строку `S` и ассоциирует с ней объект `AObject`. Место вставки определяется параметром `Index`
`procedure Move(CurIndex, NewIndex: Integer);`	Перемещает пару "строка + объект" из позиции `CurIndex` в позицию `NewIndex`
`procedure Put(Index: Integer; const S: string);`	Изменяет текст в строке с номером `Index` на `S`
`procedure PutObject(Index: Integer; AObject: TObject);`	Заменяет объект в строке с номером `Index` на `AObject`

Мы уже говорили, что наборы строк широко используются во многих компонентах Delphi. Например, экземпляр класса `TStrings` интегрирован в популярный элемент управления `TListBox`, в нем доступ к набору строк предоставляется при посредничестве свойства `Items`. Листинг 10.3 демонстрирует возможности компонента при работе со строками и объектами. Для повтора листинга вам достаточно разместить на форме единственный элемент управления — `ListBox1:TListBox`.

Листинг 10.3. Связывание элементов списка с объектами

```
{событие OnCreate, возникающее при создании формы Form1}
procedure TForm1.FormCreate(Sender: TObject);
var i:integer;
    btn:TButton;
begin
  for i:=1 to 10 do
    begin {Динамически создаем 10 экземпляров кнопок TButton}
      btn:=TButton.Create(Form1);
      with btn do
      begin
        Parent:=Form1;
        Left:=200;
        top:=Height*(i-1);
        Name:='Button'+IntToStr(i);
        {заносим в список строку с именем кнопки и ссылку на нее}
        ListBox1.Items.AddObject(Name,btn);
      end;
    end;
end;
```

```
{событие OnClick(), генерируемое по щелчку на элементе списка ListBox1}
procedure TForm1.ListBox1Click(Sender: TObject);
var i:Integer;
begin
  i:=ListBox1.ItemIndex; //узнаем индекс выбранного элемента
  if i<>-1 then //передаем фокус ввода связанной с элементом кнопки
    (ListBox1.Items.Objects[i] as TButton).SetFocus;
end;
```

Листинг содержит код двух обработчиков событий. Событие `OnCreate` возникает в момент создания главной формы проекта, в этот момент мы динамически создаем 10 экземпляров кнопки `TButton` и размещаем их на форме. В этом обработчике события наиболее примечательна строка `ListBox1.Items.AddObject(Name,btn)`, благодаря ей мы заносим в список строку с названием только что созданной кнопки `Name` и ссылку на эту кнопку.

Второе событие `OnClick` генерируется по щелчку на элементе списка `ListBox1`. В нем мы получаем индекс `i` выбранного элемента, затем по этому индексу — ссылку на связанный с этим элементом компонент. Для того чтобы визуализировать это событие, мы передаем фокус ввода этому элементу управления.

Загрузка и сохранение строк

Класс `TStrings` предоставляет возможность сохранять набор строк в поток или в файл

```
procedure SaveToFile(const FileName: string); overload;
procedure SaveToFile(const FileName: string;
                     Encoding: TEncoding); overload;
procedure SaveToStream(Stream: TStream); overload;
procedure SaveToStream(Stream: TStream; Encoding: TEncoding); overload;
```

При сохранении данных допускается конкретизировать особенности кодировки символов, для этого предназначен параметр `Encoding`.

> ***ЗАМЕЧАНИЕ***
>
> На порядок сохранения данных в файл или поток влияет свойство `LineBreak`, определяющее разделительный междустрочный символ. По умолчанию это кодовая комбинация #13#10 перевода строки.

На формат сохраняемых в потоке или файле данных оказывает влияние свойство

```
property WriteBOM: Boolean;
```

В состоянии `true` свойство активирует *маркер последовательности байтов* (Byte Order Mark, BOM). Это символ, показывающий кодировку файла: UTF-16 или UTF-32.

При загрузке данных в набор `TStrings` следует применять методы

```
procedure LoadFromStream(Stream: TStream);
procedure LoadFromFile(const FileName: string);
```

Чтобы узнать кодировку загружаемых данных (методы `LoadFrom...`), следует прочитать содержимое свойства

```
property Encoding: TEncoding; //только для чтения
```

Так же предусмотрен способ выгрузки строк в размещенный в памяти массив и чтение строки из памяти:

```
function GetText: PChar;
procedure SetText(Text: PChar);
```

Используемый в описанных операциях буфер памяти идентифицируется указателем `PChar`, строки разделяются парой символов `CR/LF`. Самый последний символ в этом буфере нулевой.

С появлением в Delphi обобщенного типа данных (о котором мы говорили в *главе 8*) в арсенале `TStrings` появилась еще пара методов, позволяющих передать текст и объекты в соответствующие массивы:

```
function ToStringArray: TArray<string>;
function ToObjectArray: TArray<TObject>;
```

Объединение строк

Класс `TStrings` предполагает существование нескольких способов представления хранящихся в нем текстовых строк в виде одной объединенной строки. Простейший способ объединения предоставит свойство

```
property Text: string;
```

Разделение строк в свойстве осуществляется с помощью символа перевода каретки (код #13#10). При желании разделитель можно заменить любым другим символом или комбинацией символов. Для этого предназначено свойство

```
property LineBreak: string;//по умолчанию #13#10
```

В классе предусмотрено еще одно, дублирующее свойство, работающее совместно с `Text`, но позволяющее представлять текст в альтернативном формате:

```
property DelimitedText: string;
```

Свойство возвратит объединенную строку, в которой текстовые элементы разделены символом, указанным в свойстве

```
property Delimiter: Char;
```

Если какая-либо из строк набора `TStrings` состоит из нескольких слов, разделенных пробелами, то эта строка возвращается заключенной в двойные кавычки или в обрамлении символов из свойства

```
property QuoteChar: Char; //по умолчанию двойная кавычка
```

Еще один способ обращения к строкам набора предоставляет свойство

```
property CommaText: string;
```

Это свойство также возвратит суммарную строку, но на этот раз в *формате системных данных* (system data format, SDF), предусматривающем разделение элементов текста запятыми.

> ***Замечание***
>
> Если какая-либо из строк набора `TStrings` состоит из нескольких слов, разделенных пробелами, то в свойствах `DelimitedText` и `CommaText` эта строка возвращается заключенной в двойные кавычки.

Поиск строки и объекта

Для поиска строки в наборе и связанного со строкой объекта предназначены функции

```
function IndexOf(const S: string): Integer;
function IndexOfObject(AObject: TObject): Integer;
```

Методы осуществляют поиск первого появления строки (объекта) в наборе. При успешном выполнении возвращается порядковый номер строки, в противном случае результат функции равен –1.

Обслуживание данных "параметр — значение"

Ряд свойств и методов класса `TStrings` создан для работы со списками, каждая строка которого хранит пару "параметр — значение", например: `FontSize=14` или `Align=Left`.

> ***Замечание***
> Для работы с данными, представленными в формате "параметр — значение", предназначен элемент управления `TValueListEditor`.

Разделительный символ по умолчанию соответствует знаку равенства, для его изменения предназначено свойство

```
property NameValueSeparator: Char; //по умолчанию "="
```

Управление параметром и значением можно осуществлять раздельно, для этого предназначены свойства

```
property Names[Index: Integer]: string;
property Values[const Name: string]: string;
```

Как вы догадались, `Names` отвечает за названия параметров, `Values` — за значения.

Для поиска порядкового номера в наборе пары вида "параметр — значение" предназначен метод:

```
function IndexOfName(const Name: string): Integer;
```

При отрицательном результате поиска возвращаемое значение равно –1.

Оптимизация производительности

Класс `TStrings` инкапсулирован в ряд визуальных элементов управления Delphi. К ним относятся такие популярные компоненты, как многострочный текстовый редактор `TMemo`, список `TListBox`, комбинированный список `TComboBox`. Перечисленные компоненты обладают графическим пользовательским интерфейсом, способны реагировать на сообщения Windows и генерировать события самостоятельно. Все эти операции достаточно ресурсоемки. Например, у многострочного редактора `TMemo` есть обработчик события `OnChange`, который вызывается при малейшем изменении в тексте компонента. А теперь представим, что логика нашей программы предполагает проведение операций со значительными объемами текстовых строк одновременно. В этом случае способность многострочного редактора вызывать событие `OnChange` для каждой добавляемой (удаляемой, редактируемой) строки станет скорее бедой, чем благом.

Для того чтобы на некоторое время отключить ненужную реакцию визуальных элементов управления на все изменения в `TStrings`, разработчики класса ввели пару методов, которые должны всегда работать совместно:

```
procedure BeginUpdate;
procedure EndUpdate;
```

Метод `BeginUpdate()` указывает на то, что мы намерены осуществить группу ресурсоемких операций со строками набора и просим компонент не реагировать на наши действия. Процедура `EndUpdate()` информирует компонент, что обновление его содержимого завершено.

Совместно с представленными выше процедурами трудится свойство

```
property UpdateCount: Integer;
```

Это счетчик обновлений. Каждый вызов метода `BeginUpdate()` дает счетчику приращение на 1, метод `EndUpdate()`, наоборот, уменьшает счетчик на 1. Как вы уже догадались, активизация элемента управления происходит только при нулевом значении свойства `UpdateCount`.

Особенности класса *TStringList*

Все идеи, заложенные в классе `TStrings`, нашли свое развитие в его прямом потомке `TStringList`. В классе `TStringList` переопределены многие методы `TStrings` и добавлены некоторые новые. Это хорошо продуманный и простой в обращении класс, с помощью которого очень легко создавать самостоятельные наборы строк. Отдельный элемент класса представляет собой структуру из двух полей (листинг 10.4).

Листинг 10.4. Структура элемента класса `TStringList`

```
type TStringItem = record
    fString: string;  //текстовая информация
    fObject: TObject; //связанный объект
end;
```

Указатели на такие записи хранятся в поле, представляющем собой массив вида

```
type TStringItemList = array[0..MaxListSize] of TStringItem;
```

Класс `TStringList` воплощает в жизнь все, что ему завещано `TStrings`, и в дополнение ко всему способен:

- искать строку в списке;
- упорядочивать строки в списке;
- запретить повтор строк;
- реагировать на изменения в содержании списка.

Для поиска строки `S` в списке строк применяйте метод

```
function Find(const S: string; var Index: Integer): Boolean;
```

Если строка обнаружена, то функция возвращает `true` и порядковый номер найденной строки в параметре-переменной `Index`. Особо отметим, что функция `Find()` работает только с отсортированными списками.

Признаком того, что строки отсортированы, служит свойство

```
property Sorted: Boolean;
```

При установке свойства в `true` автоматически выполняется процедура сортировки

```
procedure Sort;
```

Если вы полагаете, что при сортировке и поиске следует учитывать регистр символов, то установите в состояние `true` свойство

```
property CaseSensitive: Boolean;
```

Если строки упорядочены, то не стоит обращаться к методу вставки строк `Insert()`. Вместо этого воспользуйтесь методом добавления строк `Add()` и вновь отсортируйте строки (листинг 10.5).

Листинг 10.5. Загрузка, сортировка и сохранение строк в `TStringList`

```
const fname='c:\towns.txt';
...
  with TStringList.Create do //создание экземпляра класса
  begin
    If FileExists(fname) then LoadFromFile(fname);//загрузка файла
    Add('Москва');      //добавляем строку
    Add('Нижний Новгород');
    Add('Ставрополь');
    Sorted := True;    //сортировка строк
    SaveToFile(fname); //экспорт списка в файл
    Free;              //уничтожение экземпляра класса
  end;
```

Класс `TStringList` способен контролировать попытки пополнения списка дубликатами строк. Вид реакции определяется свойством

```
property Duplicates: TDuplicates;
type TDuplicates = (dupIgnore,//при совпадении добавление отклоняется
                    dupAccept,//добавление принимается
                    dupError);//генерируется искл. ситуация EListError
```

Экземпляры класса `TStringList` наделены нетривиальной способностью: они реагируют на изменение своих данных и генерируют пару событий

```
property OnChanging: TNotifyEvent;
property OnChange: TNotifyEvent;
```

Данные события возникают всякий раз при добавлении, удалении, перемещении или изменении строк. Событие `OnChanging` происходит до вышеперечисленных действий, а `OnChange` — по их окончанию.

Коллекции

Родоначальником всех коллекций в Delphi выступает описанный в модуле `Classes` класс `TCollection` (см. рис. 9.1). Коллекция `TCollection` и все ее многочисленные потомки (с которыми мы столкнемся в ряде компонентов VCL) специализируются на хранении потомков экземпляров класса `TCollectionItem` — элементов коллекции. Например, элемент управления графический список `TListView` содержит коллекцию колонок `TListColumns` с элементами `TListColumn`, строка состояния `TStatusBar` свои панельки (`TStatusPanel`) прячет в коллекции `TStatusPanels`.

Замечание

Обратите внимание на особенность присвоения имен коллекциям и их элементам. По принятой договоренности имя коллекции всегда повторяет имя класса элемента и завершается символом "s" (признак множественного числа в английском языке). Таким образом, если класс элемента называется `TGrpButtonItem`, то класс соответствующей коллекции получит имя `TGrpButtonItems`.

Элемент коллекции, класс *TCollectionItem*

Коллекции специализируются на хранении элементов определенного класса. Основу элемента коллекции составляет потомок `TPersistent` — класс `TCollectionItem`.

Для создания и инициализации экземпляра `TCollectionItem` предназначен виртуальный (доступный для модификации у классов-потомков) конструктор

```
constructor Create(Collection: TCollection); virtual;
```

В единственном параметре метода программисту требуется указать коллекцию, в подчинение которой сразу после своего рождения поступит элемент. Как правило, для добавления в коллекцию нового элемента вместо явного обращения к конструктору элемента используют методы самой коллекции, в частности метод `Add()`.

Как и у любого другого объекта, у элемента коллекции имеется деструктор

```
destructor Destroy; override;
```

предназначенный для уничтожения экземпляра класса. Коллекция-владелец, кроме прямого обращения к деструктору элемента, может воспользоваться собственным методом `Delete()`.

Элемент коллекции `TCollectionItem` владеет информацией, кому он принадлежит, для этого предназначено свойство

```
property Collection: TCollection;
```

При необходимости благодаря этому свойству можно переподчинить элемент коллекции другой коллекции.

Внимание!

При перемещении между коллекциями элемент-путешественник должен помнить, что радушный прием ему обеспечит только коллекция, специализирующаяся на хранении точно таких же, как и он, по типу элементов.

Каждый элемент знает свой индекс в коллекции:

```
property Index: Integer;
```

Этот индекс может изменяться при передвижении элемента внутри коллекции или при переходе из коллекции в коллекцию. Поэтому элемент снабжен дополнительным идентификатором:

```
property ID: Integer; //только для чтения
```

Еще один потенциальный идентификатор может находиться в свойстве

```
property DisplayName: string;
```

В отличие от доступного только для чтения свойства `ID`, в свойство `DisplayName` допускается осуществлять операции записи, например, назначая подпись к элементу в коде программы. Но в этом случае не может идти речи об уникальности, т. к. несколько элементов могут иметь одинаковое название.

Создание и уничтожение коллекции

Жизненный путь коллекции начинается в момент вызова ее конструктора. В момент создания в параметре `ItemClass` требуется определить тип хранящихся в ней элементов.

```
constructor Create(ItemClass: TCollectionItemClass);
```

Позднее, если вы позабудете, какие элементы находятся в коллекции, освежить память поможет свойство

```
property ItemClass: TCollectionItemClass; //только для чтения
```

Если коллекция инкапсулируется в состав другого компонента, то ссылку на компонент-владелец возвратит метод

```
function Owner: TPersistent;
```

Для разрушения коллекции предназначен метод

```
destructor Destroy; override;
```

Вместе с коллекцией уничтожаются все принадлежащие ей элементы.

Доступ к элементу коллекции

Массив имеющихся во владениях коллекции элементов доступен благодаря свойству

```
property Items[Index: Integer]: TCollectionItem;
```

Для выяснения числа элементов в коллекции следует опросить свойство

```
property Count: Integer;
```

Манипуляции элементами коллекции

Для создания нового экземпляра коллекции с добавлением его в массив воспользуйтесь методом

```
function Add: TCollectionItem;
```

Вновь созданный элемент помещается в конец массива. Другой вариант создания элемента опирается на метод

```
function Insert(Index: Integer): TCollectionItem;
```

Новый элемент будет вставлен в позицию `Index`, остальные элементы коллекции сдвигаются.

Удаление элемента с порядковым номером `Index` производится методом

```
procedure Delete(Index: Integer);
```

Полная очистка осуществляется процедурой

```
procedure Clear;
```

У визуальных элементов управления процесс вставки, удаления и сортировки элементов коллекции сопряжен с несколькими операциями перерисовки клиентской поверхности объекта. А если идет речь о десятках или сотнях элементов, то многократное обновление компонента неблаготворно сказывается на производительности приложения. В таких случаях перед началом операций с элементами коллекции элемент управления "замораживают" вызовом метода

```
procedure BeginUpdate;
```

"Замороженный" компонент перестает перерисовывать себя до тех пор, пока не получит на это явное разрешение в виде команды

```
procedure EndUpdate;
```

Листинг 10.6 демонстрирует порядок работы с коллекцией колонок, принадлежащей списку просмотра `TListView`. В этом элементе управления доступ к колонкам (коллекция `TListColumns`) обеспечивает свойство `Columns`. Отдельная колонка (экземпляр коллекции) описывается классом `TListColumn`.

Листинг 10.6. Пример работы с коллекцией колонок в `TListView`

```
var Column:TListColumn; //экземпляр коллекции
begin
  ListView1.ViewStyle:=vsReport;//многоколоночный стиль просмотра

  with ListView1.Columns do
  begin
    BeginUpdate; //начало обновления коллекции
    Clear;       //полная очистка коллекции
    Column:=Add; //добавляем в коллекцию новый элемент
    Column.Caption:='Колонка 1';

    Column:=Add; //добавляем в коллекцию новый элемент
    Column.Caption:='Колонка 2';

    EndUpdate;  //завершение обновления коллекции
  end;
end;
```

Поиск элемента коллекции

Обсуждая элемент коллекции, мы уже говорили, что каждый из экземпляров `TCollectionItem` идентифицируется уникальным значением, хранящимся в свойстве `ID` этого элемента. Идентификатор элемента используется коллекцией для организации поиска требуемого элемента в коллекции. Для этого предназначена функция:

```
function FindItemID(ID: Integer): TCollectionItem;
```

Сравнение коллекций

Для сравнения двух коллекций воспользуйтесь функцией модуля `Classes`

```
function CollectionsEqual(C1, C2: TCollection;
                          Owner1, Owner2: TComponent): Boolean;
```

Функция проверяет идентичность данных в коллекциях `C1` и `C2`. Возвращает `true` в случае полного соответствия.

ГЛАВА 11

Классы потоков данных

Еще в начале 1990-х годов в составе языка программирования Turbo Pascal 6.0 появился весьма неординарный класс `TStream`. В логику класса разработчики сумели вложить единую методологию для осуществления операций ввода/вывода в файлы, коммуникационные порты, массивы бинарных данных, ресурсы приложений. В результате на свет появилась удобная абстракция, которая оказалась способной описать процесс переноса битов данных от источника к приемнику. Эта абстракция нашла свое воплощение в многочисленных потомках класса `TStream`, которые мы станем называть *потоками данных*.

Поток данных способен оказать неоценимые услуги в приложениях, предназначенных для чтения (записи) данных из внешнего источника. Так класс `TFileStream` специализируется на файловых операциях, `TMemoryStream` незаменим в операциях с памятью, `TStringStream` предназначен для управления текстовыми строками в памяти. Представленные на рис. 11.1 дочерние классы — более утонченные специалисты: поток `TCustomMemoryStream` незаменим при работе с памятью; `TWinSocketStream` предназначен для работы в сетевых соединениях, построенных на основе сокетов; `TBlobStream` трудится в базах данных, обслуживая поля больших бинарных объектов; `TOleStream` применяется в рамках технологии многокомпонентной модели объектов COM; поток `TResourceStream` окажется полезным при организации доступа к ресурсам (значкам, курсорам, меню и т. п.) приложения; наконец, поток `TCustomZStream` специализируется на сжатии данных.

Изучая иерархию классов `TStream` (см. рис. 11.1), заметим, что в ней нашлось место для двух одноименных классов `TStringStream`. Это не ошибка. Прямой потомок `TStream`, класс `TStringStream` существовал в устаревших версиях Delphi и специализировался на работе с текстовыми строками ANSI. При переходе на кодировку Unicode специалисты Embarcadero были вынуждены создать "однофамильца" класса `TStringStream`, но на этот раз он стал более дальним предком головного класса потоков данных `TStream`. Компилятор Delphi самостоятельно разберется, какой из классов вы намерены применять в своих проектах.

Прототип потоков данных, класс *TStream*

Абстрактный класс `TStream` описан в модуле `Classes`. Он определяет концепцию построения всех своих наследников и определяет общие для всех принципы чтения и записи в поток данных.

Класс вооружен всего парой свойств:

```
property Size: : Int64;
property Position : Int64;
```

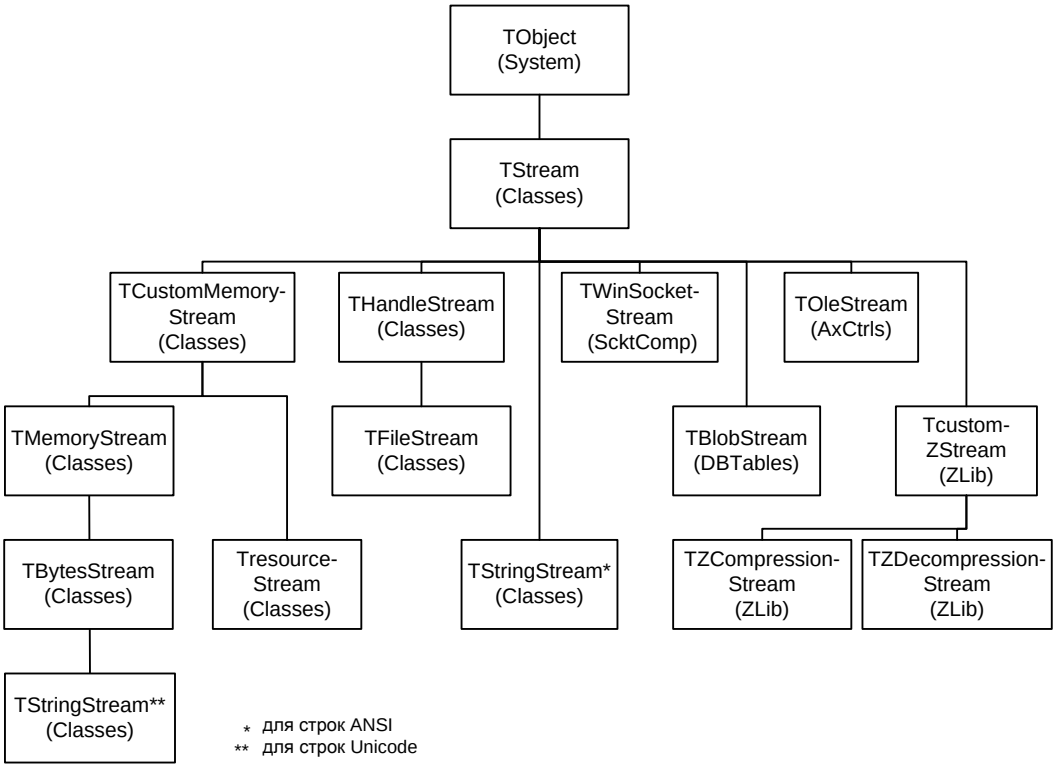


Рис. 11.1. Иерархия классов `TStream`

Размер потока данных `Size` измеряется в байтах. Попытка редактировать значение свойства в родительском классе не даст никаких результатов, но такая возможность реализована в некоторых потомках `TStream`. Свойство `Position` возвращает текущую позицию курсора внутри потока во время операций чтения или записи.

За изменение позиции курсора внутри потока отвечает метод

```
function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
```

Здесь параметр `Offset` определяет новую позицию внутри потока, а параметр `Origin` уточняет правила определения позиции.

```
type TSeekOrigin = (soBeginning, //отступ Offset байт от начала потока
                    soCurrent, //сместится на Offset от текущего положения
                    soEnd);    //отступ (Offset<=0) байт от конца потока
```

Для чтения данных из потока объявлен метод

```
function Read(var Buffer; Count: Longint): Longint;
```

Процесс записи поддерживает функция

```
function Write(const Buffer; Count: Longint): Longint;
```

Методы `Read()` и `Write()` низкоуровневые, поэтому для осуществления операций ввода/вывода я рекомендую обращаться к более совершенным процедурам:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

Данные процедуры представляют собой надстройки над методами `Read()` и `Write()`, их ключевая особенность в том, что при невозможности считать/записать указанное в аргументе `Count` число байтов немедленно генерируется исключительная ситуация: `EReadError` или `EWriteError`. Благодаря этому разработчик получает возможность контролировать корректность поведения своей программы.

Для копирования данных в поток из другого потока предназначен метод

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Здесь `Source` — поток-источник, а `Count` — число копируемых байтов. При необходимости полностью скопировать поток в параметр `Count` передается аргумент 0. Функция возвращает размер скопированного потока. Так как для выполнения операции задействуются методы `ReadBuffer()` и `WriteBuffer()`, во время копирования автоматически проверяется корректность операций чтения/записи и при возникновении ошибки вызывается исключительная ситуация.

Потоки с дескриптором, класс *THandleStream*

Класс `THandleStream` предназначен для создания потомков, способных производить чтение/запись в коммуникационный ресурс. В роли последнего могут выступать как обычный файл, так и сокет или именованный канал, главное условие в том, чтобы он обладал дескриптором коммуникационного ресурса.

```
property Handle: Integer; //только для чтения
```

Это свойство доступно только для чтения. Для того чтобы наполнить его содержанием, программист должен поработать с конструктором потока

```
constructor Create(AHandle: Integer);
```

Единственный параметр метода `Create()` как раз требует передачи дескриптора ресурса. Хороший пример (см. листинг 11.1) получения дескриптора в момент вызова конструктора демонстрирует прямой наследник `THandleStream` — класс `TFileStrem`.

Файловый поток данных, класс *TFileStream*

Файловый поток данных специализируется на обмене информацией между программой и файлом. Всеми своими умениями файловый поток обязан своим предкам, классам `TStream` и `THandleStream`, а основная особенность `TFileStream` заключена в его конструкторе

```
constructor Create(const FileName: string; Mode: Word); overload;
constructor Create(const FileName: string; Mode: Word; Rights: Cardinal); overload;
```

С вызовом конструктора создается файловый объект, который ассоциируется с файлом с именем `FileName`. Параметр `Mode` описывает режим открытия файла (табл. 11.1), параметр `Right` — права на доступ (табл. 11.2).

Таблица 11.1. *Константы режима открытия файла*

Константа	Значение	Описание
fmCreate	$0000	Создать файл с именем FileName. Если файл с таким именем уже существует, то он открывается с правами для записи

***Таблица 11.1** (окончание)*

Константа	Значение	Описание
fmOpenRead	$0001	Открыть только для чтения
fmOpenWrite	$0002	Открыть только для записи
fmOpenReadWrite	$0004	Файл одновременно доступен для чтения и записи

***Таблица 11.2.** Константы определения прав доступа*

Константа	Значение	Описание
fmShareCompat	$0000	Устаревшая константа, оставленная для обратной совместимости с проектами для DOS
fmShareExclusive	$0010	Эксклюзивный доступ к файлу, запрещает обращение из других процессов
fmShareDenyWrite	$0020	Другим процессам разрешено только чтение из файла
fmShareDenyRead	$0030	Другим процессам разрешена только запись в файл
fmShareDenyNone	$0040	Файл доступен для других процессов (приложений)

В листинге 11.1 представлен сокращенный код конструктора файлового потока.

Листинг 11.1. Конструктор класса TFileStream

```
type TFileStream = class(THandleStream)
public
...
constructor TFileStream.Create(const FileName: string;
                               Mode: Word; Rights: Cardinal);
begin
  ...
  if Mode = fmCreate then
    {режим создания файла — получим дескриптор нового файла}
    inherited Create(FileCreate(AFileName, LShareMode, Rights));
  else
    {режим открытия — получим дескриптор открытого файла}
    inherited Create(FileOpen(FileName, Mode));
end;
```

В зависимости от порядка доступа к файлу, указанного в параметре `Mode`, его дескриптор возвращает объявленные в модуле `SysUtils` функции `FileCreate()` или `FileOpen()`. Если по каким-либо причинам открыть (создать) файл не удалось, то конструктор генерирует исключительную ситуацию `EFOpenError` или `EFCreateError`.

Имя файла, ассоциированного с потоком данных, доступно в свойстве

```
property FileName: string;
```

Для организации операций чтения, записи и перемещения курсора по потоку используют унаследованные от `THandleStream` методы `Read()`, `Write()` и `Seek()`.

Завершив работу с файловым объектом, необходимо вызвать его деструктор:

```
destructor Destroy; override;
```

Листинг 11.2 демонстрирует код, создающий двоичный файл и помещающий в него 1 Кбайт данных.

Листинг 11.2. Пример записи данных в файловый поток

```
var fs:TFileStream;
    Buf : array [0..1023] of byte;
    i :integer;
begin
  fs:=TFileStream.Create('c:\demo.dat', fmCreate);{создаем файл и поток}
  for i:=0 to SizeOf(Buf)-1 do Buf[i]:=Random(255); {заполнение буфера
                    случайными числами}
  fs.Write(Buf,SizeOf(Buf)); //запись данных в файл
  fs.Destroy;  {уничтожаем файловый поток}
end;
```

Организация чтения данных из файлового потока требует учитывать текущее положение курсора чтения в потоке. Для этого нам понадобится пара свойств: `Position` и `Size`. Кроме того, размер буфера должен быть кратен размеру файла (листинг 11.3).

Листинг 11.3. Пример чтения данных из файлового потока

```
var fs : TFileStream;
    Buf : array[0..127] of byte;
begin
  fs:=TFileStream.Create('c:\demo.dat', fmOpenRead);
  while fs.Position<fs.Size do
  begin
    fs.Read(Buf, SizeOf(Buf));//считываем порцию данных в буфер
    //дальнейшая обработка данных в буфере Buf
  end;
  fs.Destroy;
end;
```

После открытия файлового потока курсор устанавливается на первом байте данных. В цикле `while..do` контролируется местоположение курсора — чтение осуществляется до тех пор, пока курсор не попытается уйти за пределы файлового потока.

> ***ЗАМЕЧАНИЕ***
>
> В составе модуля `IOUtils` объявлена интеллектуальная запись `TFile`, позволяющая создавать файл методом `Create()`, открывать `OpenRead()` и `OpenWrite()` и возвращать экземпляр потока `TfileStream`.

Пример работы с файловым потоком данных

Воспользуемся вновь полученными знаниями и разработаем небольшую утилиту, позволяющую просматривать содержимое любого файла в шестнадцатеричном формате. Для

этого нам потребуется создать новый проект и разместить на поверхности формы 3 компонента:

- главное меню (`TMainMenu`);
- диалог открытия файла (`TOpenDialog`);
- текстовый редактор (`TMemo`).

Дважды щелкнув по главному меню, вызовите редактор меню и создайте в нем единственный элемент "Открыть". Переименуйте этот элемент: `Name=miOpen`. Затем опишите обработчик события `OnClick()` по элементу меню так, как предложено в листинге 11.4.

Листинг 11.4. Обрабатываем данные с помощью `TFileStream`

```
procedure TfrmMain.miOpenClick(Sender: TObject);
var FS  :TFileStream;
    LineNum :integer;
    buf : Array[0..15] of byte;
    s   :string;
begin
  if OpenDialog1.Execute then
  begin
    Memo1.Lines.BeginUpdate;
    Memo1.Clear;
    try
      FS:=TFileStream.Create(OpenDialog1.FileName,fmOpenRead);
      LineNum:=0;
      while FS.Position<FS.Size do
      begin
        ZeroMemory(@buf,High(buf)+1); //обнуляем буфер
        FS.Read(buf,SizeOf(Buf));
        S:=Format('%.6x| %.2x %.2x %.2x %.2x %.2x %.2x %.2x %.2x
                        %.2x %.2x %.2x %.2x %.2x %.2x %.2x %.2x',
       [LineNum,buf[0],buf[1],buf[2],buf[3],buf[4],buf[5],buf[6],buf[7],
        buf[8],buf[9],buf[10],buf[11],buf[12],buf[13],buf[14],buf[15]]);
        Memo1.Lines.Add(s);
        INC(LineNum,$10);
      end;
    finally
      Memo1.Lines.EndUpdate;
      FS.Free;
    end;
  end;
end;
```

Щелчок по пункту меню активирует диалог открытия файла. В диалоговом окне пользователь выбирает для просмотра любой файл. Данные из файла считываются в массив, состоящий из 16 ячеек. Затем, после нумерации строк и несложного форматирования, текст построчно передается в компонент `Memo1`. Операция считывания повторяется вновь до тех пор, пока не доберется до конца файла.

Программа почти готова. Нам осталось написать 3 строки кода в обработчике события `OnShow()` формы проекта (листинг 11.5).

Листинг 11.5. Настройка многострочного редактора и диалога открытия файла

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Memo1.Font.Name:='Courier New';
  Memo1.ReadOnly:=True;
  OpenDialog1.Filter:='Все файлы|*.*';
end;
```

Вам осталось откомпилировать проект и попробовать его в работе на каком-нибудь небольшом файле (рис. 11.2).

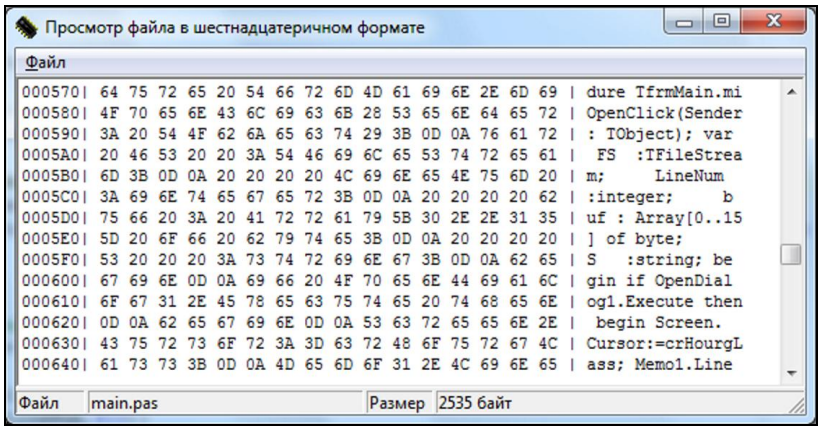


Рис. 11.2. Утилита просмотра файлов в шестнадцатеричном формате

Потоки данных в памяти

Платформой для классов, способных обслуживать данные, размещаемые в динамической памяти компьютера, служит класс `TCustomMemoryStream`. В момент создания экземпляра такого класса данные перемещаются в динамический буфер объекта, на который ссылается указатель

```
property Memory: Pointer;
```

Источником данных может стать файл, набор строк, графический образ, коммуникационный ресурс или любой другой объект с данными. Для считывания данных понадобится метод

```
function Read(var Buffer; Count: Longint): Longint; override;
```

Функция считывает `Count` байт данных в буфер `Buffer`.

Для перемещения по данным в потоке используйте метод `Seek()`, он уже подробно описан в подразделе, посвященном классу `TStream`.

Обработав данные в памяти, мы в состоянии сохранить их в файл с именем `FileName`:

```
procedure SaveToFile(const FileName: string);
```

либо передать в пользование другому потоку:

```
procedure SaveToStream(Stream: TStream);
```

Поток данных в памяти *TMemoryStream*

Экземпляр класса `TMemoryStream` предназначен для обслуживания данных, размещаемых в памяти компьютера. Кроме унаследованных от `TCustomMemoryStream` качеств наш новый знакомый может похвастаться процедурой, очищающей принадлежащую потоку область памяти.

```
procedure Clear;
```

У потока `TMemoryStream` развиты способности по загрузке внешних данных. Теперь у метода `Read()` появились коллеги. Процедура

```
procedure LoadFromFile(const FileName: string);
```

загрузит в память информацию из файла, а метод

```
procedure LoadFromStream(Stream: TStream);
```

воспользуется данными, принадлежащими другому потоку.

Если данные не загружаются из внешнего источника, а создаются динамически в коде программы, то для распределения (или перераспределения) буфера памяти следует воспользоваться свойством

```
property Capacity: Longint;
```

Если распределенной ранее памяти недостаточно (или слишком много), то для ее увеличения (уменьшения) обращайтесь к методу

```
procedure SetSize(NewSize: Longint); override;
```

Результаты работы метода найдут свое отражение в свойстве `Size`. Единственная разница между свойствами `Capacity` и `Size` в том, что значение `Capacity` может превышать значение `Size`.

> ***ЗАМЕЧАНИЕ***
>
> Обращение к процедуре `SetSize()` и свойству `Capacity` целесообразно осуществлять перед заполнением буфера данными.

Поток байтов *TBytesStream*

Класс `TBytesStream` является еще одной ступенькой вверх по лестнице VCL. Так же как и его предок, `TBytesStream` специализируется на обслуживании данных в динамической памяти компьютера, но в отличие от `TMemoryStream` наш новый знакомый является хорошим специалистом по работе с типизированными данными. В этом ему помогает изученный нами в *главе 8* класс `TArray`.

Для создания экземпляра класса предназначен конструктор

```
constructor Create(const ABytes: TBytes); overload;
```

Единственный параметр конструктора определяет обслуживаемые данные. После создания экземпляра потока доступ к этим данным обеспечит свойство

```
property Bytes: TBytes;//только для чтения
```

Поток строк *TStringStream*

До недавних пор класс `TStringStream` специализировался на обслуживании ANSI-строк, однако с переводом среды разработки на полную поддержку Unicode программисты Embarcadero столкнулись с дилеммой. С одной стороны, надо научить поток данных работать с текстом в формате Unicode, а с другой — обеспечить обратную совместимость между старыми ANSI-проектами и новой средой разработки. Выходом из положения стало следующее: в составе VCL был целиком и полностью сохранен старый класс, вдобавок в библиотеке (правда, в другой ветви наследования) появился одноименный класс (см. рис. 11.1), ориентированный исключительно на Unicode. Ответственность за "переключение" между классами была возложена на Delphi. Для этого в модуле `Classes` использовались соответствующие директивы компилятора (листинг 11.6).

Листинг 11.6. Переключение между кодировками Unicode и ANSI

```
{$IFDEF UNICODE}
  TStringStream = class(TBytesStream) //работаем с Unicode
...
{$ELSE}
  TStringStream = class(TStream) //работаем в старом формате
...
{$ENDIF}
  end;
```

Для того чтобы новая версия класса приобрела возможность работать с разными кодировками, в его распоряжении появилось полдюжины вариаций конструктора.

```
constructor Create; overload;
constructor Create(const AString: string); overload;
constructor Create(const AString: RawByteString); overload;
constructor Create(const AString: string; AEncoding: TEncoding;
                   AOwnsEncoding: Boolean = True); overload;
constructor Create(const AString: string; ACodePage: Integer); overload;
constructor Create(const ABytes: TBytes); overload;
```

Наибольший интерес представляет четвертый по счету конструктор, он определяет кодировку (параметр `AEncoding`) обслуживаемых текстовых данных.

Для выяснения, с какой кодировкой работает экземпляр класса, исследуем свойство

```
property Encoding: TEncoding;//только для чтения
```

Чтобы исключить вероятность искажения содержимого потока из-за изменения режима кодировки, свойство доступно только для чтения.

Записью и чтением текстовых данных занимаются методы

```
procedure WriteString(const AString: string);
function ReadString(Count: Longint): string;
```

Еще один способ чтения текста из потока предоставляет свойство

```
property DataString: string;//только для чтения
```

Поток с возможностью сжатия данных

Завершая обсуждение потоков данных, обсудим два очень примечательных класса — TZCompressionStream и TZDecompressionStream, позволяющих соответственно сжимать и восстанавливать данные без потерь. Указанные классы объявлены в модуле zlib и являются близкими родственниками классического потока TStream.

> ***Замечание***
>
> Основу программной библиотеки потоков сжатия и декомпрессии данных составляют разработки независимых программистов Ж. Гайлли (Jean-loup Gailly) и М. Адлера (Mark Adler), с которыми вы можете познакомиться на странице **http://www.gzip.org**.

Сжатие данных *TZCompressionStream*

Задачу сжатия данных решает класс TZCompressionStream. Экземпляр класса обычно создается для решения конкретной задачи — компрессии данных, находящихся во внешнем потоке. Поэтому уже в момент вызова конструктора следует указывать ссылку на файловый поток (поток памяти или любой другой потомок TStream), хранящий подлежащие компрессии данные.

```
constructor Create(Dest: TStream;
          CompressionLevel: TZCompressionLevel = zcDefault); overload;
constructor Create(CompressionLevel: TCompressionLevel;
          Dest: TStream); overload;
```

Поток Dest выступает получателем сжатых данных. Второй параметр, CompressionLevel, конструктора определяет степень сжатия. У нас всего четыре варианта выбора:

```
type TZCompressionLevel = (zcNone, zcFastest, zcDefault, zcMax);
```

После создания экземпляра потока TZCompressionStream настает черед метода CopyFrom(), позволяющего скопировать исходные (несжатые) данные из потока-источника.

Сам факт обращения к методу CopyFrom() инициирует процесс сжатия данных. Если размер исходных данных достаточно велик, то процесс компрессии можно контролировать с помощью обработчика события

```
property OnProgress: TNotifyEvent;
```

В рамках этого события следует обращаться к свойству

```
property CompressionRate: Single;
```

хранящему информацию о процентах выполнения задачи.

> ***Внимание!***
>
> Несмотря на то, что у класса опубликованы стандартные для всех потомков TStream методы чтения и перемещения по потоку (соответственно Read и Seek), обращение к ним нежелательно — в ответ мы получим сообщение об ошибке работы с потоком ECompressionError.

Благодаря потоку сжатия TZCompressionStream достаточно просто написать процедуру упаковки отдельного файла (листинг 11.7).

Листинг 11.7. Упаковка файла

```
uses zlib;
...
procedure SimpleCompressFile(SourceFile, DestinationFile: string);
var fsSource,fsDestination:TFileStream;
begin
  //загрузим исходный файл
  fsSource:=TFileStream.Create(SourceFile,fmOpenRead);
  //подготовим выходной файл
  fsDestination:=TFileStream.Create(DestinationFile,fmCreate);
  //создаем поток сжатия и передаем в него исходный файл
  with TZCompressionStream.Create(clDefault,fsDestination) do
  begin
    CopyFrom(fsSource,fsSource.Size);
    Free;
  end;
  //освобождаем все потоки
  fsDestination.Free;
  fsSource.Free;
end;
```

Разработанная нами процедура SimpleCompressFile() сожмет входной файл с именем SourceFile и сохранит результат в выходном файле DestinationFile.

Восстановление данных *TZDecompressionStream*

Задача восстановления данных из архива возлагается на класс TZDecompressionStream.

В момент создания экземпляра класса TZDecompressionStream следует указать исходный поток (в котором находятся упакованные данные) и передать его в конструктор объекта

```
constructor Create(Source: TStream);
```

Декомпрессия данных осуществляется автоматически сразу после создания объекта TZDecompressionStream.

Листинг 11.8 демонстрирует порядок работы с классом. В данном примере мы восстанавливаем файл с именем SourceFile и сохраняем восстановленные данные в файле DestinationFile.

Листинг 11.8. Восстановление файла

```
uses zlib;
...
procedure SimpleDeompressFile(SourceFile, DestinationFile: string);
var fsSource,fsDestination:TFileStream;
    nRead:Integer;
    buf: array[0..1023] of byte;
```

```
begin
  fsSource:=TFileStream.Create(SourceFile,fmOpenRead);{загрузили файл}
  fsDestination:=TFileStream.Create(DestinationFile,fmCreate);

  with TZDecompressionStream.Create(fsSource) do
    try
      Position:=0;
      repeat
        nRead:=Read(buf,SizeOf(buf));
        if (nRead<>SizeOf(buf)) and (nRead<>0) then
        begin
          {число прочитанных байтов отличается от размера буфера;
           это признак, что это последняя порция данных}
          fsDestination.Write(buf,nRead);
          break;
        end else fsDestination.Write(buf,SizeOf(buf));
      until nRead=0;
    finally
      Free;
    end;
  fsDestination.Free;
  fsSource.Free;
end;
```

Обратите внимание на особенность передачи данных из потока `TZDecompressionStream` в файловый поток `fsDestination`. Из-за того что нам неизвестен размер выходного файла, мы в рамках цикла `repeat..until` осуществляем чтения данных порциями по 1024 байта в буфер `buf`. Операции чтения продолжаются до тех пор, пока в экземпляре потока `TZDecompressionStream` имеются данные, контроль за этим нам помогает осуществить переменная `nRead`, в которой содержится число успешно прочитанных байтов. Не исключено, что размер восстановленных данных не будет кратен размеру нашего буфера чтения, поэтому самая последняя порция данных займет лишь какую-то часть в начале буфера, оставив в нем "мусор". Поэтому при каждой операции чтения мы проверяем количество успешно прочитанных байтов, и если оно отличается от размера буфера, то делаем вывод, что это заключительная операция чтения, и передаем в выходной файл только нужные байты.

ГЛАВА 12

Визуальные элементы управления и класс *TControl*

Сейчас нам предстоит еще глубже заглянуть в недра библиотеки компонентов Delphi и познакомиться с основоположником всех визуальных элементов управления — классом `TControl` (см. рис. 9.1). Вклад главного героя этой главы в развитие элементов управления VCL трудно не заметить. Визуальные компоненты наследуют от `TControl` свойства, методы и события, связанные с установкой местоположения компонента на форме, особенностями взаимодействия с командными объектами `TAction` и всплывающим меню, оперативной подсказкой, откликами на события мыши (в том числе операцию перетаскивания drag and drop).

Принадлежность к родительскому контейнеру

Во время визуального проектирования пользовательского интерфейса создатель приложения переносит требуемый элемент с палитры компонентов на рабочую форму проекта, размещает элемент управления в заданном месте и предает ему желаемые размеры. Благодаря высокотехнологичной интегрированной среде разработки (integrated development environment, IDE) все это делается так легко, что начинающий программист зачастую и не подозревает, что в это время происходит в проекте.

В момент попадания элемента управления на форму проекта в первую очередь ему присваивается имя (свойство `Name`) и назначается владелец (свойство `Owner`). Об этом вы узнали из материалов *главы 9*, посвященных классу `TComponent`. Затем для элемента управления назначается компонент-контейнер. Контейнер предоставит в распоряжение элементу управления часть своей поверхности (клиентской области). Элемент управления запоминает ссылку на родительский контейнер, для этого предназначено свойство

```
property Parent: TWinControl;
```

С ролью контейнера справится далеко не любой визуальный элемент управления. Для этого подойдут только *оконные элементы* (компоненты, предком которых выступает класс `TWinControl`). Язык Delphi не возражает против цепочек вложений одних элементов в другие, в таких цепочках компонент может являться контейнером для одних элементов управления и одновременно принадлежать другому родительскому контейнеру.

Предлагаю провести небольшой эксперимент со свойством `Parent`. Для этого на поверхности главной формы проекта следует поместить панель (компонент `TPanel`), затем на эту панель положить еще одну панель и т. д. На верхушку пирамиды из панелей "водрузим" обычную кнопку (рис. 12.1) и опишем обработчик события `OnClick()` по кнопке так, как предложено в листинге 12.1.

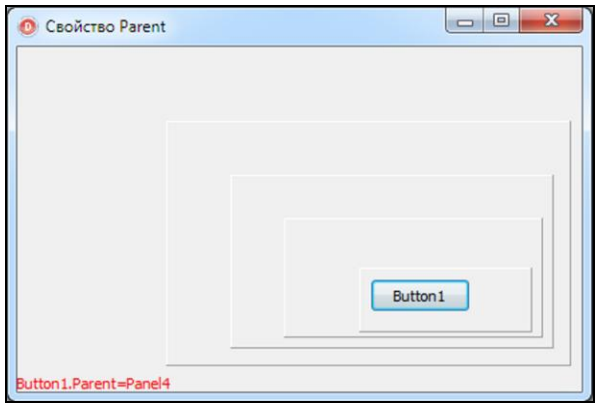


Рис. 12.1. Смена элементом управления родительского контейнера

Листинг 12.1. Смена родительского контейнера

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if Button1.Parent.Parent<>nil then
   Button1.Parent:=Button1.Parent.Parent;
end;
```

По щелчку кнопка `Button1` меняет свой родительский контейнер, поэтому компонент станет спускаться с панели на панель до тех пор, пока не окажется на поверхности формы `Form1`. Форма не принадлежит никакому из контейнеров, поэтому ее свойство `Parent` возвратит неопределенный указатель.

Размещение и размеры элемента управления

При размещении компонента на форме (или на любом другом компоненте, способном передавать свою клиентскую область в распоряжение элементов управления) следует учитывать, что в приложениях Windows за начальную точку системы координат (0, 0) принимается левый верхний угол клиентской части родительского контейнера. Из этой точки проведены две оси: ось *x* по горизонтали и по вертикали ось *y*. Ось *x* направлена слева направо, а ось *y* — сверху вниз (рис. 12.2).

Еще одна важная особенность приложений VCL в том, что за базовую единицу измерения по умолчанию принимаются не дюймы или миллиметры, а пикселы.

> ***Замечание***
>
> Пиксел — это минимальная видимая точка на вашем мониторе. Размеры пиксела определяются физическими характеристиками дисплея и выбранным разрешением экрана.

Место и размеры элемента управления хранятся в свойствах

```
property Left: Integer;{расстояние от левого края контейнера до элемента}
property Top: Integer;    {расстояние от верхнего среза
                           клиентской области контейнера}
property Height: Integer; {высота элемента управления}
property Width: Integer;  {ширина элемента управления}
```

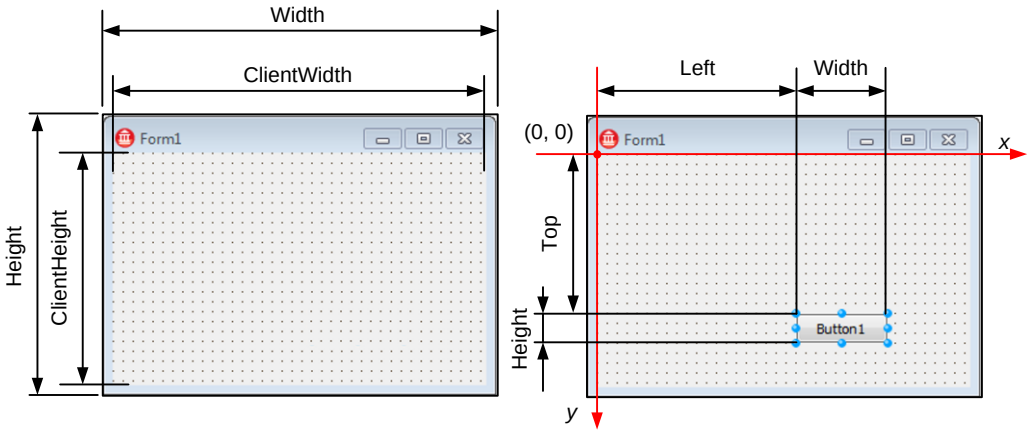


Рис. 12.2. Размеры и местоположение элемента управления

Альтернативный способ доступа к абсолютным размерам визуального элемента управления обеспечивает свойство

```
property BoundsRect: TRect;
```

В современной версии Delphi XE2 свойство возвращает прямоугольные координаты в формате интеллектуальной записи `TRect` с более чем 30 свойствами и методами. В простейшем случае, для чтения/записи координат можно обратиться к полям `Left`, `Top`, `Right`, `Bottom: Integer` или `TopLeft`, `BottomRight: TPoint`.

Наиболее универсальный способ присвоения элементу управления новых размеров и местоположения обеспечит метод

```
procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
```

Наряду с абсолютными размерами многие элементы управления характеризуются и размерами клиентской области. Под *клиентской областью* понимается внутреннее пространство объекта, на котором разрешено размещать другие элементы управления. Например, у формы в клиентскую область не входит заголовок и рамка окна (рис. 12.2). Для выяснения (а в ряде случаев и переопределения) размеров клиентской области элемента управления воспользуйтесь свойствами:

```
property ClientHeight: Integer; //высота клиентской области
property ClientWidth: Integer;  //ширина клиентской области
```

Координаты левой верхней точки клиентской области возвратит свойство

```
property ClientOrigin: TPoint;
```

Размеры и местоположение клиентской области элемента управления также можно получить, обратившись к свойству

```
property ClientRect: TRect; //только для чтения
```

Ряд потомков класса `TControl` (например, метка `TLabel`) умеют самостоятельно подстраивать свои размеры для наилучшего представления на экране своих данных (в случае метки — текста). Для активизации этой способности элемента управления следует установить в `true` свойство

```
property AutoSize: Boolean;
```

Программист обладает возможностью сформировать систему ограничений на предельные размеры элемента управления. Для этих целей предназначено свойство

```
property Constraints: TSizeConstraints;
```

Здесь можно определить ограничения на минимальные и максимальные значения вертикальных и горизонтальных размеров элемента управления (`MaxHeight`, `MaxWidth`, `MinHeight` и `MinWidth`).

События, связанные с изменением размеров

Некоторые потомки класса `TControl` обладают способностью реагировать на изменение своих размеров. Эта реакция выражается в форме трех последовательно вызываемых событий (табл. 12.1).

***Таблица 12.1.** События изменения размеров элемента управления*

Событие	Описание
`property OnCanResize: TCanResizeEvent;` `type TCanResizeEvent = procedure (Sender: TObject; var NewWidth, NewHeight: Integer; var Resize: Boolean) of object;`	Событие предваряет попытку элемента управления изменить свои габариты. Аргументы `NewWidth` и `NewHeight` содержат новые предполагаемые размеры объекта. Программист может согласиться с этими размерами или отказаться от предложенных значений. Для этого следует воспользоваться последним параметром `Resize`
`property OnConstrainedResize: TConstrainedResizeEvent;` `type TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth, MaxHeight: Integer) of object;`	Контролирует, чтобы размеры элемента управления оказались в пределах, определенных параметрами `MinWidth`, `MinHeight`, `MaxWidth` и `MaxHeight`
`property OnResize: TNotifyEvent;`	Уведомляет о том, что изменение размеров элемента управления завершено

Пересчет клиентских и экранных координат

Нередко возникает необходимость узнать координаты элемента управления не в клиентских координатах родительского контейнера, а в глобальных координатах экрана. Специалистом в этой области считается метод

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

Для решения обратной задачи — преобразования аппаратных координат определенной точки экрана к клиентским координатам — предназначен метод

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

Трансляцию клиентских координат в координаты родительского контейнера осуществит функция

```
function ClientToParent(const Point: TPoint; AParent: TWinControl = nil): TPoint;
```

Выравнивание элемента управления в контейнере

Простейший и одновременно наиболее востребованный способ выравнивания элемента управления в границах клиентской области контейнера предоставляет свойство

```
property Align: TAlign; //по умолчанию выключено (Align=alNone)
```

Свойство обеспечивает выравнивание по левой, правой, верхней или нижней границам контейнера. Кроме того, предусмотрен режим, предоставляющий в распоряжение компонента всю свободную клиентскую область контейнера. Порядок применения свойства иллюстрирует рис. 12.3. Например, при установке выравнивания в режим `alTop` элемент управления займет всю верхнюю часть контейнера.

> ***Замечание***
>
> Преимущество свойства `Align` в том, что при изменении размеров родительского контейнера элемент управления растягивается (сжимается) вместе с ним.

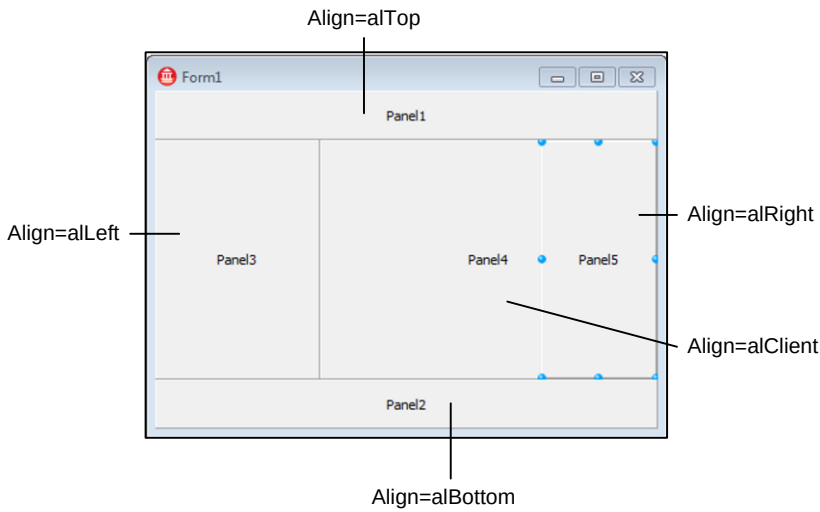


Рис. 12.3. Выравнивание элемента управления с помощью свойства `Align`

> ***Замечание***
>
> Если у элемента управления свойство `Align` установлено в состояние `alCustom`, то в процесс выравнивания можно вмешаться. Для этого предназначено событие `OnAlignPosition()`, оно генерируется у родительского контейнера, на поверхности которого выравнивается элемент управления.

Существует еще более интересный способ позиционирования элемента управления на поверхности контейнера. Он обеспечивается с помощью свойства

```
property Anchors: TAnchors;//по умолчанию Anchors = [akLeft, akTop]
```

Свойство предоставляет возможность привязки компонента к любой из четырех граней его контейнера. По умолчанию элемент управления прикрепляется к левой (`akLeft`) и верхней (`akTop`) границам контейнера.

Расположите на форме любой визуальный элемент управления, например кнопку, установите его свойство `Anchors` в режим `[akRight, akBottom]` и растяните форму. На рис. 12.4 демонстрируется поведение кнопки при изменении размеров родительской формы — элемент управления прочно "прикрепился" к правой и нижней границам формы.

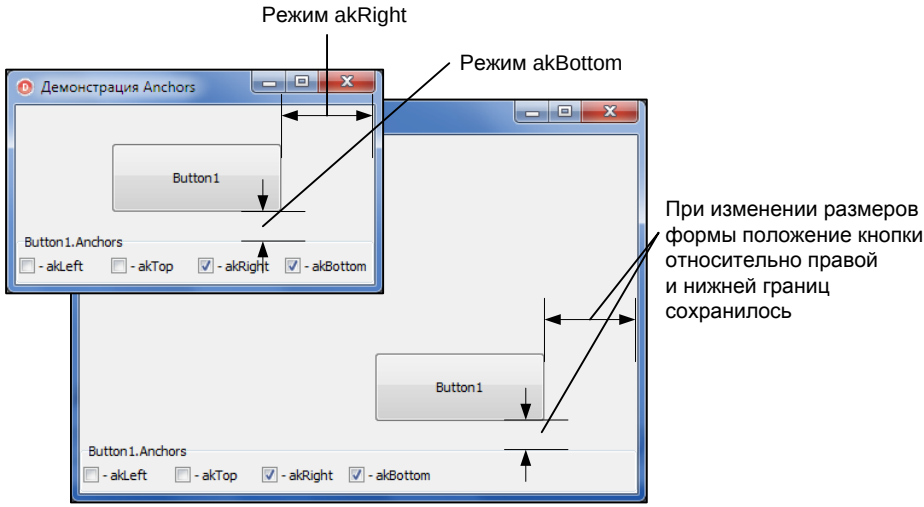


Рис. 12.4. Привязка кнопки к границам контейнера с помощью свойства `Anchors`

Видимость и активность элемента управления

Каждый визуальный элемент управления может лишиться права отображаться на экране компьютера. Для этого следует обратиться к методу

```
procedure Hide; //аналог visible:= false;
```

Для того чтобы компонент вновь появился на экране, вызывают процедуру

```
procedure Show; //аналог visible := true;
```

Вместо этих процедур можно воспользоваться свойством

```
property Visible: Boolean; //по умолчанию True
```

позволяющим включить (`true`) или отключить (`false`) отображение элемента.

> ***Замечание***
>
> Необходимым условием видимости элемента управления является видимость контейнера, на котором размещен этот элемент.

Очень часто вместо того, чтобы скрывать от пользователя ненужный компонент, программисты просто переводят его в пассивный режим. Для этого предназначено свойство

```
property Enabled: Boolean; //по умолчанию true
```

При установке свойства в состояние `false` элемент управления перестает реагировать на попытки обратиться к нему.

В случае, когда два элемента управления расположены один над другим (перекрывают друг друга), извлечь элемент на поверхность или спрятать его помогают методы

```
procedure BringToFront; //переместить вперед
procedure SendToBack;   //переместить назад
```

Эти методы доступны и во время визуального проектирования. Для их вызова щелкните правой кнопкой мыши по компоненту и выберите пункт контекстного меню **Control | Bring to front** (Переместить вперед) или **Control | Send to back** (Поместить назад).

Внешний вид

Сразу после размещения элемента управления на поверхности родительского контейнера свойствам, отвечающим за внешний вид компонента (в первую очередь, цвету и шрифту), передаются родительские настройки. Поведение по умолчанию можно и изменить. Свойства

```
property ParentColor: Boolean; //наследовать родительский цвет
property ParentFont: Boolean;  //наследовать родительский шрифт
```

разрешают (`true`) или запрещают (`false`) использовать настройки цвета и шрифта от родительского элемента управления.

Если вы рассчитываете настраивать цвет и шрифт индивидуально для каждого компонента, то обращайтесь к свойствам

```
property Color: TColor;
property Font: TFont;
```

Вывод текста

Практически любому элементу управления предоставлено право отображать (реже — редактировать) связанные с ним текстовые данные. Текст может содержаться в свойствах

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

У одного и того же компонента оба свойства одновременно не встречаются. Как правило, наличие свойства `Caption` свидетельствует о том, что элемент управления способен отображать пояснительные надписи. К таким компонентам относятся различного типа кнопки, метки, панели. Свойство `Text` подсказывает, что элемент управления в состоянии не только выводить текст на экран, но и предоставляет пользователю услуги по его редактированию.

Оперативная подсказка

Использование оперативной подсказки придает интерфейсу программного продукта дружественный вид. Подсказка появляется в момент остановки указателя мыши над элементом управления. Текст подсказки хранится в свойстве

```
property Hint: string;
```

Свойство одновременно может содержать два варианта подсказки, разделенных символом вертикальной черты `|`. Например, текст подсказки для кнопки закрытия проекта может выглядеть следующим образом:

```
btnClose.Hint := 'Выход|Завершение работы программы';
```

Для получения первой части подсказки следует воспользоваться функцией

```
function GetShortHint(const Hint: string): string;
```

Вторую часть возвратит функция

```
function GetLongHint(const Hint: string): string; ;
```

Подсказка подлежит выводу на экран только в том случае, если свойство

```
property ShowHint: Boolean;
```

у элемента установлено в `true`.

Некоторые права по управлению выводом подсказок имеются у родительского контейнера. Благодаря свойству

```
property ParentShowHint: Boolean;
```

контейнер способен централизованно разрешить или запретить отображать всплывающие подсказки всем принадлежащим ему объектам.

> ***ЗАМЕЧАНИЕ***
>
> К вопросу работы с оперативной подсказкой мы вернемся в *главе 14*, посвященной изучению приложения VCL — класса `TApplication`.

Контекстное меню

С каждым визуальным компонентом может быть сопоставлено контекстное (всплывающее) меню, появляющееся по щелчку правой кнопкой мыши или одновременному нажатию клавиш <Shift>+<F10>. Для присоединения меню к элементу управления предназначено свойство

```
property PopupMenu: TPopupMenu;
```

> ***ЗАМЕЧАНИЕ***
>
> На роль контекстного меню назначается компонент `TPopupMenu` или `TPopupActionBar`.

Если с компонентом связано контекстное меню и его свойство `AutoPopup` установлено в значение `true`, то в момент щелчка правой кнопкой мыши по элементу управления генерируется событие

```
property OnContextPopup: TContextPopupEvent;
type TContextPopupEvent = procedure(Sender: TObject; MousePos: TPoint;
                                    var Handled: Boolean) of object;
```

Командный объект

При разработке проектов большой и средней степени сложности, в особенности приложений с развитым интерфейсом управления, для централизации управления приложением программисты часто применяют особый вид невизуальных элементов управления, называемых *командами (см. главу 20)*. Этот тип объектов строится на основе класса `TBasicAction`. Для связи экземпляра класса `TControl` с командами предусмотрено свойство

```
property Action : TBasicAction;
```

Если такая связь установлена, то с этого момента элемент управления `TControl` начинает работать в интересах командного объекта. Его заголовок, всплывающая подсказка, значок изменяются в соответствии с настройками команды. Но самое главное в том, что подключенный к команде элемент управления может стать инициатором вызова ключевого события командного компонента `OnExecute()`.

Поддержка естественного ввода

С выходом в свет Delphi 2010 в языке программирования появилась уникальная технология естественного ввода. Нововведение позволяет программисту сопоставлять с элементом управления самые разнообразные (предопределенные заранее или определенные пользователем самостоятельно) жесты, представляющие собой простые геометрические фигуры (см. рис. 29.1), начерченные при помощи мыши, сенсорной панели, электронного пера или просто пальцем на чувствительном к прикосновениям экране.

Элемент управления отслеживает сопоставленный с ним жест и выполняет определенное действие, например стирает неверную строку в текстовом редакторе. Более подробно об этом мы поговорим в *главе 29*, а пока отметим, что для ассоциации жеста с потомком класса `TControl` следует обратиться к свойству:

```
property Touch: TTouchManager;
```

В момент выполнения действия генерируется событие

```
property OnGesture: TGestureEvent;
```

Обработка событий мыши

В классе `TControl` определена реакция на большую часть событий, инициируемых с помощью мыши. Эти события можно условно разделить на 4 группы:

- щелчки кнопками мыши по элементу управления;
- перемещение указателя мыши над элементом управления;
- вращение колесика мыши;
- операция перетаскивания drag and drop.

> ***ЗАМЕЧАНИЕ***
>
> Названия методов событий, связанных с реакцией на поведение мыши, начинаются со слова `OnMouse...`

Для того чтобы убедиться, что элемент управления способен реагировать на мышь, следует опросить свойство

```
property MouseCapture: Boolean;
```

Значение `true` свидетельствует о положительном ответе.

Щелчки кнопками мыши

Самым востребованным событием в Delphi является щелчок левой кнопкой мыши над областью компонента.

```
property OnClick: TNotifyEvent;
```

Замечание

Строго говоря, событие `OnClick()` может быть инициировано не только мышью. Событие может быть вызвано и с помощью клавиатуры: при нажатии клавиш управления курсором в таблицах, списках выбора, выпадающих списках; при нажатии "горячих" клавиш; при нажатии клавиш <Пробел> и <Enter> для элементов, на которых установлен фокус.

Событие `OnClick()` не только самое распространенное, но и наиболее простое с точки зрения обработки. Это событие у всех элементов управления описывается процедурой

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

обладающей всего одним параметром — `Sender`, в котором находится ссылка на объект, послуживший источником события.

В предыдущих главах мы уже встречались с примерами обработки события `OnClick()`, поэтому сейчас рассмотрим более усложненный пример, демонстрирующий возможность разделять один обработчик события сразу несколькими элементами управления и раскрывающий скрытые возможности параметра `Sender`. Для реализации примера разместите на главной форме проекта кнопку `Button1:TButton`, многострочный редактор `Memo1:TMemo` и несколько любых других визуальных элементов управления, которые вы найдете на палитре компонентов Delphi.

Выберите кнопку `Button1` и напишите всего одну строку кода в обработчике события (листинг 12.2).

Листинг 12.2. Получение имени компонента по параметру `Sender`

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.Add((Sender as TComponent).Name);
end;
```

Щелчок по кнопке приводит к появлению в многострочном редакторе `Memo1` строки с именем компонента, вызвавшего событие (в данном случае кнопки `Button1`). Предвижу вопрос: а не проще ли было написать строку `Memo1.Lines.Add('Button1')`? Нет, не проще. В особенности, если мы намерены научиться создавать разделяемый между несколькими компонентами код.

Выберите на форме любой другой компонент. Перейдите на вкладку **Events** (События) в Инспекторе объектов, в перечне событий найдите событие `OnClick` и сопоставьте с ним событие `Button1Click` (рис. 12.5). Наши действия приведут к тому, что два компонента станут разделять между собой один общий обработчик события. Аналогичным образом подключите к событию все остальные компоненты (включая форму) и запустите проект на выполнение. Щелкните по любому из компонентов — его имя сразу появится в отдельной строке многострочного редактора `Memo1`.

Внимание!

Параметр `Sender` имеется во всех обработчиках событий, благодаря ему программист сможет идентифицировать вызвавший событие элемент управления.

Благодаря наличию параметра `Sender` во всех обработчиках событий элементов управления Delphi и возможностью совместного использования несколькими объектами одного и того же события можно создавать весьма эффективный код. Хорошим примером такого кода может стать приложение Калькулятор.

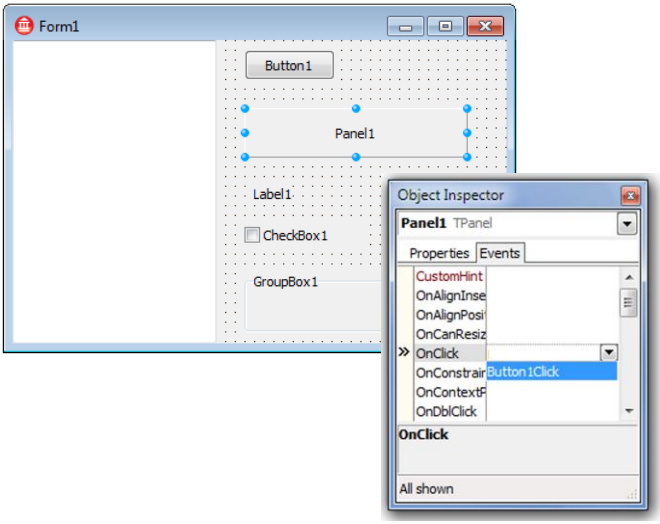


Рис. 12.5. Разделение одного обработчика события между несколькими компонентами

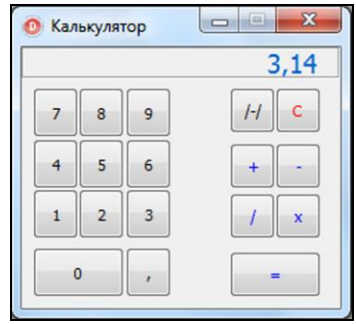


Рис. 12.6. Пользовательский интерфейс Калькулятора

У любого калькулятора есть 10 кнопок, отвечающих за ввод цифровых значений. Для того чтобы пользователь смог визуально идентифицировать назначение каждой из кнопок, в их заголовки (`Caption`) помещается соответствующий вводимой цифре символ (рис. 12.6). При нажатии на каждую из цифровых кнопок (в моем примере это быстрые кнопки `TSpeedButton`) генерируется событие `OnClick()`, в результате которого в метке `Label1:TLabel` Калькулятора отображается выбранная цифра. Как правило, начинающий программист решает подобную задачу в лоб и пишет для каждой из десяти кнопок примерно следующий код (листинг 12.3).

Листинг 12.3. Щелчок по кнопке "1" Калькулятора

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  if Label1.Caption='0' then Label1.Caption:= '1'
  else Label1.Caption:=Label1.Caption+'1';
end;
```

Более опытный разработчик поступит гораздо хитрее, обойдется услугами всего одного общего события `OnClick()` для всех кнопок, отвечающих за цифровой ввод (листинг 12.4), и разделит это событие между оставшимися 9-ю кнопками. Но теперь, вместо явного указания цифры, программист воспользуется услугами параметра `Sender`, содержащего ссылку на кнопку, по которой был произведен щелчок.

Листинг 12.4. Общее событие для всех цифровых кнопок Калькулятора

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  if Label1.Caption='0' then
    Label1.Caption:=(Sender as TSpeedButton).Caption
```

```
  else
    Label1.Caption:=Label1.Caption+(Sender as TSpeedButton).Caption;
end;
```

Процедура самостоятельно разберется, какая из кнопок послужила источником события, и добавит в заголовок метки `Label1` необходимую цифру.

Наряду с одинарным щелчком большой популярностью пользуется двойной щелчок кнопкой мыши над элементом управления. А в ответ на это у потомков `TControl` генерируется событие

```
property OnDblClick: TNotifyEvent;
```

Как видите, событие двойного щелчка кнопкой мыши типизирована процедурой `TNotifyEvent`, поэтому обработка двойного щелчка ведется по таким же правилам, что обработка одинарного.

Более широкие возможности предоставляют обработчики, раздельно контролирующие нажатие и отпускание кнопок мыши:

```
property OnMouseDown: TMouseEvent; //кнопка нажимается
property OnMouseUp  : TMouseEvent; //кнопка отпускается
```

где:

```
TMouseEvent = procedure (Sender: TObject;Button: TMouseButton;
                         Shift: TShiftState; X, Y: Integer) of object;
```

В табл. 12.2 представлен список параметров, позволяющих описывать реакцию элемента управления на событие.

Простейший прием обработки события нажатия на кнопку мыши предложен в листинге 12.5.

Таблица 12.2. *Описание параметров* `TMouseEvent`

Параметр	Возможные значения	Описание
`Sender`	Ссылка на объект	Ссылка на источник сообщения
`Button`	`mbLeft`	Щелчок левой кнопкой мыши
	`mbRight`	Щелчок правой кнопкой мыши
	`mbMiddle`	Щелчок центральной кнопкой мыши
`Shift`	`ssShift`	Удерживается в нажатом состоянии клавиша <Shift>
	`ssAlt`	Удерживается в нажатом состоянии клавиша <Alt>
	`ssCtrl`	Удерживается в нажатом состоянии клавиша <Ctrl>
	`ssLeft`	Нажимается/отпускается левая кнопка мыши
	`ssRight`	Нажимается/отпускается правая кнопка мыши
	`ssMiddle`	Нажимается/отпускается центральная кнопка мыши
	`ssDouble`	Двойной щелчок любой кнопкой
`X, Y`	`Integer`	Координаты указателя мыши

Листинг 12.5. Обработка события нажатия кнопки мыши

```
procedure TForm1.FormMouseDown(Sender: TObject;
            Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if (ssAlt in Shift)      //если удерживается клавиша <Alt>
    and (Button=mbRight)   //и пользователь нажимает правую кнопку мыши
  then
    //реакция на нажатие кнопки
end;
```

Перехват щелчков мыши

Сравнительно недавно в арсенале класса `TControl` появилось событие, позволяющее родительскому контейнеру реагировать на щелчки мышью, производимые по расположенному на контейнере элементу управления.

Родительский контейнер в рамках события

```
property OnMouseActivate: TMouseActivateEvent;
type TMouseActivateEvent = procedure (Sender: TObject;
       Button: TMouseButton; ShiftState: TShiftState; X, Y: Integer;
       HitTest: Integer; var MouseActivate: TMouseActivate) of object;
```

имеет возможность не просто реагировать на щелчок мышью по принадлежащему ему элементу управления, а даже перехватывать его. Это достигается за счет того, что событие `OnMouseActivate()` у контейнера (например, формы проекта) генерируется раньше, чем щелчок по принадлежащему контейнеру элементу управления (например, кнопке). Больше половины параметров события нам уже знакомо (табл. 12.2), поэтому заострим свое внимание только на новых. Параметр `HitTest` конкретизирует, по какой части элемента управления щелкнул пользователь. Параметр-переменная `MouseActivate` определяет реакцию на щелчок мышью и при необходимости запрещает генерацию события у дочернего элемента управления.

```
type TMouseActivate = (maDefault, //по умолчанию
       maActivate,           //активироваться
       maActivateAndEat,     //активироваться и запретить событие
       maNoActivate,         //не активироваться
       maNoActivateAndEat);  //не активироваться и запретить событие
```

Событие `OnMouseActivate` позволяет достигать весьма неординарных результатов. Разместите в центре главной формы проекта кнопку `Button1` и с помощью ее свойства `Anchors=[akLeft,akTop,akRight,akBottom]` привяжите кнопку к границам формы (рис. 12.7).

Воспользуйтесь событием `OnClick()` кнопки и напишите всего одну строку кода, закрывающую форму (листинг 12.6).

Листинг 12.6. Щелчок по кнопке закрывает форму

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Close; //закрываем форму
end;
```

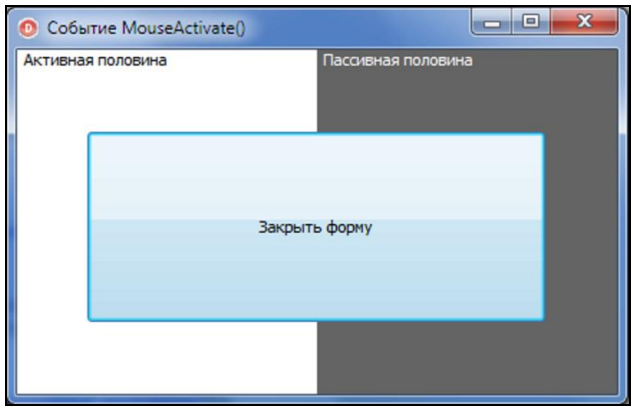


Рис. 12.7. Перехват щелчков по кнопке с помощью события `MouseActivate()`

Запустите приложение на выполнение и щелкните по кнопке — форма закроется, и работа приложения прекратится.

А теперь с помощью события `MouseActivate()` формы `Form1` попытаемся перехватить щелчок по кнопке (листинг 12.7).

Листинг 12.7. Щелчок по скрытой части формы

```
procedure TForm1.FormMouseActivate(Sender: TObject; Button: TMouseButton;
                      Shift: TShiftState; X, Y, HitTest: Integer;
                      var MouseActivate: TMouseActivate);
begin
  if HitTest = HTCLIENT then //щелчок в клиентской области формы
    if X<Form1.ClientWidth div 2 then
      MouseActivate := maActivate  //левая половина кнопки активна
    else
      MouseActivate := maNoActivateAndEat; //правая часть кнопки пассивна
end;
```

Мы сделали так, что на щелчок мышью способна реагировать только левая половина кнопки (координата указателя мыши `X` принадлежит левой половине формы). Все попытки щелкнуть по правой части кнопки перехватываются в событии `MouseActivate()` формы `Form1` и не приводят к закрытию формы.

Перемещение указателя мыши

Исследование реакции класса `TControl` на перемещение указателя мыши начнем с наиболее простых событий, позволяющих сигнализировать о факте нахождения указателя над тем или иным элементом управления.

В момент попадания указателя мыши в клиентскую область элемента управления генерируется событие

```
property OnMouseEnter: TNotifyEvent;
```

Выход указателя за пределы области сопровождается событием

```
property OnMouseLeave: TNotifyEvent;
```

> ***ЗАМЕЧАНИЕ***
>
> На уровне класса `TWinControl` объявлено свойство `MouseInClient`, позволяющее элементу управления проверить наличие указателя мыши над его клиентской областью.

При малейшем перемещении указателя мыши над поверхностью объекта у элемента управления генерируется событие:

```
property OnMouseMove: TMouseMoveEvent;
type TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState;
                           X, Y: Integer) of object;
```

Параметры события вам уже знакомы (см. табл. 12.2), аргумент `Shift` контролирует состояние служебных клавиш, координаты указателя в рамках клиентской области элемента управления доступны благодаря аргументам `X` и `Y`.

Воспользуемся услугами `OnMouseMove()` и напишем приложение, позволяющее пользователю перемещать компоненты по форме с помощью мыши. Для этого нам понадобится новый проект **VCL Forms Application** с любым визуальным элементом управления на главной форме (свой выбор я остановил на метке `Label1:TLabel`).

В разделе частных объявлений опишем две переменные — `xOffset` и `yOffset`, они позволят нам хранить отступы указателя мыши над перемещаемым компонентом (листинг 12.8). Значения отступов запоминаются в момент нажатия кнопки в событии `OnMouseDown()`. Перемещение осуществляется в рамках события `OnMouseMove()` (при условии, что пользователь удерживает в нажатом состоянии левую кнопку мыши).

Листинг 12.8. Перемещение элемента управления с помощью мыши

```
...
  private
    xOffset,yOffset:Integer;
  public
    { Public declarations }
  end;

var  Form1: TForm1;

implementation
{$R *.dfm}
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouseButton;
                                 Shift: TShiftState; X, Y: Integer);
begin
  xOffset:=X; //нажатие кнопки мыши
  yOffset:=Y; //запомнили отступы xOffset и yOffset
end;

procedure TForm1.Label1MouseMove(Sender: TObject;
                                 Shift: TShiftState; X,Y: Integer);
begin
```

```
  //перемещение мыши с нажатой левой кнопкой
  if (ssLeft in Shift) then
    with (Sender as TControl) do
    begin
      Left:=Left+X-xOffset;
      Top:=Top+Y-yOffset;
    end;
end;
```

Благодаря тому, что в параметре `Sender` находится ссылка на объект, над которым движется указатель мыши, мы вновь написали универсальный обработчик события, способный перемещать любой экземпляр класса `TControl`.

Вращение колесика мыши

Все современные мыши снабжены одним или несколькими колесиками, вращение которых позволяет пользователю управлять процессом перемещения по документу, сменой слайдов, переключением элементов управления и т. п. При повороте колесика мыши над элементом управления вызывается обработчик события

```
property OnMouseWheel: TMouseWheelEvent;
type TMouseWheelEvent = procedure(Sender: TObject; Shift: TShiftState;
                        WheelDelta: Integer; MousePos: TPoint;
                        var Handled: Boolean) of object;
```

Параметры события описаны в табл. 12.3.

***Таблица 12.3.** Описание параметров `TMouseWheelEvent`*

Параметр	Возможные значения	Описание
`Sender`	Ссылка	Ссылка на источник сообщения
`Shift`	`ssShift`	Удерживается в нажатом состоянии клавиша <Shift>
	`ssAlt`	Удерживается в нажатом состоянии клавиша <Alt>
	`ssCtrl`	Удерживается в нажатом состоянии клавиша <Ctrl>
	`ssLeft`	Нажата левая кнопка мыши
	`ssRight`	Нажата правая кнопка мыши
	`ssDouble`	Двойной щелчок любой кнопкой
`WheelDelta`	`integer`	Отрицательное значение — вращение вниз, положительное — вверх
`MousePos`	`TPoint`	Местоположение указателя мыши
`Handled`	`Boolean`	`False` — если обработкой события занимается родительский объект, иначе `True`

Листинг 12.9 демонстрирует один из вариантов управления полосой прокрутки `ScrollBar1:TScrollBar1` при посредничестве события `OnMouseWheel()` формы.

Листинг 12.9. Управления полосой прокрутки с помощью колесика мыши

```
procedure TForm1.FormMouseWheel(Sender: TObject; Shift: TShiftState;
                                WheelDelta: Integer; MousePos: TPoint;
                                var Handled: Boolean);
begin
  if WheelDelta>0 then ScrollBar1.Position:=ScrollBar1.Position-1;
  else
    if WheelDelta<0 then ScrollBar1.Position:=ScrollBar1.Position+1;
  Handled:=true;
end;
```

В листинге 12.9 мы разобрались с направлением вращения колесика мыши за счет исследования знака значения возвращаемого параметром `WheelDelta`. Создатели Delphi решили еще более упростить работу программиста с колесиком мыши. Для этого были реализованы два события, каждое из которых соответствует своему направлению вращения:

```
property OnMouseWheelUp: TMouseWheelUpDownEvent;   //вращение вверх
property OnMouseWheelDown: TMouseWheelUpDownEvent; //вращение вниз
type TMouseWheelUpDownEvent = procedure(Sender: TObject;
                              Shift: TShiftState; MousePos: TPoint;
                              var Handled: Boolean) of object;
```

Параметры событий аналогичны представленным в табл. 12.3.

Операция перетаскивания drag and drop

Класс `TControl` обеспечивает поддержку очень популярного действия drag and drop (перетащить и опустить). Эта операция позволяет реализовать операцию перетаскивания в пределах приложения, а при использовании разделяемой памяти — и между разными приложениями Delphi. Обычно перемещение происходит между двумя элементами управления, называемыми *источником* и *приемником*. Источник инициирует перетаскивание и содержит сам перетаскиваемый объект (клиент). Приемник принимает перетаскиваемый объект. Классическим примером реализации операции drag and drop служит Проводник Windows, предоставляющий услуги переноса файлов и папок при помощи операций перетаскивания.

Первым шагом на пути создания интерфейса перетаскивания должна стать настройка свойства

```
property DragKind: TDragKind;
type TDragKind = (dkDrag, dkDock);
```

В случае, если свойство находится в состоянии `dkDrag` — при перетаскивании реализуется технология drag and drop, в противном случае — drag and dock (операция буксировки элементов управления).

Способ работы с технологией перетаскивания:

```
property DragMode: TDragMode;
type TDragMode = (dmManual, dmAutomatic);
```

В ситуации, когда свойство переведено в автоматический режим (`dmAutomatic`), за старт процесса перетаскивания несет ответственность сама среда разработки Delphi. Несколько

сложнее работать в режиме ручного управления (dmManual). В таком случае программист обязан самостоятельно инициировать процесс перетаскивания, вызвав метод

```
procedure BeginDrag(Immediate: Boolean; Threshold: Integer = -1);
```

Метод рассчитан на то, что пользователь утопит кнопку мыши над объектом-клиентом и, удерживая эту кнопку в нажатом состоянии, "потащит" объект. Если параметр Immediate установлен в true, то операция перетаскивания начинается немедленно, в противном случае — после перемещения мыши на небольшое расстояние (количество пикселов указывается в Threshold).

При необходимости прервать процесс перетаскивания вызывается метод

```
procedure EndDrag(Drop: Boolean);
```

В случае если параметру Drop передано значение true, то результаты перетаскивания будут приняты. В противном случае произойдет отмена.

За внешний вид курсора при операции перетаскивания отвечает свойство:

```
property DragCursor: TCursor;
```

Если элемент-приемник по каким-то причинам отказывается получать данные от источника, то внешний вид курсора над ним будет соответствовать значению crNoDrop (перечеркнутый круг). Если все в порядке, то сигналом готовности станет курсор crDrag.

Процесс перетаскивания последовательно сопровождается четырьмя событиями (табл. 12.4).

Таблица 12.4. *Последовательность событий в операции drag and drop*

Событие	Инициатор	Описание
OnStartDrag()	Элемент-источник	Над элементом-источником нажата кнопка мыши и начато движение с удержанием кнопки в нажатом состоянии
OnDragOver()	Элемент-приемник	Перетаскиваемый элемент находится над приемником
OnDragDroup()	Элемент-приемник	Над элементом-приемником отпущена кнопка мыши
OnEndDrag()	Элемент-источник	Сигнализирует о завершении перетаскивания, используется для освобождения используемых ресурсов, отмены операции и т. п.

В момент начала перетягивания генерируется событие

```
property OnStartDrag: TStartDragEvent;
type TStartDragEvent = procedure(Sender: TObject;
                                 var DragObject: TDragObject) of object;
```

В программах Delphi обработчик события OnStartDrag() задействуется только при необходимости, например для инициализации переменных, вызова дополнительных процедур и функций. Необязательный формальный параметр DragObject применяется наиболее подготовленными программистами для создания собственного экземпляра класса TDragObject, позволяющего осуществлять более тонкую настройку операций перетаскивания.

Как мы уже говорили, событие OnDragOver возникает в то время, когда курсор мыши с перетаскиваемым клиентом движется над элементом-приемником, а также в момент отпускания

кнопки мыши. Обработчик события уведомляет о готовности элемента-получателя принять клиента.

```
property OnDragOver: TDragOverEvent;
TDragOverEvent = procedure(Sender, Source: TObject; X, Y: Integer;
                           State: TDragState; var Accept: Boolean) of object;
```

Параметры события приведены в табл. 12.5.

Таблица 12.5. *Параметры события* `OnDragOver()`

Параметр	Возможные значения	Описание
`Sender`	Любой потомок класса `TObject`	Ссылка на приемник
`Source`		Ссылка на источник
`X, Y`	`integer`	Координаты указателя над приемником
`State`	`dsDragEnter`	Указатель появился над приемником
	`dsDragLeave`	Указатель убран с поверхности приемника
	`dsDragMove`	Указатель движется над приемником
`Accept`	`Boolean`	Результат решения на прием клиента: • `true` — готовность к приему клиента; • `false` — отказ от приема

Задачей программиста является информирование приложения (через параметр `Accept`) о готовности или отказе приема операции drag and drop. В случае положительного решения при отпускании кнопки мыши возникает событие

```
property OnDragDrop: TDragDropEvent;
type TDragDropEvent = procedure(Sender, Source: TObject;
                                X, Y: Integer) of object;
```

Вне зависимости от готовности элемента-приемника к получению клиента, в финале перетаскивания элемент-источник генерирует событие:

```
property OnEndDrag: TEndDragEvent;
type TEndDragEvent = procedure(Sender, Target: TObject;
                               X, Y: Integer) of object;
```

В рамках обработчика события допускается освободить захваченные ранее ресурсы или выполнить какие-то другие действия, определяемые программистом.

Пример реализации операции drag and drop

Создайте новый проект. На главной форме проекта разместите два списка (класс `TListBox`). У обоих компонентов установите свойство `DragMode` в режим `dmManual`, а свойство `DragKind` в состояние `dkDrag`. Воспользовавшись свойством `Items` любого из списков, внесите в него несколько текстовых строк. Именно эти строки мы станем перетаскивать из одного элемента управления в другой.

В секции `private` главной формы объявите переменную `Index`, она станет хранить индекс перетаскиваемой строки. Далее нам предстоит последовательно описать 5 обработчиков

событий. Первое из них — событие движения мыши над списком (OnMouseMove), оно предназначено для инициализации операции перетягивания. Оставшийся квартет событий соответствует событиям, представленным в табл. 12.4.

Код, необходимый для реализации интерфейса перетаскивания, можно изучить в листинге 12.10.

Листинг 12.10. Реализация интерфейса перетаскивания между двумя списками

```
private
  Index : integer;
  //...
procedure TForm1.ListBox1MouseMove(Sender: TObject;
                                   Shift: TShiftState; X, Y: Integer);
begin
  {1. Инициируем процесс перетаскивания}
  if ((Sender as TListBox).ItemIndex<>-1){есть выделенный элемент}
     and (ssLeft in Shift) then          {нажата левая кнопка мыши}
   (Sender as TListBox).BeginDrag(false,8); {старт операции Drag}
end;

procedure TForm1.ListBox1StartDrag(Sender: TObject;
                                   var DragObject: TDragObject);
begin
  {2. Запоминаем индекс перетаскиваемого элемента}
  Index:=(Sender as TListBox).ItemIndex;
end;

procedure TForm1.ListBox1DragOver(Sender, Source: TObject; X, Y: Integer;
                                  State: TDragState; var Accept: Boolean);
begin
  {3. Проверяем приемлемость данных}
  Accept:=(Index<>-1) and (Sender<>Source);
end;

procedure TForm1.ListBox1DragDrop(Sender, Source: TObject;
                                  X, Y: Integer);
var s:string;
begin
  {4. Передаем строку из списка Source в список Sender}
  s:=(Source as TListBox).Items.Strings[Index]; //узнаем текст элемента
  (Sender as TListBox).Items.Add(s); //передаем текст в приемник
  (Source as TListBox).Items.Delete(Index);//удаляем элемент из источника
end;

procedure TForm1.ListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  {5. Завершаем перетаскивание}
  Index:=-1;
end;
```

Мы с вами постарались написать универсальный код, который позволит перетаскивать элементы из списка в список. Для того чтобы приложение стало работоспособным, не забудьте сделать все события общими для `ListBox1` и `ListBox2`.

Замечание

У операции перетягивания есть "коллега" — операция буксировки drag and dock. Она реализуется совместными усилиями классов `TControl` и `TWinControl`. О буксировке мы поговорим в *главе 13*, которая посвящена потомкам класса `TWinControl`.

ГЛАВА 13

Оконные элементы управления и класс *TWinControl*

Изучаемый в этой главе класс называется `TWinControl`, приставкой "Win" разработчики акцентируют внимание на том, что класс является предтечей всех оконных (window) элементов управления в библиотеке VCL. У потомков `TWinControl` выделяются три ключевые особенности:

- наличие *дескриптора окна* — связующего звена с функциями *прикладного интерфейса программирования* (application program interface, API) Windows;
- способность получать *фокус ввода*, что позволяет потомкам класса реагировать на события клавиатуры компьютера;
- возможность выступать в качестве контейнера для других элементов управления.

Класс `TWinControl` выступает некой разделительной чертой между фундаментальными классами визуальной библиотеки VCL (`TObject`, `TPersistent` и т. д.), определяющими общие характеристики будущих элементов управления Delphi, и многочисленными специализированными классами, на основе которых создаются реальные *оконные элементы управления* (рис. 13.1). Близость к границе предъявляет к `TWinControl` дополнительные требования: с одной стороны, класс все еще не предназначен для непосредственного создания элементов управления, а с другой стороны, он выступает прямым предком ряда полноценных элементов управления (например, полосы прокрутки `TSctrollBar`), которые вы легко найдете на палитре компонентов Delphi.

Дескриптор окна

Класс `TWinControl` способен самостоятельно обращаться к наиболее важным для его жизнедеятельности функциям прикладного программного интерфейса Windows. Кроме того, для взаимодействия с функциями API класс оснащен свойством

```
property Handle: HWND; //только для чтения
```

Это дескриптор оконного элемента управления, описываемый типом данных `HWND`. Физически `HWND` представляет собой беззнаковое целое типа `LongWord`. Значение, хранящееся в дескрипторе, служит уникальной ссылкой на описывающую окно структуру в памяти компьютера. Создавая оконный элемент управления, Windows автоматически заполняет и хранит дескриптор этого объекта. Благодаря такой незримой нити операционная система всегда знает, как добраться до нужного объекта-окна.

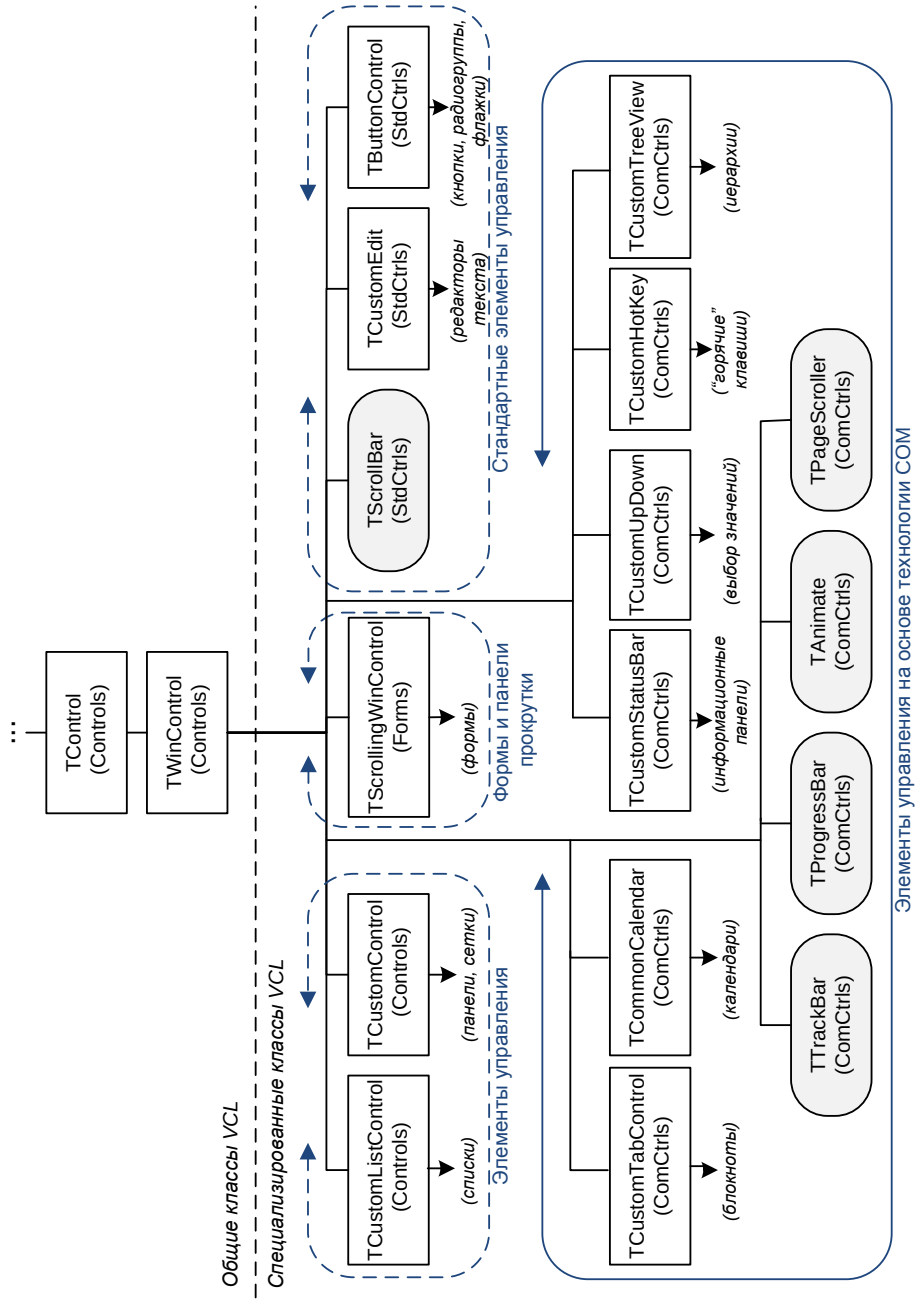


Рис. 13.1. Фрагмент иерархии VCL наследников класса `TWinControl`

Листинг 13.1 демонстрирует пример управления оконными элементами управления с помощью функции `SetWindowRgn()`, способной изменять форму любого окна. Для этого в первый параметр функции следует передать дескриптор модифицируемого окна, а во второй — характеристики нового региона.

Листинг 13.1. Изменение формы элемента управления через дескриптор окна `Handle`

```
var Rgn:HRGN;
    Rect:TRect;
begin
  Rect:=Form1.BoundsRect;      {прямоугольная область окна}
  Rgn:=CreateEllipticRgn(0,0, {определяем эллиптический регион}
      Rect.Right-Rect.Left, Rect.Bottom-Rect.Top);
  SetWindowRgn(Form1.Handle,Rgn,true); {изменяем очертание формы Form1}
  DeleteObject(Rgn);
end;
```

В нашем примере мы изменяем очертание главной формы приложения `Form1`, превращая привычное прямоугольное окно в эллиптическое. Эту же операцию несложно осуществить и со всеми элементами управления, созданными на базе класса `TWinControl`. Свидетельство тому — рис. 13.2. На нем представлены экранные снимки одного и того же приложения с элементами управления стандартной и модифицированной формами.

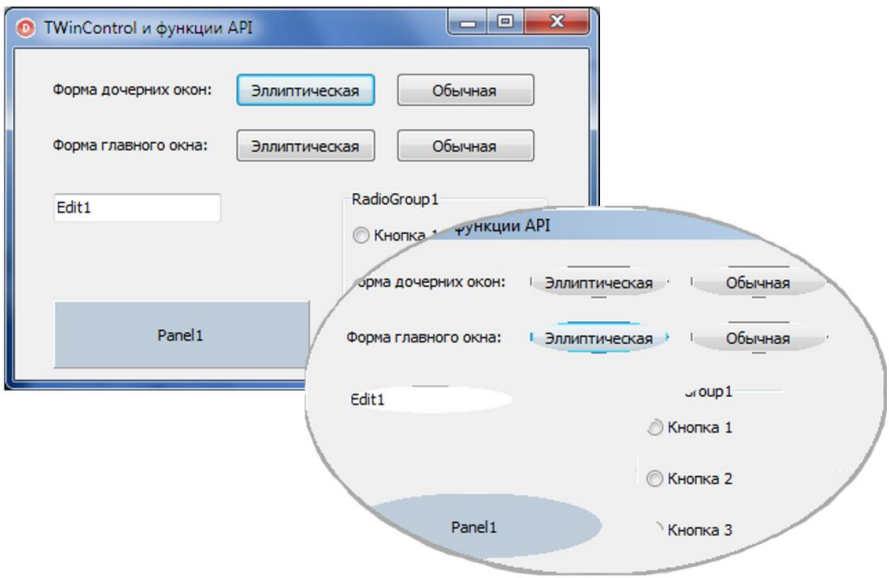


Рис. 13.2. Изменение формы окон с помощью определения региона

Управление подчиненными элементами

Важной особенностью потомков класса `TWinControl` считается возможность служить родительским контейнером для других элементов управления. Об этом мы уже говорили в *главе 12*, рассматривая свойство `Parent`.

Общее количество этих элементов содержится в свойстве

```
property ControlCount: Integer;
```

Дочерние элементы хранятся в списке, доступ к каждому из них производится по его порядковому номеру:

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

Проверка наличия элемента в списке осуществляется методом

```
function ContainsControl(Control: TControl): Boolean;
```

В случае успеха функция возвратит значение `true`.

Состав списка доступен для редактирования благодаря паре методов:

```
procedure InsertControl(AControl: TControl); //вставка элемента в список
procedure RemoveControl(AControl: TControl); //удаление элемента
```

Любой потомок `TWinControl` способен проверить наличие элемента управления в точке с клиентскими координатами `Pos`:

```
function ControlAtPos(const Pos: TPoint;  AllowDisabled: Boolean): TControl;
```

Параметр `AllowDisabled` разрешает или запрещает поиск среди элементов в зависимости от состояния их свойства `Enabled`. Если компонент обнаружен, то функция возвратит ссылку на него, в случае неудачи результат поиска будет равен `nil`.

Родительский оконный элемент управления способен разослать всем элементам своего списка сообщение

```
procedure Broadcast(var Message);
```

Сообщение в формате структуры `TMessage` будет поочередно отправлено всем подчиненным элементам родительского контейнера. Для ускорения процесса сообщение не ставится в очередь сообщений, а будет передано непосредственно в оконную процедуру каждого принадлежащего контейнеру окна.

Родительский контейнер имеет возможность дать команду на показ невидимого подчиненного элемента управления, для этого следует воспользоваться методом

```
procedure ShowControl(AControl: TControl);
```

Выравнивание подчиненных элементов

Родительский контейнер имеет возможность вмешаться в процесс выравнивания принадлежащего ему элемента управления. Единственным на то условием должно стать согласие элемента управления, оно выражается в установке в состояние `alCustom` свойства `Align`.

Реакция на выравнивание описывается в рамках события

```
property OnAlignPosition: TAlignPositionEvent;
type TAlignPositionEvent = procedure(Sender: TWinControl;
    Control: TControl; var NewLeft, NewTop, NewWidth, NewHeight: Integer;
                var AlignRect: TRect; AlignInfo: TAlignInfo) of object;
```

Первые два параметра содержат ссылки на родительский контейнер `Sender` и выравниваемый элемент управления `Control`. Следующая четверка аргументов — `NewLeft`, `NewTop`, `NewWidth` и `NewHeight` — определяет положение и размеры выравниваемого компонента. `AlignRect` содержит прямоугольные координаты клиентской области родительского кон-

тейнера, в которой осуществляется выравнивание. Заключительный параметр `AlignInfo` содержит вспомогательную информацию.

> ***Замечание***
>
> Событие `OnAlignPosition()` генерируется после изменения размеров родительского контейнера. Событие будет вызвано ровно столько раз, сколько элементов управления со свойством `Align=alCustom` расположено на контейнере.

Для каждого выравниваемого в ручном режиме элемента управления событие `OnAlignPosition()` вызывается в том порядке, в котором элементы управления хранятся в свойстве `Controls` своего контейнера. Поэтому, когда контейнеру принадлежит несколько элементов управления, нельзя исключить вероятность, что при осуществлении выравнивания один элемент помешает другому. В такой ситуации желательно уточнять порядок выравнивания элементов управления. Например, функция

```
function CustomAlignInsertBefore(C1, C2: TControl): Boolean;
```

возвратит значение `true`, если в очереди выравнивания элемент управления `C2` находится перед `C1`. Результат `false` говорит об обратном — первым в очереди стоит `C1`.

Если подлежащих выравниванию в ручном режиме элементов много, то стоит обратить внимание на событие

```
property OnAlignInsertBefore: TAlignInsertBeforeEvent;
TAlignInsertBeforeEvent = function(Sender: TWinControl;
                                   C1, C2: TControl): Boolean of object;
```

Функциональная нагрузка события точно такая же, как и у метода `CustomAlignInsertBefore()` — проверка очередности следования выравниваемых элементов управления, но на этот раз событие вызывается автоматически для всех возможных комбинаций пар выравниваемых элементов.

Фокус ввода

Фокус ввода играет ключевую роль в процессе взаимодействия пользователя и элементов управления. Наличие фокуса ввода свидетельствует о том, что элемент управления находится в готовности к получению и обработке входящих сообщений от клавиатуры. Благодаря фокусу ввода операционная система распознает, с каким именно элементом управления в данный момент работает пользователь, после этого все управляющие воздействия от клавиатуры направляются только к этому элементу.

Для проверки нахождения оконного элемента управления в фокусе предназначен метод

```
function Focused: Boolean; // возвращает true в случае успеха
```

Особо нетерпеливый элемент управления имеет возможность потребовать от системы предоставления ему фокуса ввода

```
procedure SetFocus;
```

Однако в некоторых случаях получение фокуса ввода оконным элементом управления попросту невозможно. Такая ситуация возникает, когда объект расположен на неактивном окне, невидим или вообще физически отключен. Если в таких обстоятельствах мы попытаемся программным образом передать фокус компоненту, то результатом станет генерация исключительной ситуации. Поэтому перед вызовом метода `SetFocus()` следует предварительно проверить готовность объекта получить фокус. Для этого предназначен метод:

```
function CanFocus: Boolean; //в случае успеха возвращает true
```

Все элементы управления, способные получить фокус ввода, содержатся в особом списке. Доступ к нему осуществляется при помощи процедуры:

```
procedure GetTabOrderList(List: TList);
```

Необходимым условием нахождения элемента в этом списке является установка в `true` его свойства:

```
property TabStop: Boolean;
```

Очередность перехода фокуса между элементами управления определяется порядковым номером в свойстве:

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

> ***ЗАМЕЧАНИЕ***
>
> Во время визуального проектирования для установки порядка смены фокуса между элементами управления следует воспользоваться контекстным меню контейнера (например, формы проекта) и элементом меню **Tab Order**.

События перехода фокуса происходят при щелчке мышью по компоненту или нажатии клавиши <Tab>. Процесс получения и потери фокуса ввода соответственно сопровождается двумя событиями:

```
property OnEnter: TNotifyEvent; //объект получает фокус ввода
property OnExit : TNotifyEvent; //объект теряет фокус ввода
```

Обработка событий клавиатуры

Благодаря умению получать фокус ввода потомки класса `TWinControl` способны реагировать на события клавиатуры. Клавиатурных событий всего три, и они возникают в строгой последовательности:

1. В момент опускания клавиши у находящегося в фокусе ввода оконного элемента управления генерируется событие `OnKeyDown()`.
2. В момент фиксации клавиши в нажатом положении вызывается событие `OnKeyPress()`.
3. Подъем клавиши сопровождается событием `OnKeyUp()`.

События опускания и подъема клавиши (первое и третье в нашем списке) описываются идентичным набором параметров:

```
property OnKeyDown: TKeyEvent;
property OnKeyUp: TKeyEvent;
type TKeyEvent = procedure(Sender: TObject; var Key: Word;
                           Shift: TShiftState) of object;
```

Здесь: `Sender` — ссылка на элемент управления, в котором возникло событие; `Shift` — индикатор состояния служебных клавиш <Shift>, <Ctrl> и <Alt>; `Key` — код клавиши (табл. 13.1).

Листинг 13.2 демонстрирует порядок применения события `OnKeyDown()`, в его рамках форма отслеживает коды клавиш. Если пользователь нажимает клавиши управления курсором и при этом удерживает клавишу <Shift>, то форма изменяет свой размер.

Таблица 13.1. *Коды клавиш*

Код клавиши	Клавиша	Код клавиши	Клавиша
VK_CANCEL	<Ctrl>/<Break>	VK_PRIOR	<Page Up>
VK_BACK	<Backspace>	VK_NEXT	<Page Down>
VK_TAB	<Tab>	VK_END	<End>
VK_RETURN	<Enter>	VK_HOME	<Home>
VK_SHIFT	<Shift>	VK_LEFT	<←>
VK_CONTROL	<Ctrl>	VK_UP	<↑>
VK_MENU	<Alt>	VK_RIGHT	<→>
VK_PAUSE	<Pause>	VK_DOWN	<↓>
VK_CAPITAL	<CapsLock >	VK_INSERT	<Ins>
VK_ESCAPE	<Esc >	VK_DELETE	<Del >
VK_SPACE	<Пробел >	VK_NUMLOCK	<Num Lock>
VK_SNAPSHOT	<Print Screen>	VK_SCROLL	<Scroll Lock>
Цифровые и символьные клавиши основной клавиатуры			
VK_0, ..., VK_9	<0>, ..., <9>	VK_A, ..., VK_Z	<A>, ..., <Z>
Функциональные клавиши			
VK_F1, ..., VK_F12		<F1>, ..., <F12>	
Дополнительная цифровая клавиатура (Num Lock включен)			
VK_NUMPAD0, ..., VK_NUMPAD9		<0>, ..., <9>	
VK_MULTIPLY	<*>	VK_SUBTRACT	<–>
VK_ADD	<+>	VK_DIVIDE	</>
VK_DECIMAL		<.>	

Листинг 13.2. Управление размерами формы в событии `OnKeyDown()`

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
                             Shift: TShiftState);
begin
if (ssShift in Shift) then
    case Key of
      VK_LEFT : Form1.Width:=Form1.Width-1;
      VK_UP   : Form1.Height:=Form1.Height-1;
      VK_RIGHT: Form1.Width:=Form1.Width+1;
      VK_DOWN : Form1.Height:=Form1.Height+1;
    end;
end;
```

В момент фиксации клавиши в нажатом состоянии генерируется событие:

```
property OnKeyPress: TKeyPressEvent;
type TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Обратите внимание на то, что на этот раз параметр-переменная Key в обработчике отслеживает не код клавиши, а выбранный пользователем символ. Благодаря этому можно проверять корректность вводимого текста (листинг 13.3).

Листинг 13.3. Ввод 11-значного телефонного номера

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if ((KEY<'0') or (KEY>'9') or (Length(Edit1.Text)>=11)) and (KEY<>#8) then  {#8 —
код клавиши Backspace}
  begin
    Key:=#0;
    if (Length(Edit1.Text)>=11)=false then
      ShowMessage('Допускаются только цифры!')
    else
      ShowMessage('Длина текста превысила 11 знаков!')
  end;
end;
```

Событие осуществляет контроль верности ввода телефонного номера в строку ввода Edit1. Процедура допускает ввод только цифр, причем длина номера не должна превышать 11 символов. В противном случае ввод ошибочного символа отменяется (параметру-переменной Key присваивается значение #0) и выдается сообщение об ошибке.

> ***ЗАМЕЧАНИЕ***
>
> Ограничения на ввод текстовых данных удобно накладывать с помощью компонента TMaskeEdit. Данный элемент позволяет сверять вводимые данные с предопределенным шаблоном.

Операция буксировки drag and dock

Рассмотренная в *главе 12* операция drag and drop и изучаемая сейчас операция буксировки — ближайшие родственники. Разница между ними в том, что операция drag and dock выполняет буксировку элемента управления, а не просто данных. Буксировка осуществляется между контейнерами (доками) — объектами, способными размещать на своей поверхности другие компоненты. Необходимое условие для того, чтобы элемент управления смог стать доком, — наличие в его цепочке наследования класса TWinControl.

> ***ЗАМЕЧАНИЕ***
>
> Описание и реализация свойств, методов и обработчиков событий операции буксировки распределены между классами TControl и TWinControl. Класс TControl в основном отвечает за описание буксируемого элемента управления, класс TWinControl предоставляет сервис дока.

Если мы планируем внедрить в наше приложение технологию буксировки, первым действием должен быть выбор компонентов-контейнеров, потенциальных носителей буксируемых элементов. Признаком того, что элемент управления способен превратиться в док, служит наличие у него свойства:

```
property DockSite: Boolean;
```

При установке свойства в `true` элемент управления уведомляет буксируемый объект о готовности предоставить ему "посадочную площадку". При этом для автоматического масштабирования "посадочного" пространства переведите в `true` свойство

```
property AutoSize: Boolean;
```

> ***Внимание!***
>
> При изучении технологии drag and drop мы уже столкнулись со свойствами `DragKind` и `DragMode`. При организации буксировки первое из них устанавливается в состояние буксировки `DragKind:=dkDock`, а второе в ручной (`dmManual`) или автоматический (`dmAutomatic`) режим буксировки.

Имеющихся у нас знаний уже вполне достаточно для того, чтобы создать приложение, поддерживающее технологию буксировки. Для этого разместите на главной форме проекта две панели группировки (`GroupBox1` и `GroupBox2`) и на одну из панелей положите обычную кнопку `Button1`. В событии `OnShow()` главной формы проекта напишите строки из листинга 13.4.

Листинг 13.4. Реализация операции перетаскивания между двумя списками

```
procedure TForm1.FormShow(Sender: TObject);
begin
  GroupBox1.DockSite:=True;       //элементы-доки
  GroupBox2.DockSite:=True;       //элементы-доки
  Button1.DragKind:=dkDock;       //активация режима буксировки
  Button1.DragMode:=dmAutomatic; //автоматическая буксировка
end;
```

Работу листинга демонстрирует рис. 13.3, на нем представлены три этапа буксировки кнопки из дока 1 в док 2.

Буксируемый элемент вовсе не обязан приземляться на поверхность другого дока. Допускается отпускать кнопку мыши в любом месте. В таком случае Delphi автоматически создаст временную форму-док, а буксируемые компоненты превратятся в плавающую панель инструментов. За класс создаваемой формы-дока отвечает свойство

```
property FloatingDockSiteClass: TWinControlClass;
```

> ***Замечание***
>
> Если элемент управления помещен на компонент-док во время разработки приложения, это еще не означает, что этот элемент стал клиентом дока. Чтобы наверняка закрепить компонент за доком, в обработчике события создания формы надо поместить примерно такую строку: `Button1.HostDockSite:=GroupBox1;`.

Став доком, элемент управления приобретает ряд обязательств по отношению к размещенным на нем клиентам. Он должен знать их общее количество:

```
property DockClientCount: Integer;
```

Более того, он обязан обеспечить доступ к любому из них по индексу:

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

Клиент, находясь на поверхности дока, также не остается в неведении. В его свойстве

```
property HostDockSite: TWinControl;
```

хранится ссылка на док.

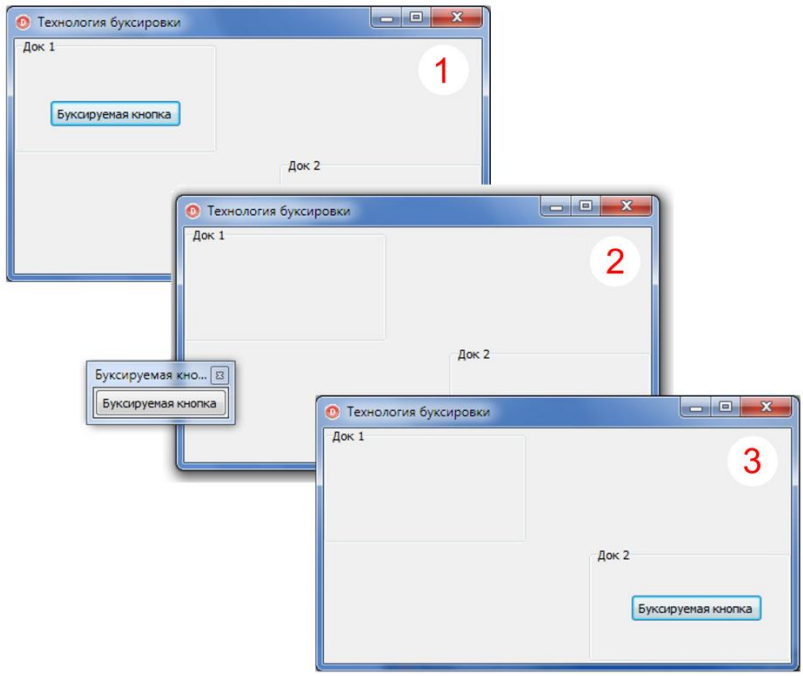


Рис. 13.3. Этапы буксировки кнопки

Во время буксировки у вовлеченных в процесс элементов управления генерируется ряд событий (табл. 13.2).

***Таблица 13.2.** События буксировки*

Событие	Инициатор	Описание
`OnStartDock()`	Буксируемый объект	Начало процесса буксировки
`OnGetSiteInfo()`	Все потенциальные доки получатели	Обмен информацией дока и клиента
`OnDockOver()`	Док-получатель	Перемещение указателя мышки с буксируемым объектом над доком
`OnDockDrop()`	Док-получатель	Док принимает буксируемый объект
`OnUnDock()`	Док-источник	Уход буксируемого объекта из первоначального дока, сразу после отпускания кнопки мыши
`OnEndDock()`	Буксируемый объект	Извещение о завершении буксировки

В начале буксировки переносимый объект инициирует обработку события:

```
property OnStartDock: TStartDockEvent;
type TStartDockEvent = procedure(Sender: TObject;
                       var DragObject: TDragDockObject) of object;
```

Здесь: `Sender` — непосредственно переносимый элемент; `DragObject` — временный объект, создаваемый только на время переноса и содержащий свойства переносимого элемента.

В самом начале транспортировки буксируемый объект автоматически оповещает все потенциальные доки (элементы со свойством DockSite, равным true) о своем намерении переместиться на новый контейнер. На это все потенциальные доки откликаются событием

```
property OnGetSiteInfo: TGetSiteInfoEvent;
type TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl;
                         var InfluenceRect: TRect; MousePos: TPoint;
                         var CanDock: Boolean) of object;
```

Здесь: Sender — ссылка на док; DockClient — ссылка на буксируемый объект; InfluenceRect — координаты прямоугольной области дока; MousePos — координаты указателя мыши. Основной параметр-переменная CanDock в состоянии true уведомляет буксируемый объект о своей готовности принять его.

Разместите на форме полосу TCoolBand (этот компонент вы обнаружите на страничке **Win32** палитры компонентов). Укажите, что полоса имеет право выступать в качестве дока, для этого установите свойства DockSite в состояние true. Разместите на полосе панель инструментов ToolBar1 (страничка **Win32**). Для того чтобы элемент управления ToolBar1 смог реагировать на процесс буксировки, потребуется перевести свойство DragKind в режим dkDock, а свойство DragMode — в режим dmAutomatic.

Листинг 13.5 демонстрирует порядок уведомления буксируемых объектов о том, что док CoolBar1 предоставит свою поверхность только потомкам класса TToolBar.

Листинг 13.5. Информирование о доке

```
procedure TForm1.CoolBar1GetSiteInfo(Sender: TObject;
      DockClient: TControl; var InfluenceRect: TRect; MousePos: TPoint;
      var CanDock: Boolean);
begin
  CanDock:=(DockClient is TToolBar);//готов стать доком для TToolBar
end;
```

Очередные события буксировки OnDockOver и OnDockDrop генерируются у дока-получателя. Первое из них

```
property OnDockOver: TDockOverEvent;
type TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject;
                          X, Y: Integer; State: TDragState;
                          var Accept: Boolean) of object;
```

возникает, когда док Sender принимает решение о предоставлении буксируемому объекту Source своей поверхности. Координаты указателя мыши находятся в параметрах X и Y, аргумент State конкретизирует текущее состояние буксируемого объекта (dsDragEnter, dsDragLeave, dsDragMove). Наиболее важный параметр события — переменная Accept. Передав в нее значение true, мы выразим согласие о приеме объекта в док.

Событие

```
property OnDockDrop: TDockDropEvent;
type TDockDropEvent = procedure(Sender: TObject;
                      Source: TDragDockObject;  X, Y: Integer) of object;
```

возникнет в тот момент, когда пользователь отпустит кнопку мыши и бросит буксируемый объект на поверхность дока. В этот момент можно позаботиться о размещении объекта.

Перед расставанием с объектом док-отправитель генерирует событие

```
property OnUnDock: TUnDockEvent;
type TUnDockEvent = procedure(Sender: TObject; Client: TControl;
                  NewTarget: TWinControl; var Allow: Boolean) of object;
```

В нем док проверяет: в надежные ли руки попадает его бывший клиент (параметр `Client`). Ссылка на будущего владельца находится в параметре `NewTarget`. Если старый владелец удовлетворен новым хозяином, то присвойте переменной `Allow` значение `false`, а если нет — то `true` (листинг 13.6).

Листинг 13.6. Предоставление разрешения на уход из дока

```
procedure TForm1.CoolBar1UnDock(Sender: TObject; Client: TControl;
                      NewTarget: TWinControl; var Allow: Boolean);
begin
  Allow:=(NewTarget is TControlBar);
end;
```

Обработчик события запрещает передавать клиента другому доку, если тот принадлежит классу `TControlBar`. В результате операция буксировки прерывается.

Честь завершить процесс буксировки предоставляется переносимому элементу:

```
property OnEndDock: TEndDragEvent;
type TEndDragEvent = procedure(Sender, Target: TObject;
                               X, Y: Integer) of object;
```

Здесь: `Target` — док-получатель; `X` и `Y` — экранные координаты нового места.

Буксировка программным способом

В Delphi предусмотрены методы, реализующие технологию буксировки и программно. На уровне класса `TControl` объявлен метод, позволяющий принудительно отправить наш элемент управления в док.

```
procedure Dock(NewDockSite: TWinControl; ARect: TRect); dynamic;
```

Здесь `NewDockSite` — ссылка на новый док, параметр `ARect` конкретизирует координаты прямоугольной области, в которой должен разместиться объект.

Более совершенной версией метода `Dock()` считается функция:

```
function ManualDock(NewDockSite: TWinControl;
   DropControl: TControl = nil; ControlSide: TAlign = alNone): Boolean;
```

Параметр `DropControl` — необязательный, он может использоваться, когда на поверхности дока имеются дополнительные объекты (например, страницы `TTabSheet` на блокноте `TPageControl`). Параметр `ControlSide` определяет метод выравнивания на посадочной площадке. В случае успешного завершения метод возвратит `true`.

Допустим и перенос элемента управления не в док, а в любое место рабочего стола. Для этого предназначен метод

```
function ManualFloat(ScreenPos: TRect): Boolean;
```

Функция переместит элемент управления в область рабочего стола, заданную в параметре ScreenPos. Пример перемещения панели инструментов ToolBar1 в левый верхний угол рабочего стола компьютера демонстрирует листинг 13.7.

Листинг 13.7. Перемещение объекта в произвольную область рабочего стола

```
var ScreenPos:TRect;
begin
  {Перемещаемся в левый верхний угол рабочего стола}
  ScreenPos.TopLeft:=Point(0,0);
  ScreenPos.BottomRight:=ToolBar1.ClientRect.BottomRight;
  ToolBar1.ManualFloat(ScreenPos);
end;
```

ГЛАВА 14

Приложение VCL

Специалисты Embarcadero приложили максимум усилий, чтобы сделать язык Delphi максимально дружелюбным и свести к минимуму затраты времени на создание программного продукта. Как следствие, Delphi значительно упрощает процесс программирования, самостоятельно реализуя ключевые этапы организации взаимодействия приложения с операционной системой. Благодаря этому программист Delphi может сразу приступать к решению своей практической задачи и не тратить время на осуществление стандартных операций по организации работы приложения в Windows.

Однако, как это иногда бывает, благие намерения приводят и к нежелательному результату. Особо не афишируя тонкости внутреннего устройства приложения, написанного в Delphi, специалисты Embarcadero стимулировали появление целого поколения "знатоков", даже не подозревающих о том, как функционирует приложение для Windows. Поэтому одной из целей этой главы станет обсуждение скрытого от взоров начинающих разработчиков механизма взаимодействия приложения VCL Forms и Windows.

Щелчок по пункту меню **File | New | VCL Forms Application** создает заготовку приложения Delphi на основе форм, другими словами, приложения с графическим интерфейсом пользователя (GUI, graphical user interface). Надо сказать, что слово "заготовка" не в полной мере раскрывает вклад Delphi в проект. На самом деле в "заготовке" имеется все необходимое для того, чтобы после компиляции мы получили полноценное приложение для Windows. Правда, в нашем распоряжении окажется лишь единственное окно, без каких-либо элементов управления. Но надо понимать: чтобы написать аналогичное приложение без использования библиотеки VCL, нам понадобятся дополнительное время и знания особенностей взаимодействия приложения и операционной системы. Новичку такие подвиги окажутся непосильными.

> ***ЗАМЕЧАНИЕ***
>
> Среди примеров к главе книги вы найдете исходный код простейшего приложения для Windows без опоры на VCL. Для реализации такого приложения понадобилось около 80 строк кода.

Любое, даже самое совершенное приложение далеко не самодостаточно — для его работы нужна помощь операционной системы. Поэтому на всем протяжении своего функционирования приложение (в первую очередь его главное окно) поддерживает контакт с Windows (рис. 14.1). Основной способ ведения диалога между Windows и приложением заключается в отправке друг другу *сообщений* (messages). Сообщения содержат самый разнообразный набор команд, управляющих приложением, и сведений о состоянии приложения и его окружения.

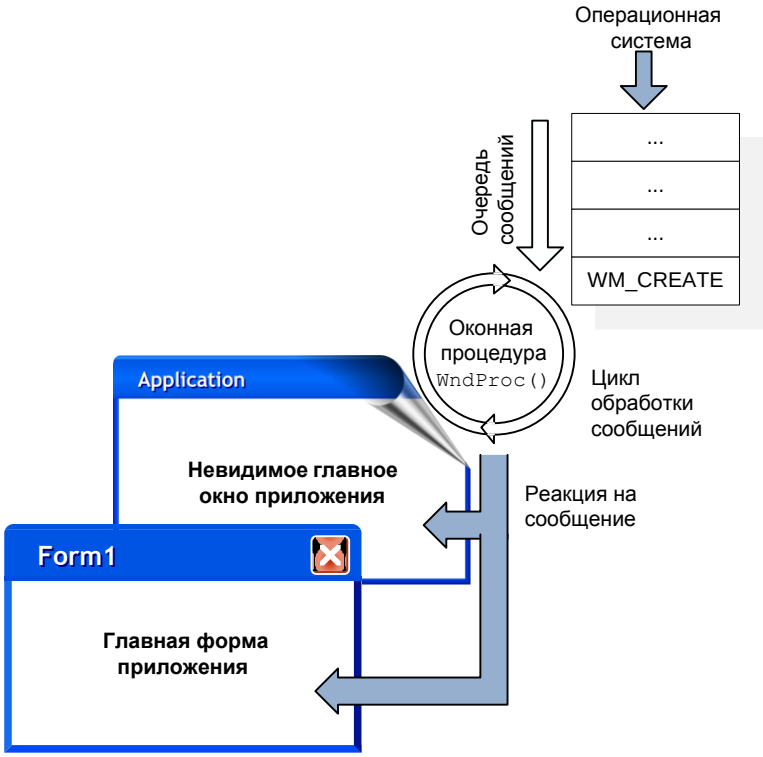


Рис. 14.1. Место объекта `Application` в приложении на основе VCL

Для того чтобы приложение смогло получить сообщение от операционной системы, оно создает специальную область памяти, называемую *очередью сообщений* (message queue). Очередь сообщений способна содержать несколько сообщений, для этого она устроена по принципу FIFO (First Input First Output), т. е. первое поступившее сообщение будет обслужено первым. Задача приложения — извлекать сообщения из своей очереди и осуществлять их обработку. Блок команд, извлекающих сообщения из очереди, называется *циклом обработки сообщений* (message loop). И если по какой-то причине у приложения перестанет выполняться цикл обработки сообщений, то оно безнадежно "зависнет".

В момент поступления в адрес приложения очередного сообщения цикл обработки сообщений вызывает *оконную процедуру*. В этой процедуре описывается реакция главного окна приложения на то или иное сообщение.

Созданное в Delphi приложение **VCL Forms Application** избавляет программиста от необходимости соблюдать "формальности" по организации взаимодействия между Windows и программным продуктом. Все необходимое уже сделано силами двух классов: `TApplication` (класс-приложение) и `TForm` (класс-форма). Рисунок 14.1 дает представление о порядке получения приложением VCL сообщений от операционной системы. Обратите внимание, что весь процесс обработки сообщений сосредоточен в границах объекта `Application`, инкапсулирующего главное невидимое окно приложения Delphi.

Приложение VCL, класс *TApplication*

Многие начинающие программисты даже не подозревают, что создаваемая по умолчанию главная форма проекта (форма `Form1`), строго говоря, не является главным окном приложения. Вместе с главной формой создается еще одно невидимое окно, на которое возлагаются задачи организации взаимодействия приложения и операционной системы. Именно это скрытое от глаз непосвященных окно и является главным окном приложения, или просто *приложением* Delphi.

В лучших традициях ООП создатели языка Delphi реализовали объект-приложение в виде класса `TApplication`. Класс объявлен в модуле `Forms` и является прямым потомком изученного в *главе 9* класса `TComponent`.

Не стоит искать `TApplication` среди компонентов Delphi, на палитре его нет. В момент создания проекта автоматически создается и объект приложения, доступ к которому осуществляется при посредничестве глобальной переменной

```
var Application: TApplication;
```

У приложения VCL много забот, экземпляр класса `TApplication` несет ответственность за:

- подготовку и создание главного окна приложения;
- старт приложения;
- перехват сообщений Windows;
- контроль за нажатием быстрых клавиш, соответствующих пунктам меню;
- вывод контекстной подсказки;
- обработку ошибок приложения.

Ко всему прочему `TApplication` спасает программиста от многих формальностей программирования на прикладном интерфейсе Windows API, связанных с объявлением и регистрацией оконных классов, описанием оконных процедур, циклов обработки сообщений и многого другого.

Оконная процедура

Если вы немного "пороетесь" в исходном коде модуля `Vcl.Forms`, в котором объявляется класс приложения `TApplication`, то вы обнаружите сердце любой программы — оконную процедуру `WndProc()`.

В адрес оконной процедуры поступают сообщения от операционной системы, других протекающих в ОС процессов и от самого приложения. Поступающие сообщения представлены в формате записи `TMessage`. Это структура с вариантным полем (листинг 14.1), содержащая код сообщения `Msg` и сопутствующие данные.

Листинг 14.1. Объявление записи `TMessage`

```
type TMessage = packed record
  Msg: Cardinal;
  case Integer of
    0: (WParam: WPARAM;
        LParam: LPARAM;
        Result: LRESULT);
```

```
      1: (WParamLo: Word;
          WParamHi: Word;
          LParamLo: Word;
          LParamHi: Word;
          ResultLo: Word;
          ResultHi: Word);
    end;
```

Еще раз подчеркну, что основная смысловая нагрузка структуры заключена в поле `Msg`. В него помещается именованная константа, описывающая характер сообщения. Например, в момент создания окна приходит сообщение `WM_CREATE`; при изменении его размеров — `WM_SIZE`; при получении фокуса ввода — `WM_SETFOCUS` и т. д. В Windows объявлены сотни сообщений, наиболее важные из них сосредоточены в модуле `Winapi.Messages`.

> ***Замечание***
>
> Аббревиатура WM расшифровывается как *оконное сообщение* (window message). В адрес окна могут приходить и другие сообщения, например: BM (button message); LB (listbox message); CB (combo box message); EM (edit control message) и т. п.

Назначение остальных параметров записи `TMessage` зависит от характера сообщения, например, в них может находиться дескриптор окна, в адрес которого отправлено сообщение или указатель на определенные данные.

Из-за большого размера оконной процедуры (около 200 строк) не стоит приводить в книге весь код, поэтому ограничимся существенно сокращенной версией листинга (листинг 14.2).

Листинг 14.2. Фрагмент оконной процедуры `WndProc()`

```
procedure TApplication.WndProc(var Message: TMessage);
var  I: Integer;
  //...
  procedure Default; //процедура обработки данных по умолчанию
  begin
    with Message do
      Result := DefWindowProc(Handle, Msg, WParam, LParam);
  end;

begin
  try
    Message.Result := 0;
    for I := 0 to FWindowHooks.Count - 1 do
      if TWindowHook(FWindowHooks[I]) Message) then Exit;
    CheckIniChange(Message);
    with Message do
      case Msg of
        WM_SIZE: //сообщение изменения размера
          if WParam = SIZE_MINIMIZED then
            FAppIconic := True;
        WM_CLOSE: //сообщение закрытия главного окна
          if MainForm <> nil then MainForm.Close;
        //...
```

```
        WM_SETFOCUS: //сообщение получения фокуса
          begin
            PostMessage(Handle, CM_ENTER, 0, 0);
            Default;
          end;
        //...
        WM_NULL: //нет сообщения
          CheckSynchronize;
      else
        Default; {если сообщение не обработано, передаем его
                  процедуре обработки данных по умолчанию}
      end;
  except
    HandleException(Self); //исключительная ситуация
  end;
end;
```

Основная задача оконной процедуры приложения — описать реакцию приложения и главного окна приложения на поступающие в их адрес ключевые сообщения Windows. Обработка сообщения заключается в рамках оператора-селектора **case** `Msg` **of**. Например, в момент появления сообщения `WM_CLOSE` мы указываем главной форме проекта о том, что она должна быть закрыта.

Вполне понятно, что разработчики класса `TApplication` не могут предусмотреть все возможные варианты жизненного цикла приложения. Поэтому в оконной процедуре `WndProc()` обрабатывается около трех десятков наиболее важных сообщений. Все остальные (не прошедшие явную обработку) сообщения передаются в адрес оконной процедуры по умолчанию `DefWindowProc()` — с ними самостоятельно разберется Windows.

Общие возможности *Application*

Объектно-ориентированный язык Delphi не афиширует существенную часть возможностей объекта `Application`. Это объясняется не столько тягой к конспирации, сколько тем, что вмешательство в функционал приложения со стороны неопытного программиста для проекта может оказаться фатальным. Вместе с тем даже начинающему изучать Delphi разрешается кое-что потрогать руками. Например, на стадии визуального проектирования разрешается назначить приложению новый значок, придумать отображаемый на панели задач заголовок и подключить файл справки. Для этого не стоит сочинять ни одной строки кода (хотя, безусловно, предусмотрен и такой способ). В простейшем случае достаточно выбрать в меню Delphi пункт **Project | Options...** или нажать комбинацию клавиш <Shift>+<Ctrl>+<F11>. В результате на экране отобразится окно настройки проекта, в котором за доступ к приложению отвечает узел **Application** (рис. 14.2 и 14.5).

Эти же свойства доступны и во время выполнения проекта. Заголовок свернутого приложения, лежащего на панели задач, определяется свойством

```
property Title : string;
```

Значок приложения назначается в свойстве

```
property Icon: TIcon;
```

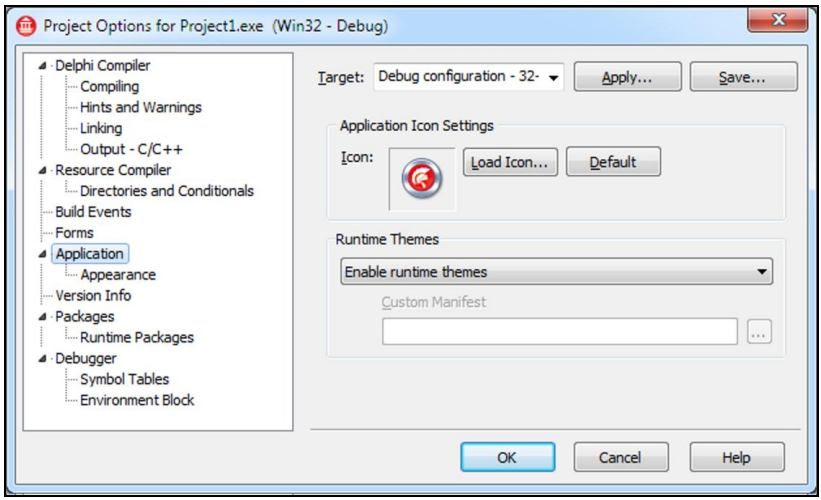


Рис. 14.2. Узел **Application** в опциях проекта Delphi XE2

Если приложение активно, то свойство

```
property Active: Boolean; //только для чтения
```

вернет значение `true`.

Очень хорошую службу способно сослужить свойство

```
property ExeName : string; //только для чтения
```

В нем хранится имя исполняемого файла приложения и путь к нему. Это весьма полезное свойство, которое позволит нам во время выполнения приложения идентифицировать место расположения нашего exe-файла. Листинг 14.3 демонстрирует порядок применения свойства в приложении, нуждающемся в протоколировании его работы.

Листинг 14.3. Протоколирование даты и времени запуска приложения

```
var FileName : string;
    F : TextFile;
begin
  FileName:=ExtractFilePath(Application.ExeName)+'work.log';
  AssignFile(F,FileName);
  try
    if FileExists(FileName)=false then {проверка наличия файла FileName}
      Rewrite(F)  {файла с именем FileName нет — создаем}
    else
       Append(F); {файл с именем FileName есть — добавляем новую строку}
       WriteLn(F, FormatDateTime('Запуск dd.mm.yy в hh:mm:ss', Now));
  finally
    CloseFile(F);
  end;
end;
```

Представленный в листинге 14.3 код может быть вызван в момент старта приложения (например, в момент создания главной формы проекта). В результате той же папке, в которой и

расположен исполняемый файл, будет создан файл work.log, в котором сохранятся дата и время каждого запуска приложения.

Доступ к основным объектам приложения

Приложение обладает информацией, какая из его форм исполняет обязанности главной формы проекта. Ссылка на экземпляр главной формы находится в свойстве

```
property MainForm: TForm;
```

Для обеспечения взаимодействия с функциями API Windows требуется знать дескрипторы основных объектов приложения. Дескриптор главного окна приложения хранится в свойстве

```
property Handle: HWND; //только для чтения
```

Дескриптор активной формы возвратит свойство

```
property ActiveFormHandle: HWND; //только для чтения
```

Дескриптор главной формы находится в свойстве

```
property MainFormHandle: HWND;   //только для чтения
```

В момент получения дескрипторов у приложения возникают события

```
property OnGetActiveFormHandle: TGetHandleEvent; //для активной формы
property OnGetMainFormHandle: TGetHandleEvent;   //для главной формы
```

> ***Внимание!***
>
> Не путайте понятия "главное окно приложения" и "главная форма проекта"! В проектах VCL Forms главное окно скрыто и принадлежит объекту `TApplication`.

Обработка поступающих сообщений

Создаваемые с помощью библиотеки VCL приложения малой и средней степени сложности самостоятельно обрабатывают поступающие в их адрес сообщения и не нуждаются во внешнем вмешательстве в этот весьма ответственный процесс. Однако если программист нацелен на создание профессионального программного обеспечения, то он должен научиться работать с приложением на уровне сообщений Windows.

Объект `Application` перехватывает все поступающие приложению сообщения, генерируя при этом событие

```
property OnMessage: TMessageEvent;
TMessageEvent = procedure(var Msg: tagMSG;
                          var Handled: Boolean) of object;
```

Первый параметр события `Msg` (листинг 14.4) описывает входящее сообщение. Второй параметр `Handled` позволит нам запрещать приложению реагировать на сообщение, для этого достаточно передать в него значение `true`.

Листинг 14.4. Состав полей записи `tagMSG`

```
type tagMSG = packed record
    hwnd: HWND;     //дескриптор получателя сообщения
    message: UINT;  //идентификатор сообщения
```

```
    wParam: WPARAM; //старший параметр
    lParam: LPARAM; //младший параметр
    time: DWORD;    //время
    pt: TPoint;     //координаты указателя мыши
  end;
```

Для того чтобы воспользоваться услугами события `OnMessage()`, можно пойти двумя путями: можно добавить к проекту компонент `TApplicationEvents` (об этом мы поговорим в конце главы) или объявить соответствующий процедурный тип в коде главной формы проекта и уполномочить его обрабатывать поступающие в приложение сообщения. Подобное решение предложено в листинге 14.5.

Листинг 14.5. Просмотр поступающих сообщений

```
type
  TForm1 = class(TForm)
  Memo1:TMemo;
  //...
  procedure FormCreate(Sender: TObject);
  private
    procedure EventsMessage(var Msg: tagMSG;  var Handled: Boolean);
  public
//...

procedure TForm1.EventsMessage(var Msg: tagMSG; var Handled: Boolean);
begin
  //регистрируем дескриптор окна-получателя и код сообщения
  Memo1.Lines.Add(
    Format('Получатель $%.8x Сообщение $%.4x ',[Msg.hwnd, Msg.message]));
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage:=EventsMessage;
  //обработка события осуществляется процедурой EventsMessage()
end;
```

Среди примеров к книге вы найдете исходный код приложения, анализирующего все поступающие в его адрес сообщения (рис. 14.3). Поэкспериментировав с примером, вы увидите, что любые операции с приложением (движение мышью, нажатие клавиш и т. п.) стимулируют появление целого каскада сообщений.

> ***Замечание***
>
> Еще со времен первых версий Delphi в состав программного пакета входит утилита WinSight, позволяющая отслеживать сообщения, поступающие в адрес практически любого выполняющегося в данный момент приложения. Файл утилиты ws32.exe находится в папке bin.

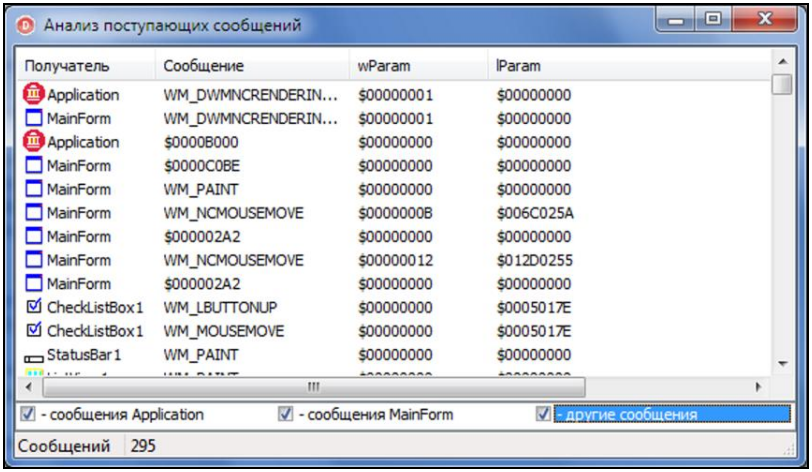


Рис. 14.3. Экранный снимок приложения, регистрирующего входящие сообщения

Управление процессом создания приложения

При запуске исполняемого файла приложения объект `Application` проходит собственную инициализацию и приступает к созданию форм приложения. В этот процесс можно внести коррективы, для этого следует исправить головной модуль проекта Delphi (dpr-файл). По умолчанию головной модуль отсутствует в списке программных модулей проекта, однако это не сложно исправить, обратившись к пункту меню **Project | View Source**. В результате в редакторе откроется самый главный листинг проекта (листинг 14.6).

Листинг 14.6. Головной модуль стандартного проекта VCL

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.res} //подключение файла ресурса

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  {...
   Создание других автоматически создаваемых форм проекта
   ...}
  Application.Run;
end.
```

Для инициализации приложения вызывается метод

```
procedure Initalize;
```

Для того чтобы на панели задач Windows появлялась ссылка на выполняющееся приложение, в состояние `true` устанавливается свойство

```
property MainFormOnTaskBar: Boolean;
```

Это свойство приобрело еще большую значимость с выходом в свет Windows Vista и Windows 7. В состоянии `true` оно указывает системе, что приложение должно быть включено в список эффектов Windows Aero.

Затем наступает черед создания форм приложения. Для этого вызывается метод

```
procedure CreateForm(FormClass: TFormClass; var Reference);
```

Первой всегда создается главная форма приложения. Метод конструирует форму с типом, указанным в параметре `FormClass`. Параметр `Reference` указывает на экземпляр формы, например `Form1`. Метод очень похож на обычный конструктор `Create()`, за исключением того, что `CreateForm()` всегда проверит, не равно ли свойство `Application.MainForm` значению `nil`.

> ***ЗАМЕЧАНИЕ***
>
> При старте приложения создаются только формы, включенные в состав автоматически создаваемых (список **Auto-create forms** на странице **Forms** опций проекта **Project Options**). Последовательность создания этих форм соответствует их очередности в этом списке.

При запуске приложения допустимо отказаться от показа главной формы. Для этого установите в `false` свойство

```
property ShowMainForm: Boolean;
```

Завершив все приготовления, объект `Application` воспользуется методом

```
procedure Run;
```

Это и есть запуск приложения. Для программиста, желающего работать с Delphi на профессиональном уровне, метод представляет огромный интерес (листинг 14.7).

Листинг 14.7. Реализация метода `Run()` приложения

```
procedure TApplication.Run;
begin
  FRunning := True;
  try
    AddExitProc(DoneApplication); //процедура выхода
    if FMainForm <> nil then      //если главная форма создана
    begin
     //особенности вывода на экран главной формы
      case CmdShow of
        SW_SHOWMINNOACTIVE:
          begin
            FInitialMainFormState := wsMinimized;
            FMainForm.FWindowState := wsMinimized;
          end;
        SW_SHOWMAXIMIZED: MainForm.WindowState := wsMaximized;
      end;
```

```
      if FShowMainForm then
        if (FMainForm.FWindowState = wsMinimized) or
           (FInitialMainFormState = wsMinimized) then
        begin
          Minimize;
          if (FInitialMainFormState = wsMinimized) then
            FMainForm.Show;
        end else
          FMainForm.Visible:= True;
      //САМЫЙ ГЛАВНЫЙ ЭЛЕМЕНТ ПРИЛОЖЕНИЯ — ЦИКЛ ОБРАБОТКИ СООБЩЕНИЙ
      repeat
        try
          HandleMessage;
        except
          HandleException(Self);
        end;
      until Terminated;

    end;
  finally
    FRunning := False;
  end;
end;
```

Поле `FRunning` представляет собой глобальный флаг приложения, свидетельствующий о том, запущено приложение (`true`) или нет (`false`). В случае возникновения исключительной ситуации (секция обработки ошибок `try..finally..end`) флаг очищается. После проверки на существование главной формы проекта `FMainForm<>nil` метод `Run()` определит, в каком виде показывать приложение (свернутым или развернутым на весь экран). Для этого в операторе-селекторе `case` проверяется состояние переменной `CmdShow` и передается соответствующее значение в поле `FWindowState` главной формы.

Самый важный программный элемент приложения заключен в цикл `repeat..until`, это не что иное, как организация цикла обработки сообщений (см. рис. 14.1). В рамках цикла приложение постоянно опрашивает метод

```
procedure HandleMessage;
```

Метод инициирует сложный процесс выборки и обработки одиночного сообщения из очереди сообщений Windows. Если очередь сообщений пуста, `HandleMessage()` генерирует событие `OnIdle()`, которое служит свидетельством, что приложение простаивает.

> ***ЗАМЕЧАНИЕ***
>
> Приложение продолжает свою работу до тех пор, пока в его адрес не поступит сообщение `WM_QUIT`.

Завершение работы приложения

Штатным способом завершения работы приложения является закрытие его главной формы. Для этого обычно вызывается метод `Close()` главной формы, все остальные необходимые действия осуществляются автоматически.

Однако существуют и другие способы прекращения выполнения приложения, например, в распоряжении TApplication имеется метод

```
procedure Terminate;
```

Метод вызывает функцию Windows API PostQuitMessage(), приложение освобождает захваченные ресурсы и прекращает работу.

> ***ЗАМЕЧАНИЕ***
>
> В экстренных случаях для немедленного останова приложения допустимо воспользоваться процедурой Halt(). Однако помните, что в этом случае не гарантируется корректное освобождение ресурсов.

Сворачивание и восстановление размеров окна приложения

Для сворачивания окна приложения предназначен метод

```
procedure Minimize;
```

Восстановление исходных размеров окна выполняется с помощью метода

```
procedure Restore;
```

Обращение к перечисленным методам заканчивается генерацией событий

```
property OnMinimize : TNotifyEvent; //сворачивание окна
property OnRestore : TNotifyEvent;  //восстановление размеров окна
```

Диалоговое окно приложения

Зачастую та или иная операция, выполняемая приложением, нуждается в подтверждении пользователем. Для этой цели в состав методов класса TApplication инкапсулирована функция вызова диалогового окна

```
function MessageBox(const Text, Caption: PChar;
                    Flags: Longint = MB_OK): Integer;
```

Текстовые параметры метода отвечают за содержание сообщения и надпись в заголовке диалогового окна. Параметр Flags определяет перечень кнопок, выводимых на окне диалога. Диалог возвращает целое число — код нажатой кнопки.

> ***ЗАМЕЧАНИЕ***
>
> Более подробно о диалоговых окнах мы поговорим в *главе 28*, а пока отметим, что наибольшими возможностями по выводу диалогов обладает функция MessageBoxEx().

Метод MessageBox() используется при решении простейших задач, листинг 14.8 демонстрирует возможности функции при запросе подтверждения пользователя на завершение работы приложения. Для этого метод вызывается внутри обработчика события OnCloseQuery() главной формы проекта.

Листинг 14.8. Запрос на завершение работы приложения

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
```

```
  CanClose:=
    Application.MessageBox(pWideChar('Завершить работу приложения?'),
                           pWideChar('Подтвердите операцию'),
                           MB_YESNO)=idYes;
end;
```

Осуществление оперативной подсказки

Для того чтобы пользователь как можно быстрее освоил порядок работы с программным продуктом, его элементы управления снабжаются функцией отображения оперативной подсказки. Оперативная подсказка заносится в свойство `Hint` элемента управления, и при наведении указателя мыши на элемент управления над ним появляется строка с поясняющим текстом о назначении этого элемента.

> ***ЗАМЕЧАНИЕ***
>
> Свойство `Hint` одновременно может содержать два варианта подсказки. Варианты отделяются друг от друга символом вертикальной черты |. Для получения первой части подсказки можно воспользоваться функцией `GetShortHint()`, второй — `GetLongHint()`.

В момент перемещения указателя мыши над элементом управления, снабженного текстом оперативной подсказки, у экземпляра класса `TApplication` вызывается обработчик события

```
property OnHint: TNotifyEvent;
```

Кроме события `OnHint()`, в классе `TApplication` имеется ряд свойств, имеющих прямое отношение к организации оперативной подсказки (табл. 14.1).

Таблица 14.1. *Свойства `TApplication`, отвечающие за оперативную подсказку*

Свойство	Описание
`property Hint: string;`	В этом свойстве окажется показываемый в данный момент текст подсказки
`property HintColor: TColor;`	Цвет фона всплывающей подсказки
`property HintPause: Integer;`	Интервал времени в миллисекундах, проходящий с момента появления курсора мыши над компонентом до появления всплывающей подсказки. По умолчанию — 500 миллисекунд
`property HintHidePause: Integer;`	Время демонстрации подсказки, по умолчанию 2500 миллисекунд
`property HintShortPause: Integer;`	Время смены всплывающих подсказок при быстром перемещении курсора над компонентами, по умолчанию 50 миллисекунд
`property HintShortCuts: Boolean;`	Указывает, включать ли в подсказку информацию о сочетаниях быстрых клавиш

Если всплывающая подсказка видна слишком долго, для ее отмены следует обратиться к методу

```
Procedure CancelHint;
```

ЗАМЕЧАНИЕ

Кроме средств оперативной подсказки программисты Delphi часто используют возможности класса `THintWindow`, позволяющего выводить всплывающие окна с дополнительной информацией.

Подключение к справочной системе

Завершающим этапом работы над проектом обычно выступает разработка системы помощи и подключение ее к приложению.

Подключение справочного файла обычно осуществляется из вызываемого с помощью меню **Project | Options** диалога (см. рис. 14.2). Однако это не запрещено сделать и во время выполнения приложения. Для этого достаточно обратиться к свойству

```
property HelpFile: string;
```

Имя текущего файла помощи, сопоставленного с приложением, вы найдете в свойстве

```
property CurrentHelpFile: string; //только для чтения
```

Для взаимодействия со справочной системой предназначены методы

```
function HelpCommand(Command: Word; Data: Longint): Boolean;
function HelpContext(Context: THelpContext): Boolean;
function HelpJump(const JumpID: string): Boolean;
```

Метод `HelpCommand()` служит для вызова макросов, описанных в файле справки. Методы `HelpContext()` и `HelpJump()` выводят конкретную страницу справки.

Доступ к интерфейсу справочной системы обеспечит свойство

```
property HelpSystem: IHelpSystem;
```

Интерфейс `IHelpSystem` обладает достаточно серьезными возможностями по управлению справочной системой проекта. В табл. 14.2 представлена краткая аннотация функций интерфейса.

Таблица 14.2. *Основные методы интерфейса* `IHelpSystem`

Метод	Описание
`procedure ShowHelp(const HelpKeyword, HelpFileName: string);`	Покажет страницу подсказки из файла `HelpFileName` по ключевому слову `HelpKeyword`
`procedure ShowContextHelp(const ContextID: Longint; const HelpFileName: string);`	Покажет страницу подсказки из файла `HelpFileName` по индексу `ContextID`
`procedure ShowTableOfContents;`	Вывод таблицы содержимого справочного файла
`procedure ShowTopicHelp(const Topic, HelpFileName: string);`	Покажет тему `Topic` из файла `HelpFileName`

ЗАМЕЧАНИЕ

Если в качестве справочной системы приложение будет использовать WinHelp, то в строку `uses` проекта следует поместить ссылку на модуль `WinHelpViewer`.

Обращение приложения к файлу помощи можно проконтролировать в обработчике события приложения

```
property OnHelp: THelpEvent;
type THelpEvent = function(Command: Word; Data: Longint;
                           var CallHelp: Boolean): Boolean of object;
```

Автоматический вызов этого события происходит в случае обращения к функциям `HelpCommand()`, `HelpJump()` и `HelpContext()`.

События приложения, класс *TApplicationEvents*

Для упрощения организации доступа к событиям, на которые способно откликнуться приложение, в VCL включен специальный невизуальный компонент `TApplicationEvents` (страница палитры компонентов **Additional**). У компонента практически нет свойств (за исключением имени `Name` и целочисленного значения `Tag`). Однако он сумел инкапсулировать почти все события (табл. 14.3), связанные с жизнедеятельностью приложения.

***Таблица 14.3.** События приложения, класс `TApplicationEvents`*

Событие	Описание
`OnActionExecute()`	Событие возникает перед выполнением команды `TAction`
`OnActionUpdate()`	Обработчик события используется для централизованного перевода элементов управления в актуальное состояние
`OnActivate()`	Событие возникает при получении приложением фокуса ввода
`OnDeactivate()`	Событие возникает в момент утраты приложением фокуса ввода
`OnException()`	Приложение вызвало исключительную ситуацию
`OnHint()`	Показ оперативной подсказки
`OnIdle()`	Приложение перешло в фоновый режим
`OnMessage()`	Получение приложением сообщения от операционной системы
`OnMinimize()`	Приложение начинает сворачиваться
`OnModalBegin()`	Одна из форм приложения открывается в модальном режиме
`OnModalEnd()`	Модальная форма приложения закрывается
`OnRestore()`	Приложение восстанавливается до исходных размеров
`OnSettingChange()`	Возникает, когда Windows оповещает приложения об изменениях в глобальных настройках ОС
`OnShortCut()`	Событие возникает при нажатии комбинации быстрых клавиш (обработчик события вызывается до события `OnKeyDown`)
`OnShowHint()`	Вызывается в момент отображения всплывающей подсказки

Допустим, что мы намерены научить приложение выводить вторую часть оперативной подсказки, спрятанную в свойство `Hint` формы. Разместите на главной форме проекта компонент `TApplicationEvents` и статусную панель `TStatusBar`. Для реализации наших планов нам потребуются услуги пары событий. В событии `OnShow()` формы проекта (возникающем

в момент первого вывода формы на экран) мы решим вспомогательные задачи — настроим поведение формы и статусной строки. В событии `OnHint()` компонента `TApplicationEvents` мы отделим вторую часть подсказки и выведем ее на поверхности статусной панели (листинг 14.9).

Листинг 14.9. Пример использования события `OnHint()` для вывода подсказки

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Form1.Hint:='Первая часть подсказки|Вторая часть подсказки';
  Form1.ShowHint:=True;
  StatusBar1.SimplePanel:=True;
end;

procedure TForm1.ApplicationEvents1Hint(Sender: TObject);
begin
  StatusBar1.SimpleText := GetLongHint(Application.Hint);
end;
```

Значок в области уведомлений

Если создаваемое вами приложение относится к разряду программных продуктов, которые всегда должны быть под рукой, то имеет смысл научить приложение создавать значок в *области уведомлений* (system tray). В таком случае пользователю не понадобится перебирать открытые приложения на панели задач Windows, вместо этого следует найти значок в области уведомлений (рядом с системными часами).

Для быстрого доступа к области уведомлений в состав VCL Delphi введен невизуальный компонент `TTrayIcon`. Он очень прост в обращении — для того чтобы получить задуманный результат, программисту необходимо применить минимум усилий.

Во-первых, если для идентификации приложения в области уведомлений достаточно одного статического значка, то следует воспользоваться свойством

```
property Icon: TIcon;
```

Если предполагается, что значок может изменяться в зависимости от состояния приложения (факт прихода почты отображается одним значком, режим ожидания — другим), то все картинки передаются в коллекцию изображений (компонент `TImageList`), и коллекция подключается к компоненту при посредничестве свойства

```
property Icons: TCustomImageList;
```

Для смены значка достаточно указать его индекс в свойстве

```
property IconIndex: Integer; //по умолчанию 0
```

Наконец, если у нас имеется список значков, то их можно заставить сменять друг друга по циклу, активировав свойство

```
property Animate: Boolean; //по умолчанию false
```

Периодичность смены значков задает свойство

```
property AnimateInterval: Cardinal; //по умолчанию 1000 миллисекунд
```

Во-вторых, разобравшись со значком (или значками), переходим ко второму этапу "программирования" — включаем в состояние `true` свойство

```
property Visible: Boolean; //по умолчанию false
```

По большому счету задача-минимум уже решена, перевод `Visible` в состояние `true` укажет системе, что наше приложение надеется разместить свой значок в области уведомлений.

Для того чтобы пользователь смог воспользоваться функционалом приложения, свернутого в область уведомлений, проще всего воспользоваться контекстным меню (компонент `TPopupMenu`). Меню подключается к экземпляру `TTrayIcon` с помощью свойства

```
property PopupMenu: TPopupMenu;
```

Особенности работы с меню мы обсудим несколько позднее *(см. главу 19)*, а пока достаточно знать, что двойной щелчок по компоненту `TPopupMenu` вызывает встроенный редактор, который позволит пользователю определить пункты меню.

После подключения меню щелчок правой кнопкой мыши по значку приложения в области уведомлений выдаст на экран описанные программистом пункты меню и позволит пользователю выполнять основные действия с программой.

В табл. 14.4 представлены свойства и методы `TTrayIcon`, благодаря которым мы сможем научить даже невидимое (свернутое в область уведомлений) приложение отображать на экране окно уведомления с произвольным текстовым сообщением.

Таблица 14.4. *Вызов информационного окна в области системных уведомлений*

Свойство	Описание
`property BalloonTitle: string;`	Заголовок всплывающего окна
`property BalloonHint: string;`	Текстовое сообщение в области окна

`property BalloonFlags: TBalloonFlags; //по умолчанию bfNone;`
`type TBalloonFlags = (bfNone = NIIF_NONE, bfInfo = NIIF_INFO, bfWarning = NIIF_WARNING, bfError = NIIF_ERROR);`
	Пиктограмма окна

`property BalloonTimeout: Integer; //по умолчанию 1000 миллисекунд`	Продолжительность показа окна на экране, по умолчанию 1000 миллисекунд
`procedure ShowBalloonHint;`	Процедура, выводящая окно на экран

Перечень событий, на которые способен реагировать компонент `TTrayIcon`, представлен в табл. 14.5.

Таблица 14.5. *Обработка событий в компоненте* `TTrayIcon`

Свойство	Описание
`property OnBalloonClick: TNotifyEvent;`	Щелчок по окну уведомлений
`property OnClick: TNotifyEvent;`	Одинарный щелчок по значку в области уведомлений
`property OnDblClick: TNotifyEvent;`	Двойной щелчок по значку в области уведомлений
`property OnMouseMove: TMouseMoveEvent;`	Перемещение указателя мыши над значком
`property OnMouseUp: TMouseEvent;`	Отпускание кнопки мыши над значком

Таблица 14.5 *(окончание)*

Свойство	Описание
property OnMouseDown: TMouseEvent;	Опускание кнопки мыши над значком
property OnAnimate: TNotifyEvent;	Событие сопровождает анимированную смену значков в области уведомлений

Пример работы с компонентом *TTrayIcon*

Как уже утверждалось в начале подраздела, работа с компонентом TTrayIcon предельна проста. Проверим это на практике. Для этого создайте новый проект **VCL Forms Application** (я его назвал SysTrayDemo) и разместите на главной форме проекта два компонента:

- контекстное меню PopupMenu1:TPopupMenu;
- компонент TrayIcon1:TTrayIcon.

Воспользовавшись свойством Icon компонента TrayIcon1, подключите к приложению значок, который вы рассчитываете разместить в области системных уведомлений. Затем с помощью PopupMenu присоедините к компоненту контекстное меню PopupMenu1. Установите свойство Visible компонента TrayIcon1 в состояние true, иначе мы не увидим значок в области уведомлений.

Откройте встроенный редактор меню (дважды щелкнув по компоненту PopupMenu1) и создайте там два элемента. Элемент меню с именем nShow станет отвечать за вывод главной формы проекта на рабочий стол (Caption:='Открыть приложение'). Элемент с именем nClose предназначен для завершения работы приложения (Caption:='Завершить работу').

Теперь нам предстоит написать несколько строк кода. Сначала внесем незначительные изменения в головной модуль нашего проекта. Для этого воспользуйтесь пунктом меню среды проектирования **Project | View Source**, запретите приложению отображаться на панели задач и отключите показ главной формы проекта в момент создания приложения (листинг 14.10).

Листинг 14.10. Изменения в головном модуле проекта

```
program SysTrayDemo;
uses  Forms, Unit1 in 'Unit1.pas' {Form1};
{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar:= false;{отказ от вывода на панели задач}
  Application.ShowMainForm:=false; {отказ от отображения главной формы}
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Вернитесь к программному модулю главной формы. Нам предстоит описать три события: событие создания главной формы проекта Form1 и события, возникающие в момент щелчка по пунктам nShow и nClose контекстного меню PopupMenu1 (листинг 14.11).

Листинг 14.11. Обработка событий в приложении

```
procedure TForm1.FormCreate(Sender: TObject); {создание формы}
begin
  TrayIcon1.BalloonFlags:=bfInfo;
  TrayIcon1.BalloonHint:='Приложение запущено!';
  TrayIcon1.ShowBalloonHint;
end;

procedure TForm1.nShowClick(Sender: TObject); {щелчок по пункту меню}
begin
  Application.MainForm.Show; //вывод формы на экран
end;

procedure TForm1.nCloseClick(Sender: TObject); {щелчок по пункту меню}
begin
  Application.MainForm.Close; //прекращаем работу приложения
end;
```

В момент старта приложения в области уведомлений появится значок проекта сообщения, а над ним ненадолго возникнет информационное окно с надписью "Приложение запущено!". Щелчок правой кнопкой мыши по значку вызовет всплывающее меню (рис. 14.4), благодаря которому мы сможем открыть главное окно проекта или завершить работу приложения.

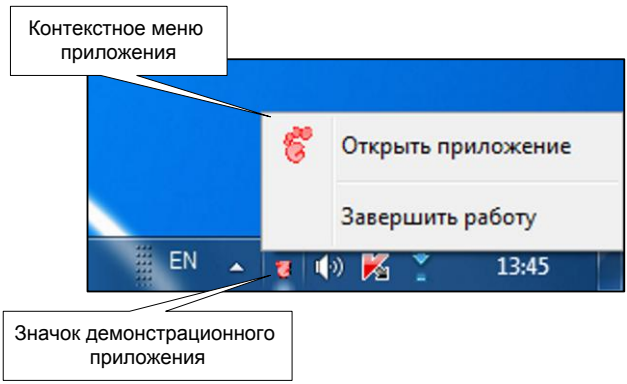


Рис. 14.4. Область уведомлений Windows со значком разработанного приложения

Стили оформления приложения

Вместе с выходом в свет Delphi XE2 у разработчиков приложений на платформе VCL появилась уникальная возможность управления внешним видом приложения с помощью заранее подготовленных стилей оформления. Стиль оформления (на сленге программистов "шкурка") приложения представляет собой набор единых правил, определяющих порядок отображения всех элементов управления и форм приложения. У приложения может быть неограниченное число стилей, которые могут выбираться пользователем в соответствии с его предпочтениями.

Замечание

Вместе с программным пакетом Embarcadero RAD Studio XE2 поставляется несколько vsf-файлов с предопределенными стилями оформления. Эти файлы расположены в папке C:\Users\Public\Documents\RAD Studio\9.0\Styles.

Для подключения стиля к приложению следует обратиться к окну настройки опций проекта (пункт меню **Project | Options**). В дереве опций выбираем узел **Application | Appearance**. В результате программист получает доступ к списку предопределенных стилей **Custom Styles**. Нам осталось отметить "галочками" предпочтительные стили и выбрать стиль по умолчанию в раскрывающемся списке **Default style**. Рекомендую по умолчанию всегда использовать классический стиль Windows, однако во время экспериментов подойдет и любой другой из предлагаемых Embarcadero стилей (рис. 14.5).

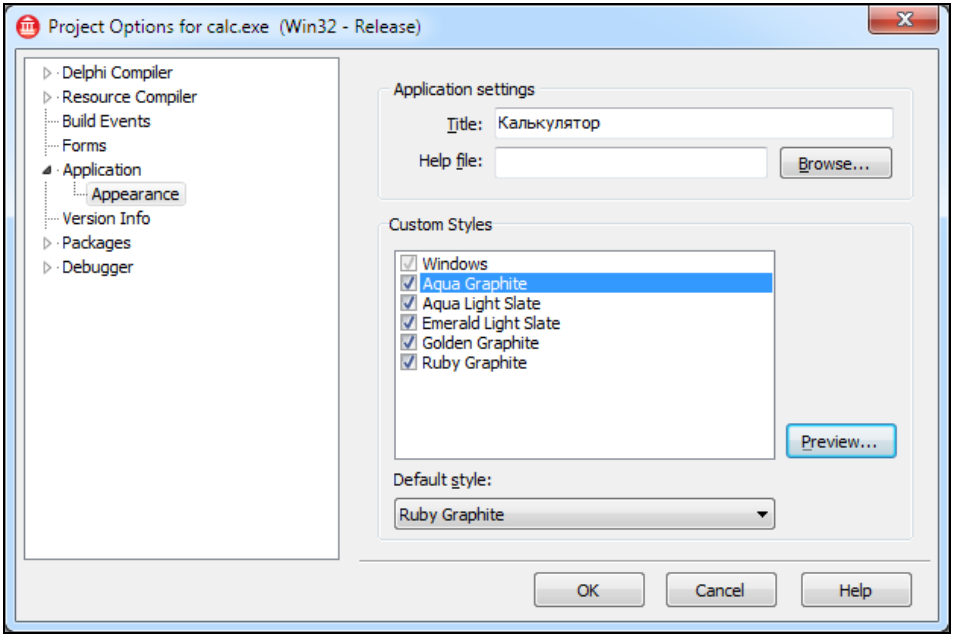


Рис. 14.5. Подключение стилей к проекту

С точки зрения программирования установка в качестве стиля по умолчанию любого стиля, отличного от "Windows", приводит к появлению в головном модуле проекта (файл с расширением dpr) дополнительной строки кода (листинг 14.12).

Листинг 14.12. Подключение стиля по умолчанию в головном модуле проекта

```
program Project1;
uses Vcl.Forms,  Unit1 in 'Unit1.pas' {Form1}, Vcl.Themes,  Vcl.Styles;
{$R *.res}
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  TStyleManager.TrySetStyle('Ruby Graphite'); //подключили стиль
```

```
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Менеджер стилей *TStyleManager*

В проектах VCL основная роль по управлению стилями оформления приложения отведена менеджеру стилей `TStyleManager`. Менеджер объявлен в модуле `Vcl.Themes` и выступает прямым потомком базового класса `TObject`. В сферу деятельности менеджера входит решение следующих задач:

- загрузка описания стиля из файла или ресурса;
- автоматическая регистрация стиля в приложении и предоставление доступа к зарегистрированным стилям;
- проверка возможности подключения стиля к приложению;
- собственно применение оформительского стиля к приложению VCL.

Нам уже известно, что стили оформления могут быть подключены еще на этапе визуального проектирования приложения. Но это не единственный способ управления стилями. Ко всему прочему менеджер способен получать стили из файлов (такие файлы идентифицируются расширением vsf) и из библиотеки ресурсов во время выполнения приложения. В первом случае для загрузки стиля из файла `FileName` следует воспользоваться методом

```
class function LoadFromFile(const FileName: string): TStyleServicesHandle;
```

Во втором случае загрузка должна производиться при посредничестве метода

```
class function LoadFromResource(Instance: HINST;
       const ResourceName: string): TStyleServicesHandle; overload;
```

Здесь первый параметр хранит указатель на загруженный в память объект, содержащий ресурс, второй параметр конкретизирует название ресурса.

Если логика разрабатываемого нами приложения допускает чтение описания стиля из файла, то перед началом загрузки (метод `LoadFromFile`) следует посоветоваться с методом

```
class function IsValidStyle(const FileName: string;
                            var StyleInfo: TStyleInfo): Boolean;
```

Кроме собственно уведомления программиста о корректности (значение `true`) или некорректности файла `FileName` метод получит описание стиля и передаст его в параметр `StyleInfo`. Описание представляет собой структуру, содержащую пять строковых полей (листинг 14.13).

Листинг 14.13. Описание записи `TStyleInfo`

```
type TStyleInfo = record
    Name: string;         //название стиля
    Author: string;       //имя автора
    AuthorEMail: string; //почтовый адрес
    AuthorURL: string;    //адрес URL
    Version: string;      //номер версии
  end;
```

Сам факт загрузки стиля не предполагает немедленной замены старой "шкурки" нашего приложения новой. В результате загрузки мы просто получаем дескриптор области памяти, в которой располагаются данные с потенциальным стилем оформления. Для того чтобы внешний вид приложения изменился, нам следует зарегистрировать стиль.

Регистрацией стиля оформления, полученного из файла (или ресурса), занимается перегружаемый метод

```
class procedure SetStyle(Handle: TStyleServicesHandle); overload;
```

Если регистрация осуществляется успешно, то метод применяет стиль и приложение приобретает новый внешний вид.

> ***ВНИМАНИЕ!***
>
> Если вы попытаетесь повторно применить к приложению уже зарегистрированный ранее оформительский стиль — ждите появления исключительной ситуации.

Успешно зарегистрированный стиль окажется в свойстве

```
class property StyleNames: TArray<string>;
```

Как минимум в массиве стилей мы обнаружим стиль "Windows", содержащий описание текущего стиля оформления.

У метода `SetStyle()` имеются еще два одноименных "коллеги". Но в отличие от своего собрата они не способны регистрировать новый стиль оформления и специализируются лишь на переключении внешнего вида приложения к одной из зарегистрированных ранее "шкурок":

```
class procedure SetStyle(const Name: string); overload;
class procedure SetStyle(Style: TCustomStyleServices); overload;
```

Первый из методов активирует стиль с именем `Name`, а второй стиль — `Style`.

Перед попыткой сменить один зарегистрированный стиль приложения на другой целесообразно воспользоваться услугами метода

```
class function TrySetStyle(const Name: string;
                           ShowErrorDialog: Boolean = True): Boolean;
```

Если подключение стиля с названием `Name` не представляет никаких сложностей, то метод возвратит значение `true` и применит стиль. Второй параметр метода разрешает (в состоянии `true`) или запрещает вывод окна с сообщением об ошибке подключения.

> ***ЗАМЕЧАНИЕ***
>
> Не стоит пытаться задействовать неизвестный менеджеру (незарегистрированный) стиль или, наоборот, повторно регистрировать уже известный стиль.

После всего изложенного нам осталось научить наше приложение оперативно менять "шкурку" не только при разработке, но и во время выполнения. Для этого достаточно воспользоваться примером из листинга 14.14. В единственный параметр предложенной процедуры направляются путь и имя файла стиля, все остальное сделает менеджер стилей `TStyleManager`.

Листинг 14.14. Подключение произвольного стиля к приложению

```
procedure TForm1.SetSkin(const filename: string);
var StyleInfo: TStyleInfo;
    Handle:TStyleManager.TStyleServicesHandle;
```

```
begin
  if TStyleManager.IsValidStyle(FileName,StyleInfo)=true then
  begin
    //проверяем возможность подключения стиля
    if TStyleManager.TrySetStyle(StyleInfo.Name,false)=false then
      begin
        //стиль следует загрузить и зарегистрировать
        Handle:=TStyleManager.LoadFromFile(FileName);
        TStyleManager.SetStyle(Handle);
      end;
  end;
end;
```

Завершая разговор о менеджере стилей `TStyleManager`, упомянем свойство

```
class property ActiveStyle: TCustomStyleServices;
```

Здесь хранится ссылка на текущий стиль оформления.

Утилита создания стилей оформления

Если у вас есть желание спроектировать собственный неповторимый стиль оформления приложения, то у меня для вас есть хорошая новость — специально для создания и редактирования стилей в состав Embarcadero RAD Studio XE2 включена утилита VCL Style Designer. Для доступа к утилите следует воспользоваться пунктом меню **Tools | VCL Style Designer**.

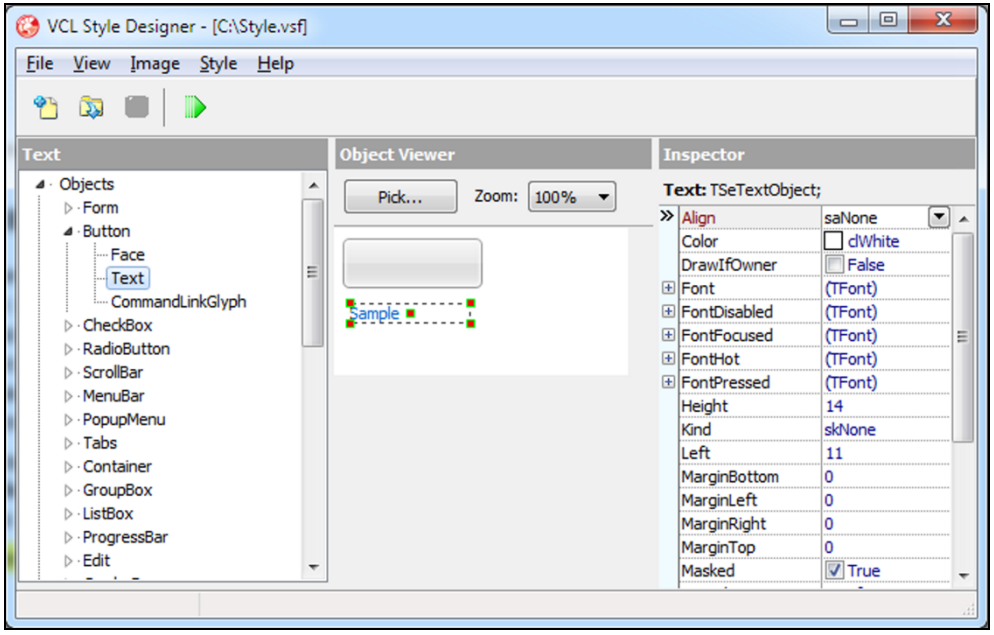


Рис. 14.6. Утилита создания стиля приложения VCL Style Designer

Для создания нового файла со стилем приложение обращаемся к пункту меню **File | New**, а если нам предстоит отредактировать уже существующую "шкурку", то выбираем **File | Open**.

Утилита предоставляет в распоряжение разработчика интуитивно понятный интерфейс. В левой части окна расположена иерархия объектов (**Objects**), изображений (**Images**), шрифтов (**Fonts**), цветов (**Colors** и **SysColors**). Развернув узел **Objects**, мы получаем доступ к перечню визуальных элементов управления VCL, способных изменять свое оформление.

Выбрав тот или иной элемент управления среди потомков узла **Objects**, обращаемся к инспектору (**Inspector**), позволяющему редактировать графические свойства объекта (рис. 14.6).

Подготовив свою версию стиля, стоит просмотреть результат. Для этого выбираем пункт меню **Style | Test** или просто нажимаем функциональную клавишу <F9>.

ГЛАВА 15

Форма, фрейм и модуль данных

В этой главе нам предстоит изучить классы `TForm`, `TFrame` и `TDataModule`, которые решают задачи построения пользовательского интерфейса приложения VCL и служат контейнерами для визуальных и невизуальных компонентов.

Форма проекта VCL, класс *TForm*

Основу подавляющего большинства приложений Delphi с графическим интерфейсом пользователя составляет экземпляр класса `TForm`, или просто форма. Основное назначение формы — служить контейнером для визуальных элементов управления, предоставляя программисту возможность создавать пользовательский интерфейс приложения, просто перенося подходящие компоненты со страниц палитры. После старта программы форма приобретает все необходимые качества полноценного окна приложения Windows. У окна имеется значок, область для вывода заголовка, стандартные кнопки управления (позволяющие сворачивать, разворачивать и закрывать окно), при необходимости окно готово создать горизонтальную и вертикальную полосы прокрутки (рис. 15.1).

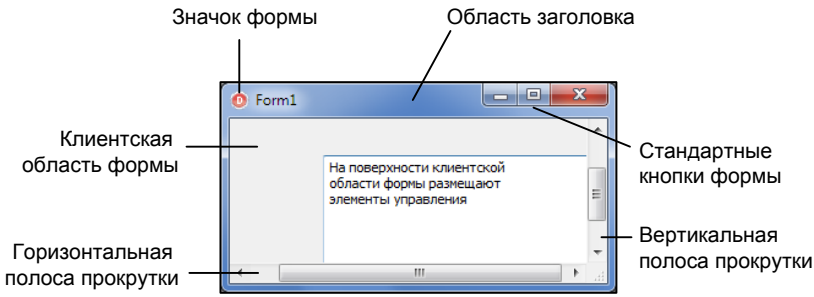


Рис. 15.1. Элементы окна приложения Windows

Число форм в проекте не ограничено, но в любом случае в приложении может быть только одна *главная форма* (main form) проекта. По умолчанию главной формой проекта назначается самая первая созданная форма — форма с названием `Form1`. При запуске приложения на выполнение именно эта форма будет создана и выведена на экран первой.

При необходимости у проекта, обладающего несколькими формами, на роль главной может быть назначена другая форма. Для этого следует воспользоваться пунктом меню **Project |**

Options и в появившемся дереве настройки опций проекта выбрать узел **Forms** (рис. 15.2). Главная форма выбирается в раскрывающемся списке **Main form**. Стоит заметить, что на роль главной формы могут претендовать только формы из списка **Auto-create forms** (Автоматически создаваемые формы). Форма из списка **Available forms** (Доступные формы) стать главной не сможет, т. к. она не подлежит автоматическому созданию в момент старта приложения.

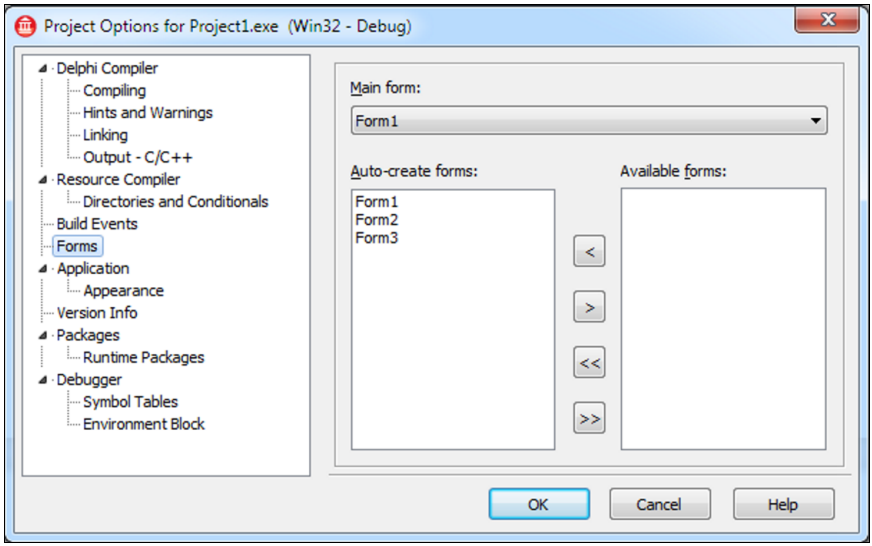


Рис. 15.2. Окно опций со страницей выбора главной формы и перечнем автоматически создаваемых форм

При разработке пользовательского интерфейса наших приложений мы всегда будем помнить, что закрытие главной формы приводит к автоматическому закрытию всех остальных форм проекта и прекращению работы приложения.

> ***Замечание***
>
> Несмотря на приставку "главная" в проектах VCL Forms Delphi, главная форма не является главным окном приложения. Главное окно приложения создается силами объекта `Application`, именно это скрытое окно и отвечает за взаимодействие приложения и Windows.

Взглянув на иерархию предков класса `TForm` (рис. 15.3), вы сразу поймете, что с большинством характеристик формы мы уже хорошо знакомы. Так, благодаря наличию в цепи предков формы класса `TWinControl` героиня этой главы приобретает право называться оконным элементом управления, со всеми вытекающими из этого особенностями (наличием дескриптора, умением получать фокус ввода, возможностью реагировать на сообщения Windows). Вместе с тем класс `TForm` обладает рядом специфичных возможностей, которые больше не встречаются ни в одном из компонентов.

Описание формы в dfm-файле

Добавление к проекту формы сопровождается созданием файла ресурсов формы с расширением dfm. Именно в этом файле и хранится текстовое описание формы и принадлежащих ей компонентов. Для того чтобы взглянуть на описание во время визуального проектирования,

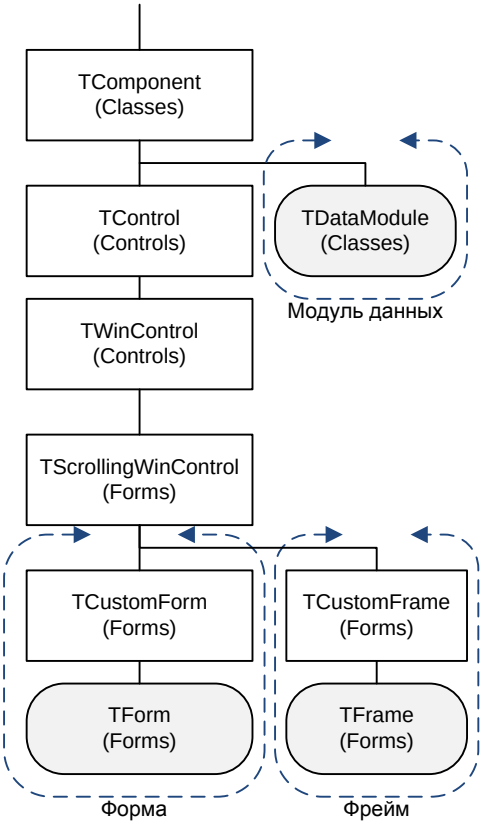


Рис. 15.3. Место формы `TForm` в иерархии VCL

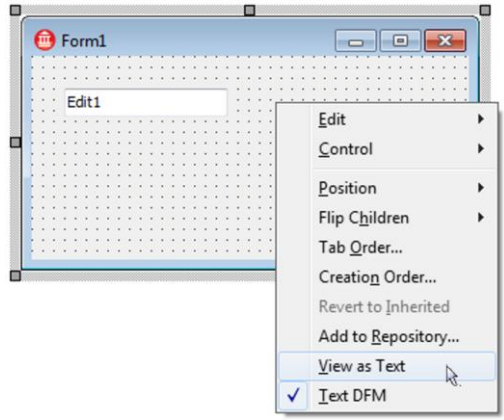


Рис. 15.4. Обращение к контекстному меню формы во время визуального проектирования

следует воспользоваться контекстным меню формы (рис. 15.4) и выбрать пункт **View as Text** (Представить как текст) или на клавиатуре нажать комбинацию клавиш <Alt>+<F12>.

Листинг с текстовым описанием ресурса формы с рис. 15.4 представлен в листинге 15.1. Обратите внимание на то, что кроме определения основных свойств формы ресурс dfm-файла содержит определение принадлежащей форме строки ввода — компонента `Edit1`.

Листинг 15.1. Описание файла ресурсов формы в текстовом формате

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Form1'
  ClientHeight = 151
  ClientWidth = 323
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
```

```
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Edit1: TEdit
    Left = 24
    Top = 24
    Width = 121
    Height = 21
    TabOrder = 0
    Text = 'Edit1'
  end
end
```

ВНИМАНИЕ!

Для того чтобы вернуть форму к обычному представлению, нажмите комбинацию клавиш <Alt>+<F12>.

Стиль, поведение и оформление формы

При разработке нового программного продукта первым шагом должно стать действие, определяющее стиль главной формы приложения.

```
property FormStyle: TFormStyle; //по умолчанию fsNormal
type TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop);
```

Стиль `fsNormal` предназначен для построения приложений с *однодокументным интерфейсом* (Single Document Interface, SDI). Если мы проектируем программу с *многодокументным интерфейсом* (Multiple Document Interface, MDI), то стиль родительской формы определяется как `fsMDIForm`. Дочерние окна приложения MDI — `fsMDIForm`. И наконец, стиль `fsStayOnTop` заставит форму разместиться поверх других окон, форма такого вида даже при потере фокуса ввода по-прежнему останется над формами данного приложения.

Пояснительная надпись в заголовке формы назначается при обращении к свойству

```
property Caption: TCaption;
```

Местоположение формы на экране компьютера зависит от значения, установленного в свойстве

```
property Position: TPosition; //по умолчанию poDefaultPosOnly
type TPosition = (poDesigned, poDefault, poDefaultPosOnly,
                  poDefaultSizeOnly, poScreenCenter, poDesktopCenter,
                  poMainFormCenter, poOwnerFormCenter);
```

При создании новой формы свойство `Position` принимает значение `poDefaultPosOnly`. Это означает, что вертикальный и горизонтальный размеры формы (свойства `Width` и `Height`) назначаются разработчиком, а место вывода формы на экран определяется операционной системой. Обратная ситуация сложится в том случае, если свойство `Position` установлено в `poDefaultSizeOnly`. Теперь Windows станет выводить форму в месте, указанном программистом (свойства `Top` и `Left`), но ответственность за установку размеров сторон заберет в свои руки. Форма появится точно в центре экрана при передаче в `Position` значения `poScreenCenter`, а для центровки относительно рабочего стола подойдет значение `poDesktopCenter`. На поведение вторичных форм проекта влияют `poMainFormCenter` и `poOwnerFormCenter`. В первом случае дополнительные формы позиционируются по центру

главной формы, во втором — по центру формы-владельца. Интересно поведение формы при установке этого свойства в `poDefault`. При первом запуске приложения форма отображается в левом верхнем углу экрана, а при каждом последующем выводе на экран форма станет автоматически смещаться вправо и вниз. Высота и ширина формы определяются операционной системой и не зависят от указаний программиста. Если вы намерены самостоятельно управлять как размерами, так и местом вывода формы, то установите свойство в состояние `poDesigned`.

Перечень стандартных кнопок, располагающихся в правой части заголовка формы, задается свойством

```
property BorderIcons :TBorderIcons;
```

Различают: кнопку закрытия окна `biSystemMenu`; кнопки сворачивания `biMinimize` и разворачивания `biMaximize` окна; кнопку обращения к справочной системе `biHelp`.

Щелчки пользователя по кнопкам сворачивания и разворачивания окна изменяют значение свойства

```
property WindowState: TWindowState;
```

Состояние `wsNormal` говорит о том, что форма в нормальном состоянии; `wsMinimized` — форма свернута; `wsMaximized` — форма развернута до максимально возможного размера. Свойство `WindowState` не только информационное, оно позволяет сворачивать, разворачивать и нормализовать размеры формы во время выполнения приложения.

Существенное влияние на внешний вид и поведение формы оказывает стиль обрамления окна:

```
property BorderStyle: TFormBorderStyle;
type TFormBorderStyle = (
   bsNone,          //размеры формы неизменяемые, границы бордюра невидимы
   bsSingle,        //размеры неизменяемые, простой бордюр
   bsSizeable,      //обычная форма с настраиваемыми размерами
   bsDialog,        //стандартное окно диалога с неизменяемыми размерами
   bsToolWindow,    //размеры неизменяемые, уменьшенный заголовок
   bsSizeToolWin); //размеры изменяются, уменьшенный заголовок
```

По умолчанию толщина бордюра (отступ от внешней границы окна до его клиентской части) минимальна — 0 пикселов. Для увеличения отступа измените значение в свойстве

```
property BorderWidth: TBorderWidth;
type TBorderWidth = 0..MaxInt;
```

Практически все формы в левом углу заголовка отображают значок приложения. По умолчанию пиктограмму вновь создаваемого приложения определяет Delphi, присваивая всем формам значок `Application.Icon`. Если этот рисунок не отвечает вашему вкусу, то, используя свойство:

```
property Icon: TIcon;
```

свяжите с формой другое изображение.

Состояние формы

Текущее состояние формы определяется с помощью свойства:

```
property FormState: TFormState; //только для чтения
type TFormState = set of (
    fsCreating,        //создание формы
```

```
    fsVisible,          //форма выведена на экран в обычном режиме
    fsShowing,          //форма выводится на экран
    fsModal,            //форма выведена в модальном режиме
    fsCreatedMDIChild, //создание дочерней MDI-формы
    fsActivated);       //форма стала активной
```

Состояние `fsActivated` окна продублировано еще в одном свойстве:

```
property Active: Boolean;
```

Вне зависимости от числа запущенных приложений и количества открытых окон в Windows *активным* может быть только одно окно. Визуальным признаком активности окна служит цвет заголовка окна. В отличие от `FormState` это поле доступно для записи. Свойство возвращает значение `true`, если форма активна.

Также продублировано значение `fsVisible`. Для этого необходимо обратиться к свойству, определяющему видимость формы `Visible`. Если форма видима (но необязательно активна), свойство возвратит значение `true`.

> ***Замечание***
>
> Перевод окна в видимое состояние можно производить с помощью методов `Show()` или `ShowModal()`, в последнем случае окно выводится в модальном режиме.

Создание, отображение и уничтожение форм

По умолчанию каждая подключаемая к проекту форма (пункт меню **File | New | Form — Delphi**) при старте программы заносится в список форм, подлежащих автоматическому созданию. При запуске приложения первой на свет появляется главная форма проекта, все остальные формы создаются согласно очередности, указанной программистом. Для изменения очередности создания форм надо заглянуть в опции проекта (**Project Options**) и расставить формы в надлежащем порядке в списке **Auto-create forms** (см. рис. 15.2). За создание форм из перечня **Auto-create forms** несет ответственность приложение `Application`. Воспользовавшись пунктом меню **Project | View Source**, вы получите доступ к головному модулю проекта и увидите, что для создания автоматически создаваемых форм приложение использует метод `CreateForm()`.

Среда проектирования позволяет нам отказаться от автоматического создания форм, такие формы переводятся в разряд *доступных* (список **Available forms**). Но в этом случае перед обращением к такой форме программист должен ее создать.

По законам ООП для создания объекта требуется воспользоваться его конструктором, и форма не является исключением из правил:

```
constructor Create(AOwner: TComponent);
```

Допустим, что описанная в программном модуле child.pas форма `TfrmChild` не является автоматически создаваемой и входит в список доступных форм нашего проекта. В этом случае для создания экземпляра формы можно воспользоваться листингом 15.2.

Листинг 15.2. Создание экземпляра формы с помощью конструктора

```
implementation
uses child; //подключение к проекту модуля с формой
{$R *.dfm}
```

```
procedure TfrmMain.Button1Click(Sender: TObject);
begin
  with TfrmChild.Create(Application) do
  begin
    Caption:=Caption+'$'+IntToHex(Handle,8);
    Show; //вывод окна на экран
  end;
end;
```

> ***ВНИМАНИЕ!***
>
> При ссылке одного модуля на другой не забудьте добавить в строку `uses` имя нового модуля. Для этого выберите пункт меню **File | Use unit** (<ALT>+<F11>) и в списке модулей найдите нужный модуль.

Форма `TfrmChild` создается по щелчку на кнопке `Button1`, расположенной на главной форме проекта `frmMain`. Основная особенность предложенного в листинге 15.2 кода в том, что будет создано ровно столько клонов формы `TfrmChild`, сколько вы сделаете щелчков по кнопке. Чтобы формы отличались друг от друга, мы выводим в заголовке создаваемых форм их дескриптор `Handle`.

> ***ЗАМЕЧАНИЕ***
>
> Когда проект содержит несколько форм, хорошей практикой является присвоение формам осмысленных имен. Например, главную форму проекта можно называть `frmMain`, а соответствующий ей программный модуль — Main.pas.

Для отображения формы на экране компьютера обычно применяют метод:

```
procedure Show;
```

Такой способ вывода формы на экран называется *немодальным*. Это означает, что пользователь получает право свободно переключаться между окнами приложения.

Вывод формы в модальном режиме

Для вызова формы в модальном режиме (режиме диалога) используйте метод

```
function ShowModal: Integer;
```

Модальная форма отличается строптивым нравом. Она отображается поверх всех открытых окон. Более того, пользователь не сможет получить доступа к другим формам проекта, пока не будет закрыта модальная форма. Закрываясь, модальная форма возвращает целочисленное значение, называемое *модальным результатом*. При необходимости в модальном результате можно закодировать решение пользователя, нажавшего ту или иную кнопку на форме.

```
type TModalResult = Low(Integer)..High(Integer);
```

Возможные значения модального результата предложены в табл. 15.1.

Таблица 15.1. *Стандартный модальный результат* `TModalResult`

Константа	Описание
`mrNone` или `0`	Значение по умолчанию
`mrOk` или `idOK`	Пользователь нажал кнопку **OK**
`mrCancel` или `idCancel`	Пользователь нажал кнопку **Отмена** (Cancel)

Таблица 15.1 *(окончание)*

Константа	Описание
`mrAbort` или `idAbort`	Пользователь нажал кнопку **Прервать** (Abort)
`mrRetry` или `idRetry`	Пользователь нажал кнопку **Повторить** (Retry)
`mrIgnore` или `idIgnore`	Пользователь нажал кнопку **Пропустить** (Ignore)
`mrYes` или `idYes`	Пользователь нажал кнопку **Да** (Yes)
`mrNo` или `idNo`	Пользователь нажал кнопку **Нет** (No)
`mrAll` или `mrNo + 1`	Используется для определения последней константы

Для передачи результата модальной форме на ее поверхности обычно размещают кнопки класса `TBitBtn` *(см. главу 18)*. Анализируя модальный результат закрывающейся формы, программист получает возможность направлять поведение программы в определенное русло (листинг 15.3).

Листинг 15.3. Вывод формы в модальном режиме

```
if Form2.ShowModal=mrOK then
begin
  {операторы для результата mrOK}
end else {операторы для результата отличного от mrOK};
```

Результат модальной операции заносится в свойство формы

```
property ModalResult: TModalResult;
```

По умолчанию это свойство установлено в 0 (константа `mrNone`). Присвоение свойству `ModalResult` ненулевого значения приводит к закрытию формы и возвращению значения этого свойства в качестве результата метода `ShowModal`. Если получено ненулевое значение `ModalResult`, осуществляется вызов метода закрытия формы.

Закрытие формы

Простейшим способом спрятать (не уничтожая) форму может стать присвоение свойству `Visible` значения `false`. Этот же результат достигается при обращении к процедуре

```
procedure Hide;
```

Кроме того, у формы имеется метод

```
procedure Close;
```

предназначенный для закрытия формы. Если этот метод применяется к главной форме проекта, то он приводит к закрытию всех выведенных на экран форм и завершению работы приложения. Если метод `Close()` вызывается для вторичных форм проекта, то форма просто скрывается с экрана.

> ***Замечание***
>
> С методом `Close()` связаны два важных события — `OnCloseQuery()` и `OnClose()`, о них мы поговорим отдельно (см. табл. 15.3).

Уничтожение формы

Хотя форма поддерживает классический для всех компонентов деструктор `Destroy` и метод `Free()`, для уничтожения экземпляра формы целесообразно вызывать метод

```
procedure Release;
```

Это ближайший соратник деструктора и добивается тех же результатов — очищает память компьютера от формы, но в отличие от деструктора делает это самым деликатным образом — дожидается завершения обработки событий, инициированных этой формой или ее компонентами.

Подключение меню

Форма способна обладать одним главным (`TMainMenu`) и контекстным меню (`TPopupMenu`). При размещении на рабочей форме главного меню (класс `TMainMenu`) в свойстве

```
property Menu: TMainMenu;
```

создается ссылка на этот компонент.

Возможность подключения контекстного меню унаследована от дальнего родственника `TControl`. Для этого у формы объявлено свойство:

```
property PopupMenu: TPopupMenu;
```

Родительская форма MDI приложения владеет секретом, позволяющим автоматически добавлять в главное меню приложения список открытых документов — дочерних форм MDI. Для этого в свойстве главной формы

```
property WindowMenu: TMenuItem;
```

укажите пункт меню, к которому будет присоединен список дочерних форм. Как правило, список открытых документов добавляется к пункту **Окно** (Window). И еще одно примечание — в качестве `WindowMenu` стоит назначать только пункт меню самого верхнего уровня.

> ***Внимание!***
>
> В современных версиях Delphi на роль главного меню вместо компонента `TMainMenu` целесообразно выбирать полосу меню на основе команд `TActionMainMenuBar` в союзе с менеджером команд `TActionManager`.

Фокус ввода

Содержащая оконные элементы управления родительская форма всегда обладает информацией об активном (владеющим фокусом ввода) элементе управления

```
property ActiveControl: TWinControl;
```

Также это свойство позволяет установить фокус ввода на один из оконных элементов управления. Ко всему прочему, свойство доступно и во время разработки, и если программист укажет в этом свойстве на какой-нибудь компонент, то выбранный элемент управления получит фокус ввода при активизации формы.

> ***Замечание***
>
> Во время проектирования пользовательского интерфейса можно настроить порядок переключения фокуса ввода между элементами управления формы. Для этого следует воспользоваться пунктом контекстного меню формы **Tab Order**.

Для передачи фокуса другому размещенному на форме компоненту во время работы приложения кроме выше предложенного свойства обращайтесь к методу:

```
procedure FocusControl(Control: TWinControl);
```

Для решения обратной задачи — снятия фокуса с элемента управления `TWinControl` предназначен метод:

```
procedure DefocusControl(Control: TWinControl; Removing: Boolean);
```

Если форма проекта содержит элементы управления OLE, то по аналогии с обычными компонентами элемент OLE может быть активен. Для этого необходимо установить ссылку на объект OLE в свойстве:

```
property ActiveOLEControl: TWinControl;
```

Полосы прокрутки

Форма обладает возможностью самостоятельно создавать полосы прокрутки благодаря наличию в своей иерархии наследования класса `TScrollingWinControl`. Появление полос прокрутки является признаком того, что геометрические размеры окна не позволяют вместить в себя все компоненты, размещенные в его клиентской области. Форма обладает двумя полосами прокрутки: горизонтальной и вертикальной. Доступ к данным полосам осуществляется благодаря свойствам

```
property HorzScrollBar: TControlScrollBar;
property VertScrollBar: TControlScrollBar;
```

Обратите внимание, что свойства возвращают ссылку на экземпляр класса `TControlScrollBar`, именно этот класс выполняет функционал полосы прокрутки.

За автоматическое включение полос прокрутки несет ответственность свойство формы

```
property AutoScroll : Boolean;
```

Если оно установлено в `true`, то форма берет на себя обязательства в случае необходимости автоматически включать вертикальную и горизонтальную полосы прокрутки.

К ключевым характеристикам полосы прокрутки (экземпляр класса `TControlScrollBar`) стоит отнести свойства, влияющие на процесс перемещения клиентской части окна. И в первую очередь это диапазон допустимых значений прокрутки

```
property Range: Integer;
```

и текущее положение (позиция) бегунка полосы:

```
property Position: Integer;
```

Диапазон значений представляет собой виртуальный размер (в пикселах) связанной с линией прокрутки клиентской части формы. Значения `Range` горизонтальной и вертикальной полос прокрутки должны превышать клиентский размер (`ClientWidth` и `ClientHeight` соответственно) формы.

Положение бегунка может принимать любое значение в пределах диапазона (`Range`) полосы. Следующий пример (листинг 15.4) устанавливает бегунок горизонтальной полосы прокрутки в ее центральной позиции.

Листинг 15.4. Установка положения бегунка горизонтальной полосы прокрутки

```
with Form1 do
begin
  HorzScrollBar.Range:=ClientWidth*2;
  HorzScrollBar.Position:= HorzScrollBar.Range div 2;
end;
```

При каждом щелчке по кнопкам полосы прокрутки позиция бегунка получает приращение в соответствии со значением, заданным в свойстве

```
property Increment: TScrollBarInc; //диапазон от 1 до 32 767
```

По умолчанию значение шага составляет 8 пикселов.

Творческие натуры могут поэкспериментировать со свойствами, отвечающими за размеры и внешний вид полос прокрутки. К таким свойствам относятся:

```
property Size: Integer;          //ширина полосы в пикселах
property ButtonSize: Integer;    //размер кнопок
property Style: TScrollBarStyle; //внешнее представление полос
TScrollBarStyle = (ssRegular, ssFlat, ssHotTrack);
```

Мы привыкли к тому, что полоса прокрутки появляется только в ситуации, когда элемент управления, скажем, кнопка, не помещается в клиентской области окна. С помощью свойства

```
property Margin: Word;
```

можно задать отступ от края оконного элемента управления до видимой границы формы, при котором будет показываться полоса прокрутки даже в том случае, если элемент управления полностью помещается в клиентской области.

Особенности графического вывода формы

Не редки случаи, когда на форме расположено большое число самых разнообразных элементов управления. Кроме того, программисты очень часто пользуются услугами события `OnPaint()` формы для вывода в ее клиентской области дополнительной графической информации *(см. главу 31)*. В той ситуации, когда форма перегружена визуальными эффектами, не исключено, что она не успеет качественно прорисоваться за отведенные для этого кванты процессорного времени. Если графический интерфейс формы перегружен, то для улучшения качества вывода формы целесообразно перевести в `true` свойство

```
property DoubleBuffered:boolean; //по умолчанию false
```

В режиме двойной буферизации формы прорисовка формы осуществляется в два приема. На первом этапе прорисовка формы происходит в оперативной памяти компьютера. После того как рисунок будет завершен, его битовый образ переносится из ОЗУ на экран быстрой растровой операцией.

Выбор монитора для вывода формы

Современные версии Windows допускают одновременный вывод на несколько устройств отображения, например на два монитора или на монитор и проектор. Для выбора устройства графического вывода по умолчанию следует обратиться к свойству

```
property DefaultMonitor: TDefaultMonitor;
```

Масштабирование шрифта

Обсудим еще одну особенность графического вывода формы, точнее говоря, текстовой информации формы. Обычно над созданием приложения программист работает на качественных дисплеях с характеристиками не хуже, чем 1280 × 1024 пикселов, с разрешением не ниже 96 точек на дюйм. Вместе с этим нередко пользователь эксплуатирует программу на мониторах с гораздо худшими показателями. Во время проектирования рабочего интерфейса программист обычно не меняет характеристики шрифта (по умолчанию высота шрифта равна 8 пунктам), однако при пропорциональном уменьшении размеров рабочих форм для работы на мониторах низкого качества шрифт также может уменьшиться до неприемлемых размеров. Для снижения остроты этой проблемы в состав свойств формы введено свойство

```
property PixelsPerInch:integer;
```

сохраняющее рабочее разрешение дисплея (число пикселов в дюйме), на котором осуществлялось проектирование формы. Зная этот показатель, при уменьшении своих размеров форма постарается сохранить такую высоту шрифта, чтобы текст оставался читабельным. Корректное масштабирование шрифта возможно лишь при условии, что в значение `true` установлено свойство

```
property Scaled: Boolean;
```

Эффект прозрачности

Delphi предлагает программистам простой и одновременно действенный механизм создания эффекта прозрачности для определенного участка формы. Эффект прозрачности применяется к пикселам формы, окрашенным в заранее определенный цвет

```
property TransparentColorValue: TColor;
```

После этого достаточно перевести в значение `true` свойство

```
property TransparentColor: Boolean;
```

Участок формы становится не просто зрительно прозрачным. Прозрачный участок исключается из региона отсечения, так что вы получите возможность обратиться к объекту, находящемуся под прозрачным участком формы. Чтобы убедиться в этом, достаточно добавить две строки кода в событие `OnShow()`, генерируемое в момент вывода формы на экран (листинг 15.5).

Листинг 15.5. Назначение прозрачного цвета

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Form1.TransparentColorValue:=Form1.Color;
  Form1.TransparentColor:=True;
end;
```

Результат выполнения кода представлен на рис. 15.5.

Альфа-канал

Графический интерфейс приложений для современных версий Windows зачастую опирается на использование альфа-канала, позволяющего программисту определять степень прозрачности формы.

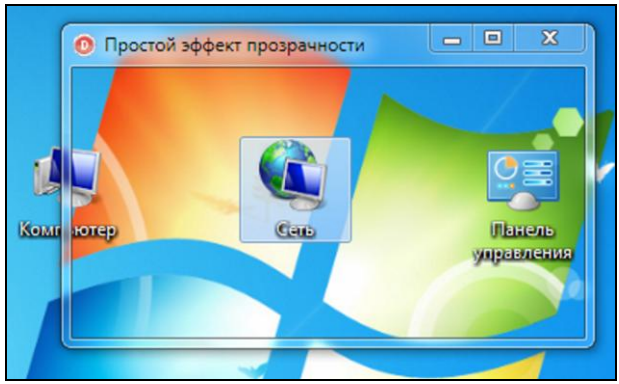


Рис. 15.5. Форма с прозрачной поверхностью

Степень прозрачности назначается с помощью свойства

```
property AlphaBlendValue: Byte; //диапазон 0..255
```

Нулевое значение соответствует абсолютно прозрачной форме. Увеличивая значение, мы снижаем эффект прозрачности. Значению 255 соответствует полностью непрозрачная форма.

Для активизации эффекта следует установить в `true` свойство

```
property AlphaBlend: Boolean;
```

> ***Замечание***
>
> Эффект прозрачности на основе альфа-канала затрагивает все принадлежащие форме визуальные элементы управления.

Эффект размытого стекла

Формы Delphi поддерживают популярный графический эффект Windows Aero — эффект размытого стекла. Для этого в состав формы внедрен фрейм `TGlassFrame`, доступ к которому осуществляется благодаря свойству

```
property GlassFrame: TGlassFrame;
```

Фрейм создается автоматически в момент появления на свет формы. Для активации возможностей "стеклянного" фрейма следует включить его свойство

```
property Enabled: Boolean;
```

Для конкретизации, какую именно область формы необходимо сделать размытой, следует воспользоваться свойствами `Top`, `Left`, `Right` и `Bottom` (по умолчанию они принимают нулевые значения).

Для распространения эффекта Aero glass на всю область формы можно установить в `true` свойство

```
property SheetOfGlass: Boolean;
```

Получаемый результат весьма впечатляет (рис. 15.6), однако поверхность формы в виде размытого стекла плохо подходит для размещения на ней других элементов управления. Поэтому в деловых приложениях эффект надо применять осторожно.

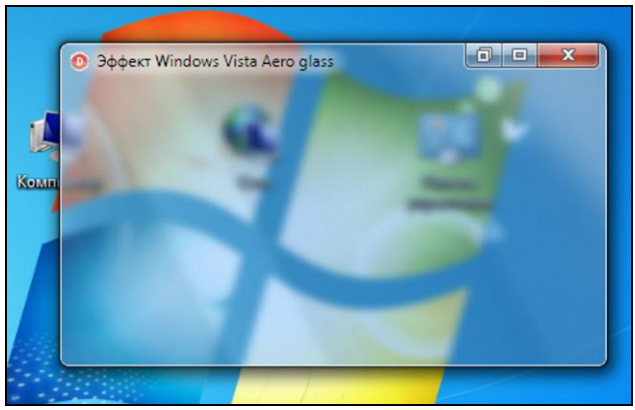


Рис. 15.6. Применение фрейма `TGlassFrame` для получения эффекта размытого стекла

Дескрипторы окна

Для профессиональных программистов, активно применяющих в своих программах функции Windows API, надо знать, что дескриптор окна содержится в свойстве

```
property Handle: HWND;
```

Дескриптор клиентской области окна в свойстве:

```
property ClientHandle: HWND;
```

Оба дескриптора доступны только для чтения и автоматически устанавливаются в момент создания формы.

Прикрепление формы к границам экрана

Когда приложение состоит из нескольких подлежащих одновременному выводу форм, пользователю бывает достаточно сложно быстро разместить эти формы на своем рабочем столе так, чтобы они, с одной стороны, оставались легкодоступными, а с другой — эффективно использовали площадь экрана. Для того чтобы ускорить решение задачи размещения форм, можно воспользоваться услугами свойств

```
property ScreenSnap: Boolean; //по умолчанию false
property SnapBuffer: Integer; //по умолчанию 10 пикселов
```

Как только перемещаемая пользователем форма (у которой `ScreenSnap=true`) подойдет к любой из границ рабочего стола на расстояние, установленное в свойстве `SnapBuffer`, форма самостоятельно примкнет к границе окна.

Обработка событий формы

Как и любой оконный элемент управления, форма способна реагировать на стандартные события системы, которые унаследованы ею от ее предков — классов `TControl` и `TWinControl`. Кроме того, форма может похвастаться рядом собственных обработчиков событий (в основном объявленных на уровне класса `TCustomForm`).

Жизненный цикл формы

Жизненный путь формы начинается в момент ее создания. В табл. 15.2 приведена последовательность событий, возникающих при создании и выводе на экран компьютера новой формы.

Таблица 15.2. *Процесс создания и вывода на экран форм*

Событие	Описание
`OnCreate()`	Создание формы. Обычно этот обработчик события используется для инициализации глобальных переменных формы и других подготовительных операций, необходимых для дальнейшей работы с формой
`OnShow()`	Предваряет вывод формы на экран, событие генерируется в момент обращения к методу `Show()` (или `ShowModal()`) или присвоения свойству `Visible` формы значения `true`
`OnActivate()`	Форма становится активной
`OnCanResize()`	Форма проверяет корректность своих новых размеров
`OnConstrainedResize()`	Проверка на соответствие размеров ограничениям свойства `Constraints`
`OnResize()`	Генерируется сразу после изменения размеров формы
`OnPaint()`	Осуществляется окончательная перерисовка формы

Не меньший интерес представляют события, возникающие при закрытии формы. Для запуска процесса закрытия формы необходимо закрыть форму или вызвать метод

```
procedure Close;
```

Последовательность событий, возникающих у формы в процессе закрытия, представлена в табл. 15.3.

Таблица 15.3. *Процесс закрытия и уничтожения формы*

Событие	Описание
`OnCloseQuery()`	Запрос разрешения на закрытие формы
`OnClose()`	Форма закрывается
`OnDeactivate()`	Форма перестает быть активной
`OnHide()`	Форма скрывается с экрана
`OnDestroy()`	Форма уничтожается

Более подробно последовательность событий отражена на фрагменте блок-схемы закрытия формы (рис. 15.7).

Перед закрытием формы производится вызов обработчика события:

```
property OnCloseQuery: TCloseQueryEvent;
type TCloseQueryEvent = procedure(Sender: TObject;
                        var CanClose: Boolean) of object;
```

Присвоив параметру-переменной `CanClose` значение `false`, программист отменяет процесс закрытия формы. В листинге 15.6 предложен пример, осуществляющий контроль за изме-

нением текста в многострочном редакторе Memo1. В случае если в элементе управления текст модифицировался, то при попытке закрытия формы отображается диалоговое окно, предлагающее сохранить изменения.

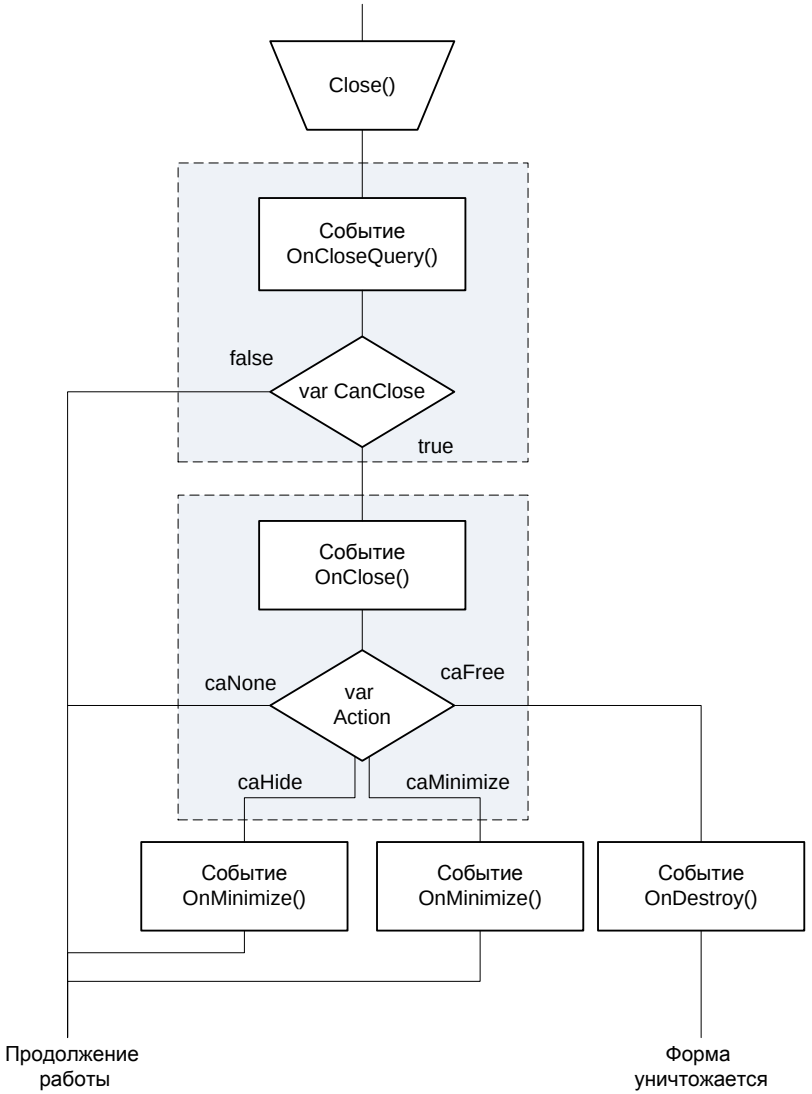


Рис. 15.7. Блок-схема закрытия формы проекта

Листинг 15.6. Контроль сохранения данных при закрытии формы

```
procedure TfrmMain.FormCloseQuery(Sender: TObject;
                                  var CanClose: Boolean);
begin
if Memo1.Modified=true then {если текст модифицирован}
    case MessageBoxEx(frmMain.handle,
```

```
              'Сохранить изменения в тексте?',
              'Подтвердите операцию',
              MB_YESNOCANCEL OR MB_ICONQUESTION, LANG_RUSSIAN) of
    mrYes : {код сохранения данных, в простейшем случае
             Memo1.Lines.SaveToFile(...)};
    mrNo : CanClose := true; //отказ от сохранения, форма закрывается
    else CanClose := false;  //форма не закрывается
  end;
end;
```

Если форма получила разрешение на закрытие, то генерируется событие

```
property OnClose: TCloseEvent;
TCloseEvent = procedure(Sender: TObject;
                        var Action: TCloseAction) of object;
```

Изменяя значение переменной `Action` (табл. 15.4), программист сможет управлять поведением закрываемой формы.

Таблица 15.4. *Возможные значения параметра* `Action:TCloseAction`

Значение	Описание
`caNone`	Ничего не происходит
`caHide`	Форма не закрывается, а только становится невидимой, к данной форме приложение имеет полный доступ. Значение по умолчанию для дочерних SDI-форм
`caFree`	Форма закрывается и освобождает занимаемые ресурсы
`caMinimize`	Вместо закрытия форма сворачивается. Значение по умолчанию для дочерних MDI-форм

Реальное уничтожение формы и освобождение всех задействованных в ее интересах системных ресурсов произойдет только в случае, если присвоить переменной `Action` значение `caFree`. Только тогда будет вызвано событие

```
property OnDestroy: TNotifyEvent;
```

При этом форма и все объекты ей принадлежащие удаляются из памяти компьютера.

Нажатие быстрых клавиш

При рассмотрении класса `TWinControl` мы встретились с тремя основными событиями, связанными с нажатием пользователем клавиш: `OnKeyDown()`, `OnKeyPress()` и `OnKeyUp()`. Кроме перечисленных событий у формы предусмотрена дополнительная реакция на использование быстрых клавиш:

```
property OnShortCut: TShortCutEvent;
TShortCutEvent = procedure (var Msg: TWMKey;
                            var Handled: Boolean) of object;
```

В очередности событий, связанных с работой на клавиатуре, `OnShortCut()` вызывается самым первым и соответственно успевает обработать комбинацию быстрых клавиш до того, как за дело берутся другие обработчики события (как формы, так и элементов управления).

Для того чтобы не допустить дальнейшую обработку нажатых клавиш в последующей цепочке событий, установите параметр `Handled` в `true`, тем самым вы укажете Windows, что

обработка завершена. Параметр `Msg` соответствует стандартному сообщению Windows для обработки нажатия клавиш (листинг 15.7).

Листинг 15.7. Объявление записи `TWMKey`

```
type
  TWMKey = packed record
    Msg: Cardinal;
    CharCode: Word;
    Unused: Word;
    KeyData: Longint;
    Result: Longint;
end;
```

Здесь параметр `CharCode` содержит код виртуальной клавиши (см. табл. 13.1). Параметр `KeyData` содержит дополнительную информацию: число повторов данного сообщения, предыдущее состояние клавиши, состояние дополнительных клавиш (таких как <Alt>, <Ctrl>). В поле `Result` будет записан результат обработки.

Простейший пример использования данного обработчика события приведен в листинге 15.8, здесь нажатие клавиши <Esc> (код `VK_ESCAPE`) приведет к закрытию формы.

Листинг 15.8. Перехват нажатия клавиши выхода

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled: Boolean);
begin
  if Msg.CharCode=VK_ESCAPE then Close;
end;
```

Форма имеет возможность перехватывать все клавиатурные события, а точнее говоря, сообщения от клавиатуры, направленные в адрес активного, принадлежащего форме элемента управления. Для этого следует установить в `true` свойство

```
property KeyPreview: Boolean;
```

С этого момента форма станет первой получать сообщения, направленные ее элементам управления (листинг 15.9).

Листинг 15.9. Перехват нажатия клавиши выхода

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Form1.ActiveControl=Edit1) and (Key =VK_Escape)
    then  Edit1.Clear;
end;
```

Пользовательские интерфейсы SDI и MDI

Обсуждая свойство `FormStyle` формы, мы уже упомянули о существовании двух подходов к проектированию пользовательского интерфейса приложения, содержащего несколько ра-

бочих форм. Создаваемый по умолчанию проект VCL Forms Application нацелен на построение приложения с *однодокументным интерфейсом* (Single-Document Interface, SDI), альтернативным подходом считается *многодокументный интерфейс* (Multiple Document Interface, MDI).

Каждый из указанных стилей интерфейса обладает своими уникальными возможностями и, соответственно, особенностями программирования. Характерной особенностью приложения MDI является то, что все дочерние окна размещаются исключительно внутри клиентской области главного окна приложения. Вы не сможете переместить дочерние окна за пределы главного, такая попытка просто приведет к появлению полос прокрутки у главной формы проекта.

Приложение SDI также состоит из главного и дочерних окон, но на этот раз дочерние окна SDI имеют гораздо больше степеней свободы, они создаются со стилем `FormStyle=fsNormal`, поэтому не столь крепко привязаны к главной форме проекта и могут свободно "разгуливать" по всему пространству рабочего стола. Вместе с этим главная форма проекта SDI утрачивает некоторые возможности приложения MDI, например, она не в состоянии самостоятельно упорядочивать свои дочерние окна на экране.

Выбор интерфейса будущего программного продукта — индивидуальное право программиста, определяемое его личными предпочтениями. Интерфейс MDI целесообразно применять в проекте, предназначенном для одновременной работы с несколькими однотипными документами (например, в текстовых и в графических редакторах). Интерфейс SDI более универсален и может использоваться во всех остальных типах проектах.

Особенности проекта MDI

Познакомимся с основными аспектами построения проекта с многодокументным интерфейсом. Для этого разработаем текстовый редактор, который по своему функционалу будет схож с классическим Блокнотом из состава Windows. В отличие от своего известного "коллеги", наше творение сможет одновременно работать с несколькими текстовыми документами.

Свою работу начнем с создания нового проекта. Пока единственной форме проекта укажем, что она станет главной формой MDI `FormStyle:=fsMDIForm`. Кроме того, предлагаю отказаться от присвоенного по умолчанию невыразительного имени `Form1` и дать форме осмысленное название `Name:=frmMain`. Сохраните проект в отдельной папке. Во время сохранения переименуйте программный модуль главной формы в Main.pas, имя проекта выберите на свое усмотрение.

Разместите на форме следующие компоненты:

- диалог `TOpenTextFileDialog` (страница палитры компонентов **Dialogs**), компонент позволит пользователю выбирать на жестком диске текстовые файлы и открывать их;
- диалог `TSaveTextFileDialog` (страница **Dialogs**), компонент предназначен для сохранения текстового файла на диске;
- главное меню проекта `TMainMenu` (страница **Standard**).

Двойной щелчок по компоненту `MainMenu1` вызовет редактор главного меню. Создайте в нем минимальный набор элементов, позволяющих открывать текстовые файлы и сохранять их на жестком диске (рис. 15.8).

Добавим к проекту новую форму, именно она станет дочерним MDI-окном. Для этого воспользуйтесь пунктом меню **File | New | Form**. Для того чтобы форма поняла, что она попала

в проект с интерфейсом MDI, измените ее стиль `FormStyle:=fsMDIChild` и поменяйте название `Name:=frmChild`. Вновь сохраните проект, при этом переименуйте программный модуль дочерней формы в child.pas.

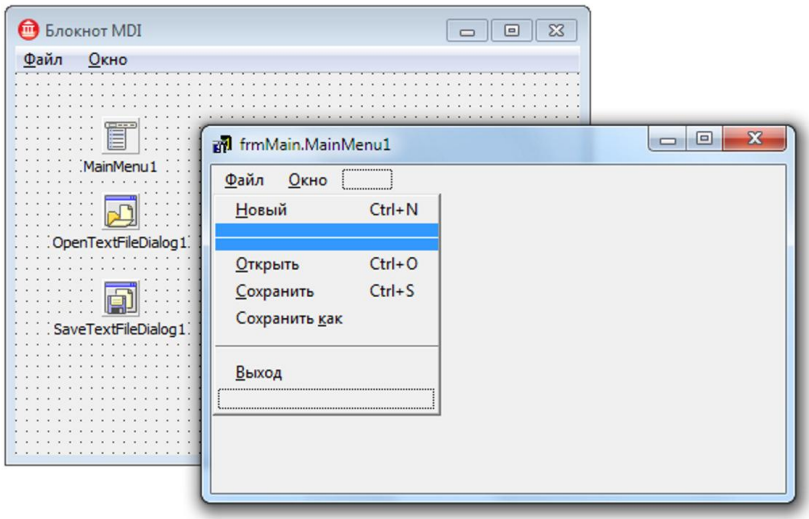


Рис. 15.8. Проект приложения MDI с открытым редактором меню

Разместите на дочерней форме единственный элемент управления — многострочный редактор `TMemo`. Именно этот компонент позволит отобразить и редактировать текстовый файл. Кроме того, в секции публичных объявлений дочерней формы следует упомянуть переменную строкового типа (листинг 15.10), она возьмет ответственность за хранение имени открытого файла.

Листинг 15.10. Объявление строковой переменной

```
unit child;
...
public
    fName:string;//путь и имя обслуживаемого файла
//...
```

Если вы сейчас запустите проект на выполнение, то вместе с главной формой приложения будет автоматически создана и дочерняя. Чтобы исключить эту ненужную операцию, следует удалить форму `frmChild` из перечня создаваемых по умолчанию, переместив ее в список доступных форм. Как это сделать, мы уже обсуждали на первых страницах главы, для этого понадобится окно опций проекта, вызываемое из меню **Project | Options** (см. рис. 15.2).

Возвратимся к главной форме проекта и разработаем несколько ключевых процедур для нашего приложения. Но сначала подключим дочерний программный модуль child.pas к главному программному модулю main.pas. Проще всего это сделать, обратившись к услугам пункта меню среды разработки **File | Use unit...** или нажав комбинацию клавиш <Alt>+<F11>.

ВНИМАНИЕ!

Для того чтобы один программный модуль мог получить доступ к переменным, объектам, методам объектов, описанных в другом модуле, его следует подключить к списку используемых модулей с помощью `uses`.

В листинге 15.11 предложен код события `OnClick()`, которое возникает при щелчке по элементу меню **Новый**, отвечающему за создание нового пустого текстового документа.

Листинг 15.11. Создание нового пустого документа

```
procedure TfrmMain.nNewClick(Sender: TObject);
begin
  {Создание нового пустого документа}
  with TfrmChild.Create(frmMain) do //создаем новую дочернюю форму
  begin
    Caption:='Новый документ'; //заголовок дочернего окна
    Memo1.Modified:=False; //признак, что документ не модифицировался
  end;
end;
```

Листинг 15.12 хранит код щелчка по пункту меню **Открыть**. В рамках этого события мы вызываем диалог открытия файла, в котором пользователь выбирает требуемый ему текстовый файл. Затем создается новая дочерняя форма и в ее компонент `Memo1` загружается файл.

Листинг 15.12. Загрузка текстового документа из файла

```
procedure TfrmMain.nOpenClick(Sender: TObject);
begin
  {загрузка документа из файла}
  if OpenTextFileDialog1.Execute(frmMain.Handle) then
    with TfrmChild.Create(frmMain) do
    begin
      fName:=OpenTextFileDialog1.FileName;
      Caption:=ExtractFileName(fName);
      Memo1.Lines.LoadFromFile(fName);
      Memo1.Modified:=False;
    end;
end;
```

Ненадолго вернемся к теории. В классе `TForm` предусмотрен ряд свойств и методов, специализирующихся именно на приложениях с интерфейсом MDI. Например, родительское окно приложения MDI всегда обладает информацией о количестве созданных дочерних окон:

```
property MDIChildCount: Integer;
```

Доступ ко всем созданным дочерним окнам можно получить из перечня

```
property MDIChildren[I: Integer]: TForm;
```

Дочерние формы заносятся в список в порядке их создания. Если какая-то форма в процессе работы приложения уничтожается, то список самостоятельно упаковывается. Благодаря этим двум свойствам программист приобретает возможность найти необходимое дочернее окно (или окна) и произвести с ним какие-то действия (листинг 15.13).

Листинг 15.13. Перебор всех открытых дочерних окон

```
var i : integer;
begin
  for i:=0 to frmMain.MDIChildCount-1 do
  if frmMain.MDIChildren[i].Caption='Новый документ'
          then frmMain.MDIChildren[i].WindowState:=wsMinimized;
end;
```

Предложенный код позволяет среди всех дочерних окон обнаружить формы, в заголовке которых написано "Новый документ", и свернуть их.

Для того чтобы узнать, какое из дочерних окон в настоящий момент активно, достаточно обратиться к свойству

```
property ActiveMDIChild: TForm;
```

Если у родительского окна в настоящий момент нет дочерних окон (свойство `MDIChildCount=0`), то свойство `ActiveMDIChild` вернет значение `nil`. Свойство, предоставляющее ссылку на активную дочернюю форму, пригодится в листинге 15.14, в нем отражен исходный код щелчка по элементу меню **Сохранить как**.

Листинг 15.14. Сохранение документа активной дочерней формы

```
procedure TfrmMain.nSaveAsClick(Sender: TObject);
begin
  {щелчок по элементу меню "Сохранить как"}
  if frmMain.ActiveMDIChild<>nil then
    with frmMain.ActiveMDIChild as TFrmChild do
      if SaveTextFileDialog1.Execute(Handle) then
        begin
          Memo1.Lines.SaveToFile(SaveTextFileDialog1.FileName);
          fName:=SaveTextFileDialog1.FileName; //запомним новое имя
          Caption:=ExtractFileName(fName);//название файла в заголовке
        end;
end;
```

Ряд методов главного окна решает задачи, связанные с размещением и упорядочиванием дочерних окон в своей клиентской области. Для упорядочивания значков свернутых дочерних окон предназначен метод

```
procedure ArrangeIcons;
```

Эта процедура разместит свернутые значки вдоль нижней рамки главной формы.

Другой метод используется для размещения дочерних окон каскадом:

```
procedure Cascade;
```

Процедура позволяет упорядочить окна таким образом, что пользователь сможет видеть заголовки всех дочерних окон и быстро найти из них нужное.

Операционная система также позволяет разместить дочерние окна таким образом, что они не будут перекрываться:

```
procedure Tile;
```

Метод сотрудничает со свойством:

```
property TileMode: TTileMode;
type TTileMode = (tbHorizontal, tbVertical);
```

Для перемещения по дочерним формам предназначены два метода:

```
procedure Next;     //активировать следующую форму
procedure Previous; //активировать предыдущую форму
```

Названия методов говорят сами за себя: процедура `Next()` выбирает следующее окно, `Previous()` возвращается к предыдущему.

> ***ЗАМЕЧАНИЕ***
>
> У главной формы проекта MDI имеется свойство `WindowMenu`, в которое передается ссылка на один из элементов верхнего уровня главного меню приложения. С созданием каждого нового дочернего окна к этому пункту меню будет добавляться новый пункт со ссылкой на дочернее окно.

Фрейм *TFrame*

Фрейм является ближайшим родственником обычной формы. Вернувшись к рис. 15.3, вы увидите, что `TFrame` зиждется на том же самом перечне классов, что и его более именитый "коллега" — форма `TForm`. Расхождения между классами `TForm` и `TFrame` проявляются в самом конце цепочки наследования, на уровне последнего общего предка `TScrollingWinControl`. Как следствие, фрейм на 90% обладает тем же самым перечнем свойств, методов и обработчиков событий, что и форма.

Так в чем же разница между фреймом и формой? Ключевое отличие заключается в том, что фрейм никогда не сможет стать самостоятельным окном приложения. Фрейм обречен на судьбу контейнера, на котором разрешено размещать любые другие визуальные и невизуальные компоненты Delphi. Но это очень непростой контейнер, у него есть ряд очень полезных особенностей:

- контейнер на основе фрейма может содержать исходный код, в котором реализуется вся внутренняя логика фрейма;
- контейнер на основе фрейма сохраняется в отдельных файлах (с расширениями dfm и pas), что позволяет многократно использовать один и того же фрейм в разных проектах.

Фреймы часто применяются в ситуации, когда объединенная единой логикой функционирования одна и та же группа элементов управления задействуется в нескольких формах проекта. В таком случае наличие фрейма существенно упрощает и, что очень важно, ускоряет процесс разработки приложения.

Создание простого фрейма

Допустим, что мы разрабатываем программное обеспечение, автоматизирующее работу предприятия, в том числе и отдела кадров. Вполне естественно, что в приложении наиболее востребованными станут формы, позволяющие вводить персональные данные (фамилию, имя, отчество, дату рождения и т. д.) сотрудника предприятия. Не исключено, что подобные формы позднее потребуются и в других программных модулях нашей информационной

системы, например в бухгалтерии. В таком случае самым разумным решением станет использование услуг фрейма.

Создайте обычный проект **VCL Forms Application** и сохраните его под любым именем. Чтобы добавить к проекту фрейм, выберите пункт меню **File | New | Other**, в появившемся окне **New Items** раскройте папку **Delphi Files**. Среди значков найдите фрейм (**Frame**) и щелкните по кнопке **OK** (рис. 15.9). В результате к проекту будет добавлен новый пустой фрейм.

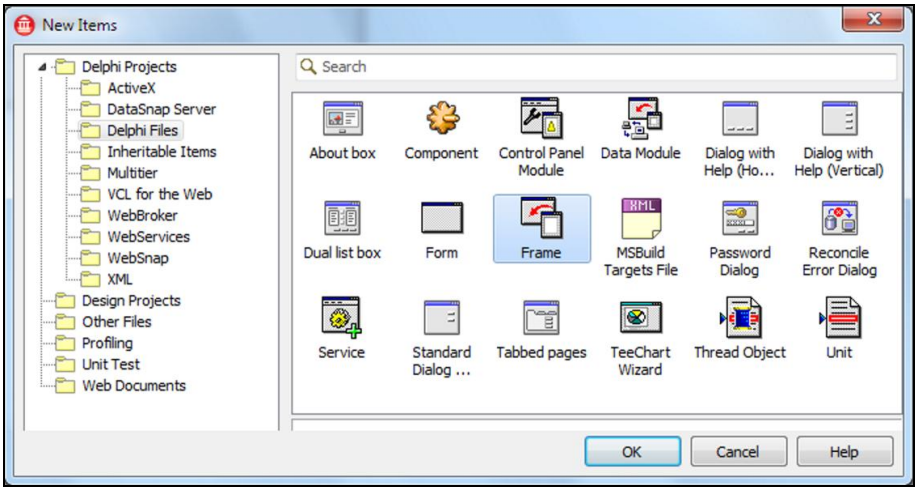


Рис. 15.9. Диалог **New Items** с активной папкой **Delphi Files**

Разместите на фрейме следующие элементы управления:

- три строки ввода `TEdit` или `TLabeledEdit`. Они служат для ввода фамилии (`Name:=edSName`), имени (`Name:=edFName`) и отчества (`Name:=edLName`);
- строку ввода с маской `TMaskEdit`, предназначенную для ввода даты рождения (`Name:=edBirthday`). Настройте шаблон ввода даты, для этого передайте в свойство `EditMask` маску `'!99/99/0000;1;_ '`;
- группу переключателей `TradioGroup`, позволяющую указать уровень образования сотрудника. Переименуйте компонент в `rgEducation` и в свойство `Items` введите два варианта ответа: среднее и высшее образование.

Сохраните фрейм под именем `FrameStep1`, а соответствующий ему программный модуль переименуйте в `Step1`.

Еще не принадлежащий ни одной из форм проекта фрейм уже имеет возможность создавать свою программную логику. Допустим, нам надо обладать механизмом, позволяющим убедиться, что все поля фрейма (фамилия, имя и т. д.) заполнены данными. Эта услуга может оказаться крайне полезной в приложениях баз данных, когда поля таблиц (в которые передаются данные из формы ввода) не допускают неопределенных значений.

Для реализации механизма проверки заполнения полей в секции `public` фрейма объявляем функцию `DataComplite()`, возвращающую значение `true`, если все необходимые данные введены (листинг 15.15).

Листинг 15.15. Исходный код фрейма Step1

```
unit Step1;
interface
uses  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
      Forms, Dialogs, StdCtrls, ExtCtrls, Mask;

type
  TFrameStep1 = class(TFrame)
    edSName: TLabeledEdit;
    edFName: TLabeledEdit;
    edLName: TLabeledEdit;
    rgEducation: TRadioGroup;
    Label1: TLabel;
    edBirthday: TMaskEdit;
  public
    Function DataComplite:boolean;
  end;

implementation
{$R *.dfm}
function TFrameStep1.DataComplite: boolean;
begin
  {функция возвратит true, если заполнены все необходимые данные}
  Result:=(Trim(edSName.Text)<>'') and //введена фамилия
          (Trim(edFName.Text)<>'') and //введено имя
          (Trim(edLName.Text)<>'') and //введено отчество
          (StrToDateDef(edBirthday.Text,0)<>0) and //введена дата рождения
          (rgEducation.ItemIndex<>-1); //выбрано образование
end;
end.
```

Внимание!

Для того чтобы расположить уже готовый фрейм на форме, следует воспользоваться компонентом `Frames`. Этот компонент вы найдете на странице **Standard** палитры компонентов Delphi.

Диалоги-помощники на основе фреймов

Фреймы крайне полезны при разработке интерфейсной части различного рода диалогов-помощников, собирающих необходимые данные или поэтапно ведущих пользователя к определенному решению. В таких диалогах зачастую реализуются сложные разветвленные алгоритмы выбора, а внешний вид и набор элементов ввода данных очередного окна диалога зависит от полученных от пользователя ответов на предыдущем этапе опроса.

Примером интеллектуального диалога ввода данных может стать проект приложения для отдела кадров. Подлежащие вводу персональные данные сильно зависят от многочисленных индивидуальных характеристик сотрудников. Если сотрудник — мужчина, то в базу данных отдела кадров необходимо передать сведения о прохождении службы в армии, а для женщины эти данные необязательны. Многочисленные ветвления предполагают пункты

анкеты: семейное положение, наличие детей, образование, специальность, наличие ученых степеней и званий, правительственные награды и т. п.

Попробуем продемонстрировать идею построения подобных диалогов-помощников на основе фреймов. Разработанный в листинге 15.15 фрейм (мы его предусмотрительно назвали фреймом первого шага `FrameStep1`) в числе прочих данных предлагает указать образование сотрудника (среднее или высшее). Вполне естественно, что от выбранного уровня образования зависят вводимые далее данные. Если сотрудник имеет среднее образование, то отделу кадров стоит знать название учебного заведения и дату получения аттестата. Если сотрудник имеет высшее образование, то стоит подумать о сборе данных: о названии вуза, полученной специальности, дате выдачи и номере диплома.

Для ввода данных об образовании на втором шаге работы диалога-помощника нам понадобятся еще два фрейма. Если пользователь получил высшее образование, то будет вызван фрейм `FrameStep2HighEdu`, а для ввода данных о среднем образовании предназначен `FrameStep2SecEdu`.

Кроме того, нам нужна обычная форма (назовем ее `frmWizard`), в нижней части которой следует разместить три кнопки `TButton`. Две кнопки `btnPrior` (`Caption:= '<<Назад'`) и `btnNext` (`Caption:= 'Далее>>'`) возьмут на себя ответственность за переход от фрейма к фрейму. Кнопка `btnCancel` (`Caption:='Отмена'`) позволит прервать работу мастера ввода данных.

Подключите к строке `uses` формы `frmWizard` программные модули фреймов, а в секции частных объявлений `private` формы введите (листинг 15.16):

- объектные переменные всех трех фреймов;
- переменную `StepNum`, в которой мы станем хранить номер шага;
- основную процедуру `ChangeFrame()`, которая возьмет на себя ответственность за смену фрейма в соответствии с номером шага.

Листинг 15.16. Секция частных объявлений формы `frmWizard`

```
unit wizard; //модуль формы frmWizard
interface

uses Windows, ...,
     Step1, Step2SecEdu, Step2HighEdu; {подключим модули фреймов}
type
  TfrmWizard = class(TForm)
//...
private
  FrameStep1:TFrameStep1;                {фрейм шага 1 ФИО}
  FrameStep2SecEdu:TFrameStep2SecEdu;    {фрейм шага 2 среднее образование}
  FrameStep2HighEdu:TFrameStep2HighEdu;{фрейм шага 2 высшее образование}
  StepNum:byte;                          {счетчик шагов}
  procedure ChangeFrame(const Step:byte); {метод смены фрейма}
```

Функциональное назначение метода смены фрейма раскрыто в листинге 15.17. На вход метода поступает константа номера шага. На первом шаге создается первый фрейм, на втором — фрейм среднего или высшего образования.

Листинг 15.17. Фрагмент метода смены фреймов

```
procedure TfrmWizard.ChangeFrame(const Step : byte);
begin
  case Step of     //номер шага
    1:begin {1 шаг: фрейм ввода ФИО, даты рождения и образования}
        if FrameStep1=nil then {если фрейм 1 шага пока не существует}
          begin {создание и размещение 1-го фрейма на форме}
            FrameStep1:=TFrameStep1.Create(frmWizard);
            FrameStep1.Parent:=frmWizard;
            FrameStep1.Left:=100;
            FrameStep1.top:=0;
          end;
        //...
        FrameStep1.Show; {отображение фрейма}
      end;
    2:begin {2 шаг: создаем фрейм среднего или высшего образования}
        FrameStep1.Hide; {скрываем фрейм 1 шага}
        //...
        {код выбранного образования хранится в rgEducation.ItemIndex}
        case FrameStep1.rgEducation.ItemIndex of
          0: begin //фрейм 2 шага — среднее образование
               if FrameStep2SecEdu=nil then {если фрейм отсутствует}
               begin {создаем фрейм}
                 FrameStep2SecEdu:=TFrameStep2SecEdu.Create(frmWizard);
                 //...
               end;
               FrameStep2SecEdu.Show; {отображение фрейма}
             end;
          1: begin //фрейм 2 шага — высшее образование
               if FrameStep2HighEdu=nil then{если фрейм отсутствует}
               begin {создаем фрейм}
                 FrameStep2HighEdu:=TFrameStep2HighEdu.Create(frmWizard);
                 //...
               end;
               FrameStep2HighEdu.Show; {отображение фрейма}
             end;
        end;
      end;
    3:begin
        {3 шаг}
      end;
    //...
```

В момент вывода формы на экран устанавливаем начальное значение шага и вызываем метод смены фреймов (листинг 15.18).

Листинг 15.18. Первый ввод формы диалога `frmWizard` на экран

```
procedure TfrmWizard.FormShow(Sender: TObject);
begin
  StepNum:=1;
  btnPrior.Enabled:=false;
  ChangeFrame(StepNum);
end;
```

Нам осталось предусмотреть вызов метода смены фреймов по щелчку на кнопках `btnNext` и `btnPrior`. При этом надо не забывать увеличивать (или уменьшать) номер шага. В результате мы получим удобный диалог-помощник, существенно упрощающий ввод данных (рис. 15.10).

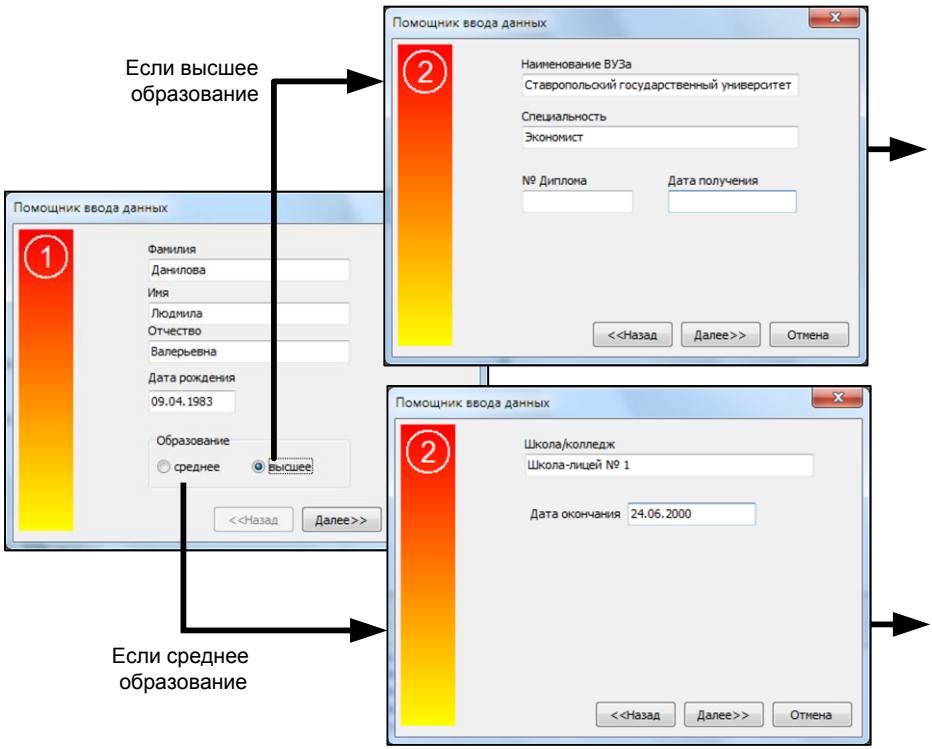


Рис. 15.10. Помощник ввода данных на основе фреймов

Модуль данных *TDataModule*

Многие приложения Delphi помимо обычных визуальных элементов управления пользуются услугами многочисленного семейства невизуальных компонентов. На их основе строятся проекты баз данных, многоуровневые приложения DataSnap, сетевые проекты Indy, проекты Internet и Web. И хотя после запуска приложения на выполнение невизуальные компоненты исчезают с поверхности форм, оставляя занимаемое пространство для обычных эле-

ментов управления, во время визуального проектирования дела обстоят иначе — невизуальные компоненты загромождают формы, мешая программисту формировать пользовательский интерфейс приложения.

Именно для решения проблемы размещения невизуальных компонентов в составе Delphi имеется специальный контейнер, называемый *модулем данных*. Модуль данных создается на основе класса `TDataModule`, который в свою очередь выступает наследником класса `TComponent` (рис. 15.3).

Число модулей данных в проекте не ограничено. Для создания нового модуля данных и подключения его к проекту следует (обратившись к пункту меню **File | New | Other**) вызвать диалог **New Items**. В появившемся окне диалога следует выбрать папку **Delphi Files** и найти значок **Data Module** (см. рис. 15.9).

После подключения модуля данных к проекту мы получим удобный контейнер для размещения невизуальных компонентов (рис. 15.11).

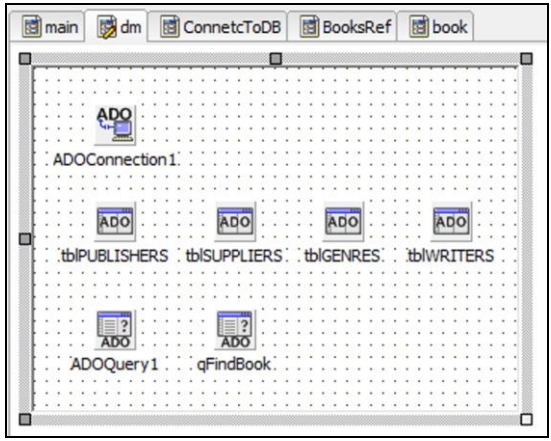


Рис. 15.11. Модуль данных с невизуальными компонентами со страницы **dbGo**

ЗАМЕЧАНИЕ

По умолчанию все добавленные к проекту модули данных создаются автоматически во время запуска приложения. Для отказа от создания модуля следует воспользоваться пунктом меню **Project | Options**, в появившемся окне настройки опций проекта выбрать вкладку **Forms** и убрать модуль данных из списка **Auto-create forms**.

Свойств и методов у модуля данных совсем немного, и весь их перечень нам уже знаком. Упомянем лишь два обработчика события

```
property OnCreate: TNotifyEvent;  //создание модуля данных
property OnDestroy: TNotifyEvent; //разрушение модуля данных
```

которые возникают соответственно в момент создания и в момент уничтожения экземпляра модуля данных.

ГЛАВА 16

Исключительные ситуации

Хотите — верьте, хотите — нет, но в природе очень мало программных продуктов, в которых полностью отсутствуют ошибки. Крупнейшие корпорации, разрабатывающие программное обеспечение, тратят много времени и огромные средства на тестирование и отладку своих творений, в ходе которых выявляются многочисленные ошибки и ошибочки. Устранение старых ошибок приводит к появлению новых, круг замыкается. Довольны только альфа- и бета-тестеры — они всегда при деле...

Хотя в 9 случаях из 10 причина ошибок кроется в недостаточно скрупулезной проработке программистом кода, существуют и другие источники неприятностей. Среди них: огрехи в операционной системе; недостатки программных продуктов сторонних производителей, с которыми взаимодействует ваше приложение; сбои в работе оборудования; бреши в системе безопасности (вашей программкой с удовольствием может полакомиться вирус); ошибки администрирования; и, конечно же, удивительные по своей абсурдности действия особой категории пользователей. Говоря о них, на ум сразу приходит один из законов Мерфи для программистов: "Защиту от дураков создать невозможно, т. к. они крайне изобретательны".

Раз ошибки — это неизбежность, то что им можно противопоставить? В событийно-ориентированном языке программирования Delphi возникновение ошибки прерывает нормальный ход выполнения приложения, ошибка выполнения *(runtime error)* перехватывается и автоматически конвертируется в так называемую *исключительную ситуацию* (ИС, exception). Дальнейшее поведение приложения определяется уровнем исключительной ситуации: незначительную ошибку можно просто проигнорировать, а более существенные сбои в работе способны привести к полной остановке программы. Раз ошибка служит источником события, то мы получаем шанс отреагировать на это событие и, по возможности, нивелировать последствия. Для этих целей разработчики Delphi создали весьма эффективный механизм обработки ИС, одно из ключевых достоинств которого заключается в том, что на стороне программиста стоит вся мощь парадигмы объектно-ориентированного программирования. Поэтому уже на уровне модуля `SysUtils` объявлены опорные классы исключительных ситуаций и ряд методов, осуществляющих помощь в обработке ошибок.

Защищенные от ошибок секции

Еще со времен Object Pascal программисту предлагался ряд специальных кодовых конструкций, которые если не избавляют нас от возникновения ошибок, то позволяют корректным образом их обработать. В первую очередь это конструкции, создающие защищенные от ошибок секции: `try..except` и `try..finally`.

Конструкция *try..except*

Конструкция `try..except` позволяет программисту создать специальный участок кода, который подлежит выполнению только в случае возникновения исключительной ситуации, а в штатном режиме работы программы этот код просто пропускается.

Программная конструкция `try..except` состоит из двух секций:

- защищенной секции, содержащей потенциально опасную часть кода, в котором высока вероятность возникновения исключительной ситуации;
- секции обработки исключительной ситуации — части кода, выполняющейся только в случае возникновения ошибки в защищенной секции.

С точки зрения синтаксиса конструкция выглядит следующим образом:

```
try
  {операторы защищенной секции}
except
  {операторы секции обработки ИС}
end;
```

Рассмотрим пример (листинг 16.1), разъясняющий порядок применения конструкции `try..except`. Допустим, что мы работаем с динамическим массивом `A` и по рассеянности забыли распределить под него память. Вполне понятно, что любая попытка чтения данных из несуществующего массива обречена на провал. Небольшим утешением в сложившейся ситуации станет сообщение об ошибке.

Листинг 16.1. Обработка ИС в секции `try..except`

```
var A:Array of integer;
    x:integer;
begin
  try
    //SetLength(A,10); мы "забыли" распределить память под массив
    x:=A[0]; //попытка прочитать данные из массива вызовет ИС
  except
    raise Exception.Create('Ошибка обращения к массиву!');
  end;
end;
```

Конструкция *try..finally*

Конструкция `try..finally` также предназначена для обработки исключительных ситуаций, но порядок ее применения принципиально отличается от работы с ее соратником по борьбе с ошибками — конструкцией `try..except`. Программный блок `try..finally` вновь содержит две секции. Потенциально опасный код, выполнение которого способно привести к ошибке, заносится в *защищенную секцию*, начинающуюся с ключевого слова `try`. Но на этот раз вторая секция, называемая *секцией завершения*, содержит гарантированно выполняемый код. Другими словами, чтобы не стряслось в рамках защищенной секции, операторы секции `finally` выполнятся в любом случае (конечно при условии, что в компьютере не расплавится процессор).

```
try
  {операторы защищенной секции}
finally
  {операторы секции завершения}
end;
```

Порядок применения конструкции в проектах Delphi отражает листинг 16.2.

Листинг 16.2. Обработка ИС в секции `try..finally`

```
var Bitmap : TBitmap;
begin
  Bitmap :=TBitmap.Create; //создание экземпляра TBitmap
  try
    Bitmap.LoadFromFile('... путь к файлу...');
    Form1.Canvas.Draw(0,0, Bitmap);
  finally
    Bitmap.Free; //гарантированное освобождение экземпляра TBitmap
  end;
end;
```

После создания объекта `Bitmap` в этот объект загружается растровый образ из файла и выводится на поверхности холста. Нельзя исключить вероятности возникновения ошибки в момент загрузки картинки в объект, например, из-за ошибки в имени файла. Однако независимо от течения процесса созданный нами объект `Bitmap` будет гарантированно уничтожен в секции `finally`.

> ***ЗАМЕЧАНИЕ***
>
> Если быть принципиальным, то стоит отметить, что конструкция `try..finally` не предназначена для ликвидации исключительной ситуации. Она только лишь способна сгладить ее последствия, выполнив операции из секции завершения.

Вложенные конструкции *try*

Язык Delphi допускает взаимное вложение конструкций `try..except` и `try..finally` друг в друга. Вложение конструкции `try..finally` вовнутрь защищенной секции конструкции `try..except` позволяет гарантированно выполнить группу операторов из секции `finally`, и в случае возникновения исключительной ситуации выполняются операторы из секции `except`. Также жизнеспособно и обратное решение — вложение конструкции `try..except` в защищаемую секцию `try..finally`.

```
try
  try
    {группа защищаемых операторов}
  finally
    {всегда выполняемая секция
     завершения}
  end;
except
  {операторы, выполняющиеся
   только при возникновении ИС}
end;
```

```
try
  try
    {группа защищаемых операторов}
  except
    {обработка ИС}

  end;
finally
  {секция гарантированно
   выполняемых операторов}
end;
```

Листинг 16.3 содержит код, сохраняющий динамический массив целых чисел `Ar` в файл с именем `FileName`. В примере предложен вариант применения вложенных защитных конструкций.

Листинг 16.3. Пример вложенных конструкций `try`

```
procedure SaveArrayToFile(AR: array of integer; FileName: string);
var F : file of integer;
    i : integer;
begin
  try
    AssignFile(F,FileName); //связывание файловой переменной F с файлом
    FileMode:=1;            //открыли только для записи
    Rewrite(F);
    try
      for i:=Low(Ar) to High(Ar)
        do Write(F, ar[i]); //сохраняем массив в файл
    finally
      CloseFile(F); //Закрытие файла и освобождение файловой переменной
    end;
  except
    {обработка исключительной ситуации}
    raise Exception.Create('Ошибка!');
  end;
end;
```

Вложенная конструкция `try..finally` предназначена для безоговорочного освобождения дескриптора файла `F`. Но в силу того, что `try..finally` не умеет обрабатывать ИС, эта почетная обязанность возложена на конструкцию `try..except`.

Объектная модель исключительных ситуаций

С точки зрения Delphi исключительная ситуация — это не что иное, как объект, поэтому подход к обработке ИС опирается на законы объектно-ориентированного программирования. Таким образом, исключительная ситуация полноправно пользуется всеми преимуществами ООП: абстрагированием, инкапсуляцией, наследованием и полиморфизмом. Опорой и отправной точкой для объектов ИС служит класс `Exception` (модуль `SysUtils`). Как показано на схеме (рис. 16.1), единственный предок `Exception` — наш старый знакомый `TObject`. Потомков же его не счесть. Кстати, на схеме представлены только наиболее влиятельные наследники опорного класса ИС.

> ***ЗАМЕЧАНИЕ***
>
> Чтобы выделить объекты исключительных ситуаций из общей канвы объектов VCL, разработчики Delphi пришли к следующему соглашению: аббревиатура названия класса ИС начинается не с классического для классов символа "T" (type), а с префикса "E" (exception). Впрочем, опорный класс иерархии ИС так и называется `Exception`, без всяких там префиксов.

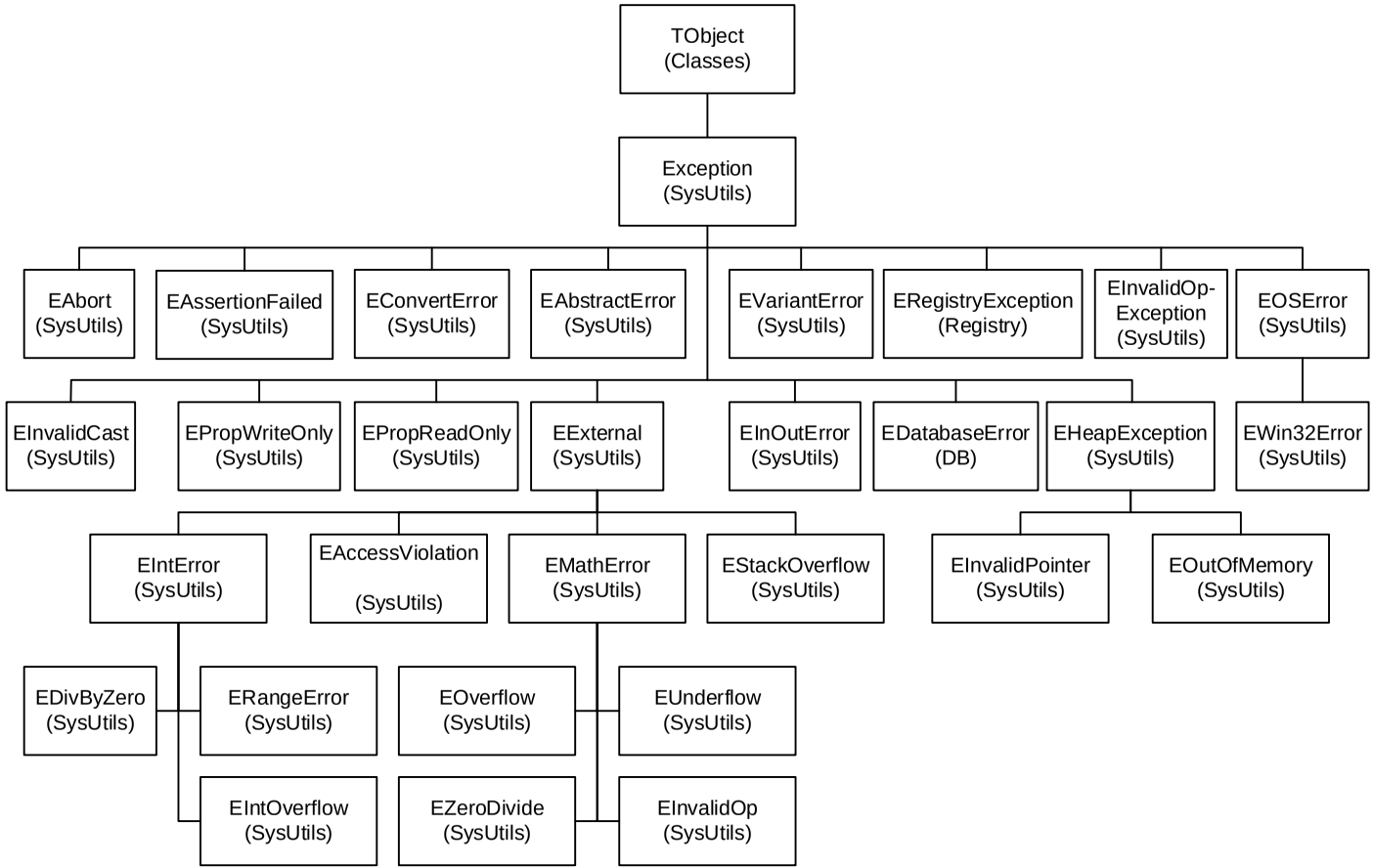


Рис. 16.1. Иерархия объектов исключительных ситуаций в VCL

Назначение основополагающих классов ИС предложено в табл. 16.1.

Таблица 16.1. *Описание основных классов исключительных ситуаций*

Классы исключительных ситуаций	Описание
`Exception`	Опорный класс для всех ИС
`EAbort`	Исключительная ситуация, используемая программистами для незаметной обработки ошибки
`EAbstractError`	ИС инициируется при попытке обратиться к абстрактному методу
`EAssertionFailed`	ИС разработчика. Она способна указать имя модуля и номер строки, вызвавших ИС. Используется только во время отладки приложения
`EConvertError`	ИС, генерируемая при ошибке конвертирования целых и действительных чисел, даты и времени в строку, при ошибке обратных преобразований, при ошибках в аргументах различных функций форматирования, при ошибочном связывании объектов несоответствующих типов
`EExternal`	Класс, осуществляющий взаимодействие с операционной системой и способный собирать данные об ИС из заимствованной у Win32 структуры. Для этого объявлена специализированная запись `TExceptionRecord`. Класс является базовым для целой ветви ИС
`EHeapException`	ИС для ошибок, связанных с памятью и хранением в ней данных. В частности, этот класс служит предком для таких ИС, как `EOutOfMemory` (нет памяти) и `EInvalidPointer` (ошибка указателя)
`EInOutError`	ИС операций ввода/вывода
`EInvalidCast`	Ошибка приведения типов. Происходит при попытке приведения несовместимых типов данных
`EOSError`	Класс исключительных ситуаций операционной системы
`EPackageError`	ИС, появление которой возможно только в период разработки. Сигнализирует об ошибке при загрузке или утилизации пакета
`EPropReadOnly`	ИС, возникающая при попытке записи в поле, доступное только для чтения
`EPropWriteOnly`	ИС, возникающая при попытке чтения из поля, доступного только для записи
`ESafecallException`	ИС нарушения соглашения о вызове `safecall`. Этот вид вызова применяется при обращении к методам Win32 API и к объектам автоматизации
`EVariantError`	Ошибка с данными `Variant`. Класс служит предтечей целого перечня ИС, обслуживающих операции с этим типом данных

Базовый класс *Exception* исключительной ситуации

Традиционно для ООП создание экземпляра исключительной ситуации начинается с обращения к его конструктору. Разработчики класса `Exception` потрудились на славу, реализовав дюжину различных вариаций конструкторов (табл. 16.2).

Таблица 16.2. *Конструкторы* `Exception`

Конструктор	Описание
`constructor Create (const Msg: string);`	Простейший вариант конструктора, где `Msg` — текст неформатированное сообщение об ошибке
`constructor CreateFmt (const Msg: string; const Args: array of const);`	Конструктор, сообщение которого форматируется при помощи функции `Format()`
`constructor CreateHelp (const Msg: string; AHelpContext: Integer);`	Неформатированное сообщение плюс идентификатор контекстной справки
`constructor CreateFmtHelp (const Msg: string; const Args: array of const; AHelpContext: Integer);`	Все в одном флаконе. Конструктор сразу форматирует текстовое сообщение и присваивает идентификатор контекстной справки
`constructor CreateRes (Ident: Integer); overload;` `constructor CreateRes(ResStringRec: PResStringRec); overload;`	Создается экземпляр исключительной ситуации, текстовое сообщение загружается из ресурса. Допускаются два варианта указания на ресурс: идентификатор ресурса `Ident` или имя ресурса `ResStringRec`
`constructor CreateResFmt(Ident: Integer; const Args: array of const); overload;` `constructor CreateResFmt(ResStringRec: PResStringRec; const Args: array of const); overload;`	Симбиоз методов `CreateRes()` и `CreateFmt()`. Текст сообщения хранится в файле ресурса и форматируется данными из массива `Args`
`constructor CreateResHelp(Ident: Integer; AHelpContext: Integer); overload;` `constructor CreateResHelp(ResStringRec: PResStringRec; AHelpContext: Integer); overload;`	Объединение возможностей методов `CreateRes()` и `CreateHelp()`. Создание ИС с загрузкой текстового сообщения из ресурса с одновременной установкой идентификатора контекстно-зависимой страницы справки
`constructor CreateResFmtHelp(Ident: Integer; const Args: array of const; AHelpContext: Integer); overload;` `constructor CreateResFmtHelp(ResStringRec: PResStringRec; const Args: array of const; AHelpContext: Integer); overload;`	Симбиоз методов `CreateRes()` и `CreateFmtHelp()`. Текст хранится в ресурсе, массив аргументов и идентификатор страницы справки передаются как внешние параметры

В простейшем случае нам будет достаточно услуг простейшего из перечня конструкторов. Допустим, что в нашем проекте следует предусмотреть генерации ИС в случае попытки доступа к несуществующему файлу. Наиболее распространенное решение этой задачи представлено в листинге 16.4.

Листинг 16.4. Обработка ИС в секции `try..except`

```
if FileExists(FileName)=True then
   begin
    {операции с файлом}
   end else
     raise Exception.Create('Файл '+FileName +' не найден!');
```

Обратите внимание на применение в листинге 16.4 зарезервированного слова `raise`, оно должно предшествовать вызову конструктора ИС.

ВНИМАНИЕ!

Хотя все классы ИС обладают деструктором `Destroy()`, в его вызове нет необходимости, т. к. экземпляр исключительной ситуации уничтожается автоматически.

Хотя Delphi не возражает против появления на свет рукотворных (как в листинге 16.4) исключительных ситуаций, но в подавляющем большинстве случаев ИС генерируется вне зависимости от пожеланий программиста.

Тихая исключительная ситуация *EAbort*

Основное отличие исключительной ситуации `EAbort` от всех ее "коллег" в том, что это единственный класс, который по умолчанию не предусматривает вывода на экран сообщения о возникновении ИС. И вообще `EAbort` не любит шуметь и выносить сор из избы, именно за эти "исключительные" заслуги этот класс назвали *тихой исключительной ситуацией* (silent exception). Хотя класс обладает конструктором для вызова этой исключительной ситуации, целесообразнее применять объявленную в модуле `SysUtils` процедуру

```
procedure Abort;
```

Рассмотрим пример, в котором в рамках цикла с помощью гипотетической процедуры `ChangeObjectState()` пытаются изменить состояние какого-то объекта (листинг 16.5).

Листинг 16.5. Пример работы с `EAbort`

```
i:=0; //обнуляем счетчик итераций цикла
try
  while ControlObjectState=True do //продолжаем цикл "пока"
    begin //ControlObjectState=True}
      ChangeObjectState(); //попытка изменить состояние объекта
      INC(i);              //даем приращение счетчику
      {если осуществлена 1 тыс. безуспешных попыток повлиять
       на объект функцией ChangeObjectState(), то прервем цикл
       "тихой" ИС}
      if i>1000 then abort;
    end;
except
  on EAbort do{если цикл прерван ИС типа EAbort}
    begin
      MessageBeep(0);
      {остальные операторы обработки этой ИС}
    end
              //если же причина ИС — не EAbort, то
  else raise; //вызываем настоящую "громкую" ИС
end;
```

Цикл продолжается до тех пор, пока метод `ControlObjectState` возвращает значение `true`. Для того чтобы цикл не выполнялся вечно, контролируем состояние счетчика итераций `i`. При превышении этим счетчиком значения 1000 вызываем ИС типа `EAbort` и прерываем цикл, причем никто этого и не заметит, если убрать вызов звукового сигнала `MessageBeep()`.

Исключительная ситуация отладки *EAssertionFailed*

Класс исключительной ситуации `EAssertionFailed` и соответствующий ей метод `Assert()` применяются только во время отладки приложения. Особенность этой ИС в том, что она вместо абсолютно безликого кода ошибки умеет информировать программиста об имени модуля и номере строки в этом модуле, послуживших причиной для вызова ИС.

В простейшем случае процедура `Assert()` требует передачи одного-единственного параметра `expr`.

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

Если значение этого параметра равно `false`, метод инициирует создание ИС `EAssertionFailed`. Второй параметр может содержать комментарий программиста к ошибке.

В листинге 16.6 предложен код, демонстрирующий способ применения процедуры `Assert()`. Это небольшое консольное приложение с единственной процедурой `AddStringToList()`, добавляющей в список `List:TStringList` новую строку `S`. Для удобства строки программы пронумерованы.

Листинг 16.6. Отладка проекта с помощью ИС `EAssertionFailed`

```
{01} program assert_demo;
{02}
{03}{$APPTYPE CONSOLE}
{04}{$ASSERTIONS ON} //директива включения проверки ASSERT
{05}
{06}uses SysUtils, Classes;
{07}
{08}procedure AddStringToList(List : TStringList; const S : string);
{09}begin
{10}  Assert(Assigned(List) and (List is TStringList),'Ошибка!');
{11}  List.Add(S);
{12}end;
{13}
{14}var List: TStringList;
{15}begin
{16}
{17} try
{18}   List:=TStringList.Create;
{19}   AddStringToList(List,'NewLine 1');
{20}   AddStringToList(List,'NewLine 2');
{21}   AddStringToList(nil, 'NewLine 3'); //закладываем ошибку
{22} finally
{23}   List.Free;
{24} end;
{25}end.
```

Если вдруг, по каким-то причинам, первым параметром метода `AdStringToList()` передается неверная ссылка на экземпляр `TStringList`, то это приведет к возникновению ИС. Предвидя такое развитие событий, в 10-й строке листинга программист применяет метод `Assert()`, который контролирует значение параметра `List`. Благодаря этой процедуре, если

параметр пуст или не является классом TStringList, во время запуска приложения (под управлением отладчика Delphi) появится информационное окно отладчика (рис. 16.2) с адресом ошибки — 10-я строка модуля assert_demo.dpr.

Рис. 16.2. Сообщение отладчика об ИС с номером строки вызвавшей ошибку

ВНИМАНИЕ!

Компилятор Delphi позволяет централизованно включать или отключать реакцию на ИС EAssertionFailed, генерируемую методом Assert(). Для включения контроля за ИС этого типа используйте директиву {$ASSERTIONS ON} или {$C+}, для отключения — {$ASSERTIONS OFF} или {$C-}.

Определение новых классов ИС

Открытость конструкций исключительных ситуаций наделяет программиста правом объявлять собственные классы ИС. Разработчик приобретает возможность вносить свои характеристики и описания, присущие только его исключительной ситуации, и затем, при стечении необходимых условий, генерировать объявленный класс ИС с помощью команды Raise. Такой подход позволит нам обеспечить единый стиль обработки ошибок в приложении.

Допустим, что мы намерены вызывать исключительную ситуацию при вводе неверного имени или пароля пользователя (листинг 16.7). Эта задача решается в два этапа:

1. Объявляем новый класс ИС.
2. Вызываем исключительную ситуацию заданного класса при ошибке регистрации пользователя.

Листинг 16.7. Создание нового класса ИС

```
type ELoginInvalide=class(Exception); //объявление типа ИС
...
implementation
uses DBLogDlg;
...
procedure DatabaseConnect(aUserName, aPassword: string);
begin
  if LoginDialog('MyDatabase', aUserName, aPassword)=true then
    begin
      if ConnectedFailed(aUserName, aPassword)=true then
      raise ELoginInvalide.Create('Ошибка доступа'); //генерация ИС
    end;
end;
```

Расширенные возможности *try..except*

Появление в Delphi объектов ИС, специализирующихся на различных типах ошибок, значительно расширило возможности конструкции `try..except`. Теперь в рамках единой конструкции защиты от ИС программист способен перехватывать конкретные типы ИС и обрабатывать их по отдельности. Взгляните на расширенный синтаксис `try..except`.

```
try
  {операторы защищенной секции}
except
   on <ТИП ОШИБКИ 1> do <обработка ИС ТИП ОШИБКИ 1>;
   on <ТИП ОШИБКИ 2> do <обработка ИС ТИП ОШИБКИ 2>;
    . . .
   else <обработка неучтенных ИС>;
end;
```

Следующие за ключевым словом операторы `on..do` применяются для перехвата исключительной ситуации заданного типа. Код, следующий за ключевым словом `else`, вызывается, только если произойдет ИС, не учтенная в операторах `on..do`.

Предложенный в листинге 16.8 пример демонстрирует вариант использования операторов `on..do` в функции деления двух чисел. Делитель передается через указатель `pY`, а делимое — через указатель `pX`. В случае успешного выполнения метод помещает результат вычисления в формальный параметр `Itog` и возвращает значение `true`. В секции `except` предусматривается обработка трех наиболее вероятных ошибок: ошибки доступа к памяти, ошибки неверного указателя и попытки деления на ноль.

Листинг 16.8. Тип ИС определяет порядок обработки ошибки

```
function MyDivision(pX,pY : Pointer; var Itog : real): boolean;
begin
  Result:=False;
  try
    Itog:=Real(pX^)/Real(pY^);
    Result:=True;
  except
    on EAccessViolation do
         MessageDlg('Ошибка доступа к памяти',mtError,[mbOK],0);
    on EInvalidPointer do
         MessageDlg('Неверный указатель',mtError,[mbOK],0);
    on EZeroDivide do
         MessageDlg('Деление на ноль',mtError,[mbOK],0);
    on EInvalidOp do
         MessageDlg('Неверная операция с плавающей точкой',
                    mtError,[mbOK],0);
    else raise Exception.Create('Неизвестная ошибка!');
  end;
end;
```

Централизованная обработка ошибок в приложении

К сожалению, далеко не всегда заранее можно предусмотреть все возможные варианты ошибок в разрабатываемом приложении. В таком случае, для "отлова" незапланированных ИС стоит воспользоваться событием `OnException` приложения `TApplication` или его ближайшего помощника — компонента `TApplicationEvents`.

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

Здесь `Sender` — приложение, в котором возникла ИС, `E` — экземпляр исключительной ситуации. Так как в этот обработчик события будут стекаться практически все произошедшие в приложении ошибки, и эти ошибки могут быть абсолютно разными, то `OnException()` обычно не специализируется на обработке частного вида ошибок. Наоборот, в его рамках описывается общая концепция защиты программного продукта.

Листинг 16.9 демонстрирует одну из полезных сторон события `OnException()`, благодаря которому мы с вами сможем вести протокол исключительных ситуаций, возникших в приложении.

Листинг 16.9. Ведение протокола ИС в событии `OnException`

```
procedure TfrmMain.ApplicationEvents1Exception(Sender: TObject;
                                                E: Exception);
const FileName='errors.log'; //файл протокола
var   UserName : array[0..24] of char;//имя пользователя
      nSize : DWORD;
begin
  if NOT (E is EAbort) then //если это не "тихая" ИС, то ...
    begin
      nSize:=SizeOf(UserName);
      GetUserName(UserName,nSize); //узнаем имя пользователя
      with TStringList.Create do
        begin
          //загрузка файла протокола
          if FileExists(FileName) then LoadFromFile(FileName);
            //формируем строку описания ИС и добавляем в протокол
            Add(Format('%s'+#9+'%s'+#9+'%s',
                [DateTimeToStr(Now),StrPas(UserName),E.Message]));
            SaveToFile(FileName);//сохраняем файл
            Free;           //освобождаем список
        end;
      Application.ShowException(E);//информируем пользователя об ИС
    end;
end;
```

В момент возникновения события `OnException()` мы убеждаемся, что ИС не относится к классу `EAbort`. Затем собираем сведения о дате/времени возникновения ошибки, имени пользователя и собственно ИС и добавляем собранную информацию в файл протокола.

Позднее протокол поможет разработчику приложения выявить причину ИС и доработать свой проект.

Настройка поведения Delphi при обработке исключительных ситуаций

По умолчанию интегрированная среда разработки Delphi настроена на перехват подавляющего большинства исключительных ситуаций и информирования о них программиста. Иногда такое дружелюбное поведение IDE становится навязчивым и мешает нам реализовать собственную обработку ИС. Поэтому при необходимости Delphi допускает отключение автоматического отлова исключительных ситуаций. Для этого требуется открыть окно **Options** (пункт меню **Tools | Options**) и в дереве опций выбрать ветвь **Debugger Options | Embarcadero Debuggers | Language Exceptions** (рис. 16.3). Если флажок **Notify on language exceptions** установлен (состояние по умолчанию), отладчик остановит выполнение программы при возникновении ИС. Для отказа от этой услуги снимите "галочку".

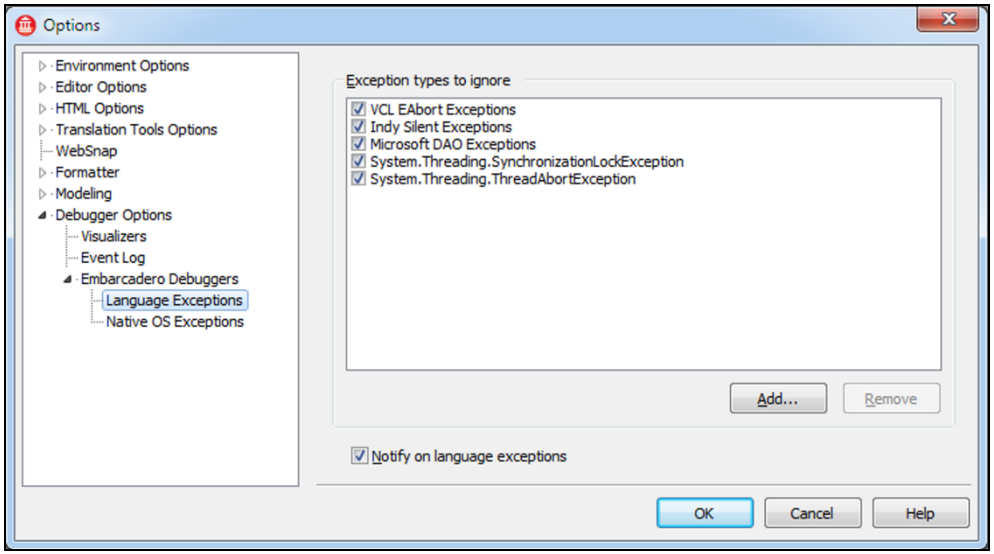


Рис. 16.3. Настройка реакции отладчика Delphi на появление ИС

Еще более тонкая настройка поведения отладчика осуществляется с помощью списка игнорируемых типов исключительных ситуаций (**Exception types to ignore**). Допустим, что мы только лишь хотим отказаться от помощи отладчика при обработке ошибок конвертирования типов данных, а во всем остальном его поведение нас вполне устраивает. В таком случае щелкаем по кнопке **Add...** и в появившейся строке ввода указываем название класса ИС, контролирующего процесс конвертирования — `EConvertError`. Затем нажимаем кнопку **OK**. После этой операции список игнорируемых типов ИС пополняется классом `EConvertError`. И пока флажок в левой части строки отмечен, отладчик будет игнорировать все ошибки, связанные с этим типом ИС.

ГЛАВА 17

Компоненты отображения и редактирования текста

Сложно привести пример современного приложения, которое было бы способно обойтись без услуг компонентов отображения и редактирования текстовых данных, ведь область применения текстовых данных практически необъятна. Она начинается с простейших окон регистрации пользователя и форм ввода и заканчивается сложными текстовыми редакторами.

Библиотека визуальных компонентов Delphi предоставляет широчайший набор компонентов, специализирующихся на отображении и обработке текстовых данных. В перечень элементов управления, способных выводить на экран пояснительные надписи, входят: метка `TLabel`, статический текст `TStaticText`. Кроме того, существует метка `TLinkLabel`, способная работать со ссылками HTML.

Список предназначенных для редактирования текста компонентов возглавляет строка ввода `TEdit`. Этот элемент управления позволит пользователю ввести простейшую строку текста. Строка ввода с маской `TMaskEdit` решает проблему ввода текста в соответствии с установленным шаблоном (маской). Элементы управления `TLabeledEdit` и `TButtonedEdit` представляют собой симбиоз строки ввода с текстовой меткой и кнопкой. Многострочный текстовый редактор `TMemo` способен обрабатывать неформатированные текстовые строки. На вершине пирамиды текстовых элементов управления располагается полноценный текстовый редактор `TRichEdit`, обладающий существенным набором функций для работы с текстом с возможностью расширенного форматирования (Rich Text Format, RTF).

Компоненты отображения текста

Общая черта всех рассматриваемых в этом разделе компонентов в том, что они предназначены для вывода пояснительных надписей на формах проекта. Отображаемая элементом управления текстовая строка недоступна для редактирования пользователем. Изменения в текст могут быть внесены, только если это предусмотрено программистом в исходном коде программы.

Метка *TLabel*

Из всех рассматриваемых в этой главе элементов управления метка `TLabel` является единственным графическим компонентом — потомком класса `TGraphicControl`. Из-за отсутствия в перечне предков класса `TWinControl` метка не обладает возможностью получать фокус

ввода, и, как следствие, в перечне событий TLabel вы не найдете и упоминания о событиях, связанных с нажатием клавиш.

> ***Внимание!***
>
> Все построенные на основе класса TGraphicControl графические элементы управления менее ресурсоемки, чем их "коллеги" — компоненты, базирующиеся на классе TWinControl.

Ключевое свойство метки (впрочем, как и двух ее "коллег" с рис. 17.1) — заголовок

```
property Caption : string;
```

Здесь хранится отображаемая компонентом текстовая надпись.

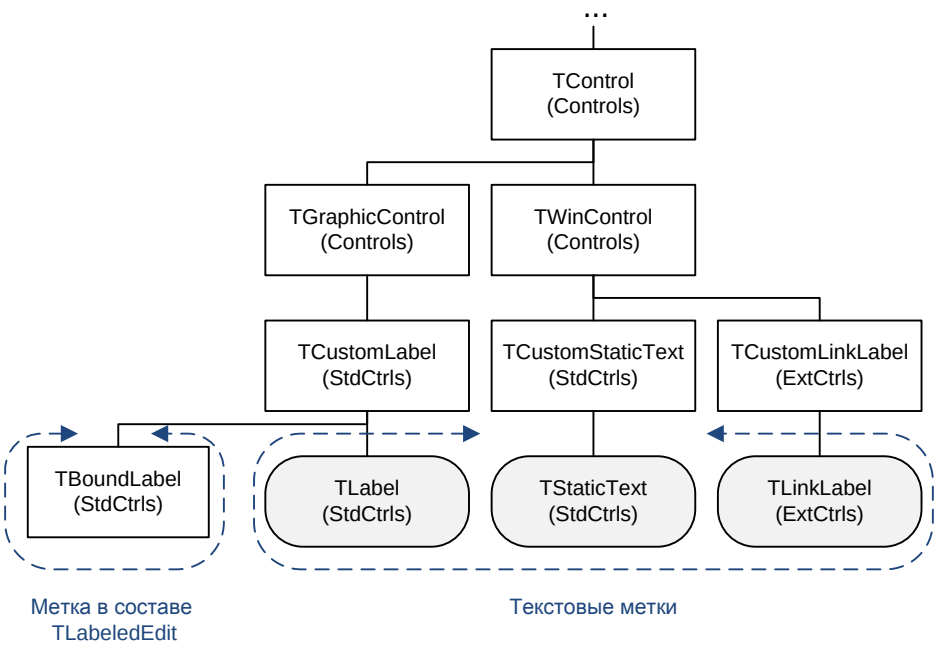


Рис. 17.1. Место компонентов-меток в иерархии VCL

За автоподстройку геометрических размеров компонента при выводе текста отвечает пара свойств:

```
property AutoSize: Boolean; //по умолчанию true
property WordWrap: Boolean; //по умолчанию false
```

По умолчанию (AutoSize=true и WordWrap=false) текст в метке располагается в одну строку, и метка растягивается по горизонтали так, чтобы вместить весь текст из свойства Caption. Размер компонента по вертикали зависит только от высоты установленного шрифта. Если оба свойства переведены в true, то компонент зафиксирует свой горизонтальный размер, но позволит изменять размер по вертикали для вывода текста в многострочном режиме.

Если режим автоматической подстройки размеров метки отключен (AutoSize=false), программисту будет позволено самостоятельно управлять выравниванием надписи в границах компонента. За выравнивание по горизонтали несет ответственность свойство

```
property Alignment: TAlignment;
type TAlignment = (taLeftJustify,  //по левому краю
                   taRightJustify, //по центру
                   taCenter);      //по правому краю
```

Вертикальное выравнивание текста определяется состоянием свойства

```
property Layout: TTextLayout;
type TTextLayout = (tlTop,     //по верхнему краю
                    tlCenter,  //по центру
                    tlBottom); //по нижнему краю
```

В ситуации, когда геометрические размеры метки физически не позволяют отобразить на экране весь текст из свойства `Caption`, об этом стоит уведомить пользователя. В этом случае поведение метки определяется свойством

```
property EllipsisPosition: TEllipsisPosition;//по умолчанию epNone
```

Возможные варианты состояния свойства отражены в табл. 17.1.

Таблица 17.1. *Возможные значения* `TEllipsisPosition`

Значение	Описание
`epNone`	Свойство неактивно
`epEndEllipsis`	Заменяет окончание текстовой строки многоточием, если строка не вмещается в отведенные ей границы
`epWordEllipsis`	Все не помещающиеся в пределы ограничивающего прямоугольника слова завершаются символом многоточия
`epPathEllipsis`	Заменяет многоточием середину строки с целью вместить начало и окончание текста в ограничивающий прямоугольник, если строка содержит наклонную черту (символ `\`), многоточие по возможности добавляется после наклонной черты

Несмотря на то что метка ни в коем случае не способна получать фокус ввода, она в состоянии проявить сообразительность и присоединиться к любому оконному элементу управления (потомку класса `TWinControl`) с помощью свойства

```
property FocusControl: TWinControl;
```

С момента присоединения к метке оконный элемент управления сможет получать фокус ввода при нажатии пользователем связанной с меткой клавиши-акселератора. Еще одним условием передачи фокуса ввода должен быть перевод в состояние `true` свойства

```
property ShowAccelChar: Boolean; //по умолчанию true
```

и включение в состав отображаемой меткой надписи (свойства `Caption`) символа амперсанда `&`. С этого момента метка начинает подчеркивать следующий за амперсандом символ.

Ряд свойств метки нацелен на поддержку привлекательного внешнего вида компонента. Для включения режима прозрачности установите в `true` свойство

```
property Transparent: Boolean; //по умолчанию true
```

Свойство

```
property GlowSize: Integer; //по умолчанию 0
```

определяет радиус области подсветки вокруг метки.

Статический текст *TStaticText*

В 9 случаях из 10 при выводе пояснительных надписей на рабочей форме приложения следует пользоваться услугами обычной метки `TLabel`. Единственное исключение из правил имеет место в тех случаях, когда логика работы проекта предусматривает необходимость обращаться к пояснительной надписи, как к окну, средствами Win API. В таком случае функция прикладного программного интерфейса потребует от нас дескриптор окна, а им обладают только потомки класса `TWinControl`.

Возможности функций Win API практически безграничны и позволяют профессиональным разработчикам создавать высокоэффективный исполняемый код. Листинг 17.1 демонстрирует процесс получения данных об имени файла исполняемого модуля, которому принадлежит оконный элемент управления `TStaticText`.

Листинг 17.1. Пример обращения к компоненту `TStaticText` средствами Win API

```
var FileName: array[0..MAX_PATH] of WideChar;
begin
  GetWindowModuleFileName(StaticText1.Handle,FileName,MAX_PATH);
  SetWindowText(StaticText1.Handle,FileName);
end;
```

Одно из достоинств компонента `TStaticText` — возможность обрамления текста бордюром, для этого компонент вооружен свойством

```
property BorderStyle: TStaticBorderStyle; //по умолчанию sbsNone
type TStaticBorderStyle = (sbsNone, sbsSingle, sbsSunken);
```

Кроме того, в качестве наследства от `TWinControl` у компонента имеется широкий набор свойств (`BevelKind`, `BevelEdges`, `BevelInner`, `BevelOuter`), позволяющих настроить визуальный вид фасок элемента управления.

В остальном набор свойств, методов и обработчиков событий класса `TStaticText` похож на класс `TLabel`.

Метка-ссылка *TLinkLabel*

Метка `TLinkLabel` может пригодиться в ситуации, когда на поверхности рабочей формы проекта необходимо отобразить ссылку на интернет-ресурс. Ссылка на ресурс должна быть оформлена по правилам языка гипертекстовой разметки HTML, тег с описанием ссылки передается в свойство `Caption` компонента.

Листинг 17.2 демонстрирует один из вариантов определения в заголовке метки URL (Uniform Resource Locator) адреса сайта издательства "БХВ-Петербург" во Всемирной паутине.

Листинг 17.2. Строка HTML со ссылкой на интернет-ресурс

```
LinkLabel1.Caption:=
'<a href="http://www.bhv.ru/">Издательство БХВ-Петербург</a>';
```

В заголовке метки `LinkLabel1` будет выведено не все содержимое тега HTML, а только подчеркнутый текст "Издательство БХВ-Петербург".

Для того чтобы приложение смогло воспользоваться услугами метки-ссылки, следует обратиться за помощью к ключевому событию компонента

```
property OnLinkClick: TSysLinkEvent;
TSysLinkEvent = procedure(Sender: TObject; const Link: string;
                          LinkType: TSysLinkType) of object;
type TSysLinkType = (sltURL, sltID);
```

Обработчик события самостоятельно выделит из строки HTML адрес и передаст его в константу `Link`. Тип соединения определяет параметр `LinkType`.

В листинге 17.3 вы найдете пример, демонстрирующий порядок запуска Internet Explorer из приложения Delphi в момент возникновения события `OnLinkClick()`. Для старта браузера и передачи ему ссылки на ресурс мы вновь воспользовались услугами Windows API, на этот раз нам пригодилась функция `CreateProcess()`. В первый параметр функции направляется путь и имя файла браузера, а во второй — ссылка `Link`.

Листинг 17.3. Запуск Internet Explorer по ссылке

```
procedure TForm1.LinkLabel1LinkClick(Sender: TObject; const Link: string;
  LinkType: TSysLinkType);
var StartupInfo : TStartupInfo;
    ProcessInformation : TProcessInformation;
    SD : TSecurityDescriptor;
    SA : TSecurityAttributes;
begin
  //инициализация и заполнение атрибутов безопасности
  InitializeSecurityDescriptor(@SD,SECURITY_DESCRIPTOR_REVISION);
  SA.nLength:=SizeOf(TSecurityAttributes);
  SA.lpSecurityDescriptor:=@SD;
  SA.bInheritHandle:=True;
  FillChar(StartupInfo,SizeOf(StartupInfo),#0);

  CreateProcess(
     PWideChar('C:\Program Files\Internet Explorer\iexplore.exe'),
     PWideChar('open '+Link),//командная строка с адресом
              @SA,
              @SA,
              false,
              NORMAL_PRIORITY_CLASS,
              nil,
              nil,
              StartupInfo,
              ProcessInformation);
end;
```

Компоненты редактирования текста

Наиболее важные элементы управления, позволяющие пользователю не только видеть, но и редактировать текст, отражены на рис. 17.2. Все рассматриваемые в главе компоненты редактирования текста объединяют две ключевые родовые черты:

- благодаря своему предку, классу `TWinControl`, они являются классическими оконными элементами управления и приобрели характерные для объектов такого типа особенности (наличие дескриптора окна, возможность реагировать на ввод данных с клавиатуры и т. д.);
- основные навыки по редактированию текстовых данных были приобретены благодаря наличию в цепочке наследования класса `TCustomEdit`.

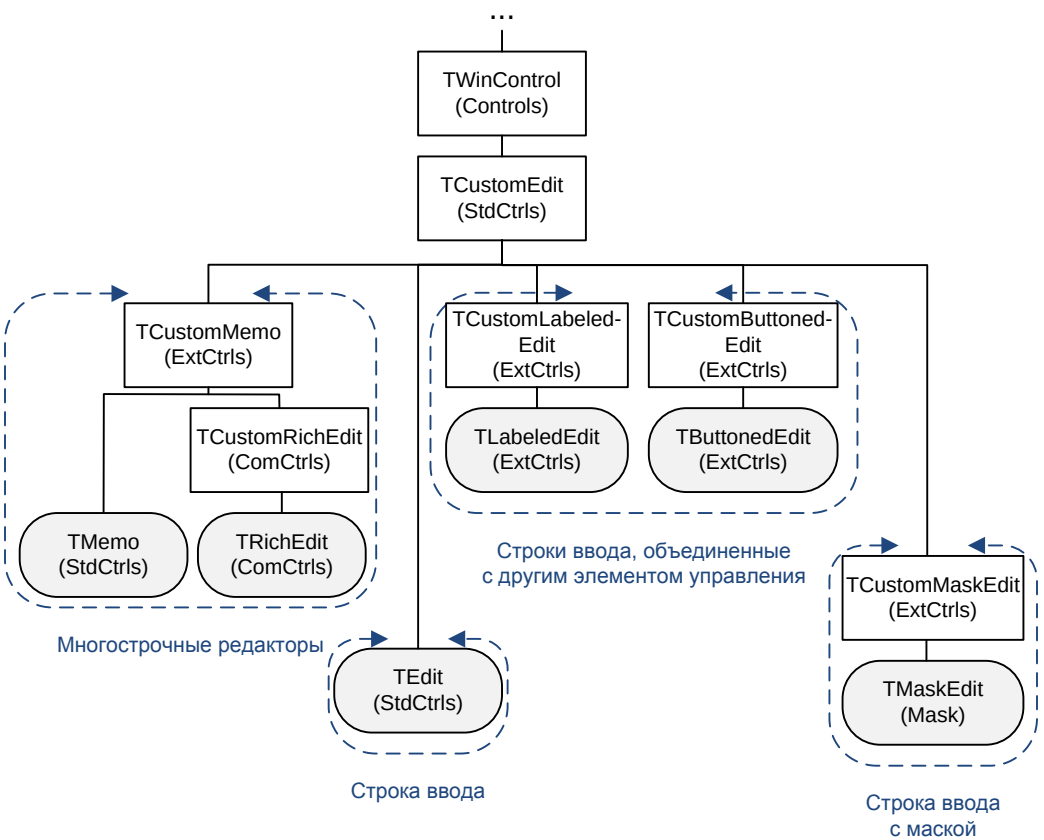


Рис. 17.2. Компоненты — текстовые редакторы в иерархии классов VCL

Основа текстовых редакторов, класс *TCustomEdit*

Основой всех компонентов VCL, специализирующихся на редактировании текстовых данных, является класс `TCustomEdit`. В нем инкапсулированы свойства и методы, обеспечивающие:

- хранение и редактирование текста;
- выделение части текста с возможностью редактирования только этой части;
- реагирование на любые изменения в содержании текста.

С основным свойством всех текстовых компонентов мы знакомы уже давно. Еще при изучении класса `TControl` упоминалось о свойстве, специализирующемся на хранении текстовой информации

```
property Text;
```

Именно со свойством `Text` прямо или косвенно связана деятельность подавляющего большинства свойств и методов `TCustomEdit`.

> ***Примечание***
>
> Обратите внимание, что, упомянув свойство `Text`, мы не стали конкретизировать тип данных (допустим `String` или `AnsiString`). Современная версия класса `TCustomEdit` самостоятельно разберется с обрабатываемым типом текстовых данных.

Полную очистку текста осуществляет метод:

```
procedure Clear;
```

При каждом изменении текста (будь то ручной ввод или модификация текста программным образом) вызывается обработчик события

```
property OnChange: TNotifyEvent;
```

Это событие обычно задействуется для контроля обрабатываемого текста. Например, листинг 17.4 демонстрирует, как можно уведомить пользователя о том, что введенная им текстовая строка достигла ограничения в 10 символов.

Листинг 17.4. Контроль изменений текста в событии `OnChange()`

```
procedure TForm1.Edit1Change(Sender: TObject);
var P:TPoint;
    BH:TBalloonHint;
begin
  if Length(Edit1.Text)>=10 then
  begin
    BH:=TBalloonHint.Create(Edit1); //создаем сообщение
    BH.FreeOnRelease;               //сообщение будет уничтожено
    BH.HideAfter:=2500;             //после 2,5 секунд
    BH.Title:='Достигнуто ограничение 10 символов!';
    {рассчитываем координаты точки вывода сообщения}
    P:=Form1.ClientToScreen(Point(Edit1.Left, Edit1.Top+Edit1.Height));
    BH.ShowHint(P);                 //вывод сообщения
  end;
end;
```

Предложенный код контролирует длину текстовой строки. Как только длина достигнет 10 символов, на экран выводится предупредительное сообщение.

Ограничения на ввод

Класс позволяет наложить некоторые ограничения на вводимые текстовые данные. Запрет на редактирование текста накладывает свойство

```
property ReadOnly: Boolean;//по умолчанию false
```

при установке его в `true`.

Ограничение на максимальное количество символов, хранящихся в свойстве Text, поможет назначить свойство

```
property MaxLength: Integer;//по умолчанию 0 — ограничений нет
```

Нулевое значение говорит о том, что длина строки не ограничена.

Можно запретить вводить нецифровые символы, для этого следует установить в состояние true свойство

```
property NumbersOnly: Boolean;
```

Если планируется применение потомка класса TCustomEdit для ввода конфиденциальной информации (например, пароля доступа) и пользователь программного обеспечения не планирует ознакомить с нею случайно заглянувшего в монитор прохожего, то для скрытия вводимого текста рекомендуется воспользоваться свойством

```
property PasswordChar: Char;//по умолчанию #0
```

Если значение свойства отличается от символа с нулевым кодом, то при отображении содержимого строки реальные символы станут подменяться символом из PasswordChar.

Установкой регистра символов вводимого текста занимается свойство

```
property CharCase: TEditCharCase;
type TEditCharCase = (ecNormal    {Нормальный регистр символов},
                      ecUpperCase {ПРОПИСНЫЕ СИМВОЛЫ},
                      ecLowerCase {строчные символы});
```

Если предполагается, что вводимые текстовые данные могут быть в кодировке OEM, то для преобразования данных в текущую кодировку ANSI (и наоборот) потребуется свойство

```
property OEMConvert: Boolean;
```

> ***ЗАМЕЧАНИЕ***
>
> При изменении кодировки символов не забывайте о существовании у класса шрифтов TFont свойства Charset, определяющего номер набора символов шрифта. Для вывода кириллицы используйте RUSSIAN_CHARSET.

Автоматическую подстройку вертикального размера элемента управления под высоту используемого шрифта осуществляет свойство

```
property AutoSize: Boolean;
```

Выделение части текста

У прямых потомков класса TCustomEdit строк редактирования TEdit и TMaskEdit объявлено свойство

```
property AutoSelect: Boolean;
```

Если свойство установлено в состояние true, то, получая фокус ввода, текстовый редактор автоматически выделит весь текст.

По умолчанию подсветка выделенного в элементе управления текста снимается, когда он теряет фокус ввода. Положение вещей можно изменить с помощью свойства

```
property HideSelection: Boolean;
```

Значение true требует скрывать выделение, а false, напротив, сохраняет выделение.

Благодаря свойству

```
property SelText: string;
```

возможно обратиться только к выделенному фрагменту текста.

При желании выделение текста производится не только мышью или клавишами управления курсором, но и программным способом. Для этого объявлены два свойства:

```
property SelStart: Integer;
property SelLength: Integer;
```

Они определяют порядковый номер первого выделяемого символа и всю длину выделяемого текста соответственно (см. листинг 28.10).

Для выделения всего текста воспользуйтесь методом

```
procedure SelectAll;
```

Для удаления выделенного текста пригодится процедура:

```
procedure ClearSelection;
```

Для работы с выделенным текстом предусмотрены методы редактирования, совместимые с форматом функций Windows API:

```
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer;
procedure SetSelTextBuf(Buffer: PChar);
```

Первый метод позволяет считать текст в буфер, второй передает в элемент управления текст из буфера.

Взаимодействие с буфером обмена

Взаимодействие с буфером обмена Windows производится с помощью процедур:

```
procedure CopyToClipboard;    //копировать в буфер
procedure CutToClipboard;     //вырезать в буфер
procedure PasteFromClipboard; //вставить из буфера
```

Отмена изменений

Для проверки, модифицировался ли текст компонента, воспользуемся свойством:

```
property Modified : Boolean;
```

Если содержимое свойства `Text` изменялось, то свойство автоматически переводится в состояние `true`. С этим свойством мы уже сталкивались в листинге 15.6. Здесь, контролируя `Modified` многострочного редактора `Memo1`, мы сумели предотвратить несанкционированную потерю данных при закрытии формы.

У потомков `TCustomEdit` предусмотрена возможность отмены последних изменений в редактируемом тексте. Если содержимое свойства `Text` изменялось и элемент управления помнит его предыдущее значение, то в `true` устанавливается свойство

```
property CanUndo: Boolean; //только для чтения
```

В таком случае откат к исходному состоянию обеспечит метод

```
procedure Undo;
```

Строка ввода *TEdit*

Из всех компонентов, способных редактировать текст, у программистов наибольшей популярностью обладает строка ввода `TEdit`. Заметим сразу, что набор свойств и методов строки ввода ничем не отличается от перечня свойств и методов опорного класса `TCustomEdit`.

Строка ввода с кнопками *TButtonedEdit*

Класс `TButtonedEdit` представляет собой симбиоз двух элементов управления: классической строки ввода на основе класса `TCustomEdit` и одной или двух кнопок, построенных на базе класса `TEditButton`. Кнопки могут подключаться к левой и правой частям строки ввода. Для этого предназначены свойства

```
property LeftButton: TEditButton;
property RightButton: TEditButton;
```

Для включения кнопки достаточно передать значение `true` в свойство `Enabled`.

Для того чтобы кнопки стали видимыми, недостаточно включить свойство `Visible`. Кроме этого к строке ввода следует присоединить коллекцию изображений — компонент `TImageList` с загруженными значками, для этого предназначено свойство

```
property Images: TCustomImageList;
```

С одной кнопкой можно связать до 4 значков, их индексы передаются в свойства

```
property ImageIndex: TImageIndex;         //текущее изображение
property HotImageIndex: TImageIndex;      //при наведении указателя мыши
property PressedImageIndex: TImageIndex; //кнопка нажата
property DisabledImageIndex: TImageIndex;//кнопка отключена
```

Обязательно для заполнения только свойство `ImageIndex`, в нем находится идентификатор основного значка кнопки. Остальные свойства просто делают поведение кнопки более наглядным.

Для реагирования на щелчок предназначены события

```
property OnLeftButtonClick: TNotifyEvent; //щелчок по левой кнопке мыши
property OnRightButtonClick: TNotifyEvent;//щелчок по правой кнопке мыши
```

Строка ввода с меткой *TLabeledEdit*

Строка ввода с меткой представляет собой логическое объединение строки и метки, содержащей пояснительную надпись. Метка реализуется на основе класса `TBoundLabel` и по своим возможностям практически полностью повторяет рассмотренный в начале главы классический компонент `TLabel` (см. рис. 17.1).

Для обращения к метке следует воспользоваться свойством

```
property EditLabel: TBoundLabel;
```

Если вы обратитесь к свойству в Инспекторе объектов во время визуального проектирования, то перед вами развернется список свойств, практически полностью повторяющий перечень свойств классической метки `TLabel`.

Свойство

```
property LabelPosition: TLabelPosition;
type TLabelPosition = (lpAbove, lpBelow, lpLeft, lpRight);
```

определяет местоположение пояснительной надписи относительно основной строки ввода. По умолчанию метка располагается над строкой, что соответствует состоянию `lpAbove`.

Отступ метки от строки ввода назначает свойство

```
property LabelSpacing: Integer;
```

Изначально отступ равен 3 пикселам.

Строка ввода с маской *TMaskEdit*

Строка ввода с маской служит специальным редактором, осуществляющим контроль за вводимым пользователем текстом в соответствии с заданным шаблоном. Шаблон может представлять собой номер телефона, дату, время, счет в банке и т. п.

Шаблон ввода задается в редакторе маски

```
property EditMask: TEditMask;
```

Структурно шаблон ввода состоит из трех полей, разделенных точкой с запятой. В первом поле задается маска ввода. Управляющие символы маски приведены в табл. 17.2. Особый интерес представляет символ наклонной черты `\`. Он указывает на то, что следующий за ним символ является *литералом* (символом оформления, отображаемым в строке ввода). В примере (рис. 17.3) наклонная черта предшествует круглым скобкам, в которые заключается телефонный код города, т. е. круглые скобки и являются литералами.

Таблица 17.2. *Управляющие символы шаблона ввода*

Символ	Описание	Символ	Описание
!	Подавление пробелов в тексте	>	Перевод в верхний регистр
<	Перевод в нижний регистр	<>	Отмена перевода регистров
\	Следующий символ — литерал	_	Пустое поле
L	Обязательна буква	l	Может быть буква
A	Обязательна буква или цифра	a	Может быть буква или цифра
C	Обязателен любой символ	c	Может быть любой символ
0	Обязательна цифра	9	Может быть цифра
#	Может быть цифра, знак "+" или "–"	:	Разделитель часов, минут и секунд
/	Разделитель дней, месяцев и годов	;	Разделение полей в маске

Второе поле маски способно принимать два значения — `0` или `1`. В первом случае в свойство строки ввода `Text` символы-литералы не включаются, при втором варианте в обработанный текст войдут все символы.

Число символов в свойстве `Text` возвратит функция:

```
function GetTextLen: Integer;
```

Вне зависимости от состояния второго поля шаблона в свойстве компонента

```
property EditText: string;
```

окажется полный вариант текста. И наконец, третье поле маски хранит единственный символ, называемый символом подстановки. До тех пор, пока пользователь не введет весь текст, этот символ будет отображаться во всех пустых полях строки ввода.

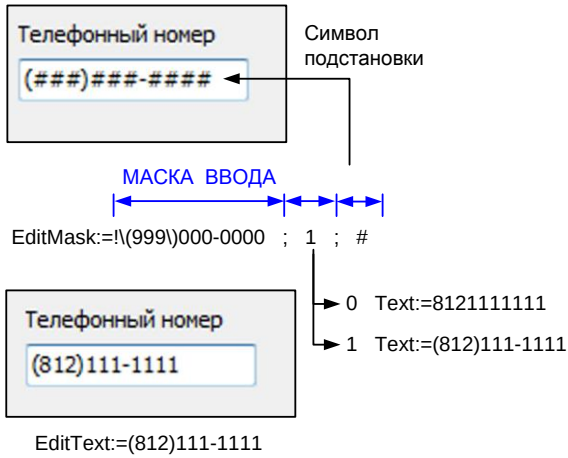


Рис. 17.3. Состав маски ввода на примере шаблона телефонного номера

Соответствие введенного текста маске ввода проверяет метод:

```
procedure ValidateEdit;
```

При ошибке ввода методом генерируется исключительная ситуация `EDBEditError`. Как правило, нет необходимости использовать эту процедуру в исходном коде, т. к. она автоматически вызывается при утере строкой фокуса.

Многострочный текстовый редактор *TMemo*

Создаваемые на базе класса `TMemo` элементы управления применяются для ввода и редактирования многострочного неформатированного текста. Многострочный редактор представляет собой симбиоз оконного элемента управления и класса `TStrings`, инкапсулированного в качестве свойства

```
property Lines: TStrings;
```

Свойство `Lines` позволяет хранить многострочные текстовые данные.

> ***Замечание***
>
> Возможности класса `TStrings`, осуществляющего операции с набором строк, подробно изложены в *главе 10*.

Местоположение каретки можно узнать благодаря свойству

```
property CaretPos: TPoint; //только для чтения
```

Свойство возвращает номер строки и порядковый номер символа в строке.

Если все текстовые строки не помещаются в видимой части окна, включите полосы прокрутки:

```
property ScrollBars: TScrollStyle;
type TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth);
```

Настройка поведения элемента управления в зависимости от длины строки производится свойством

```
property WordWrap: Boolean;
```

Если свойство установлено в `true` и длина строки превышает горизонтальный размер видимой части окна, то осуществляется мягкий перенос непомещающихся символов на новую строку.

Реакция на нажатие клавиш ввода и табуляции определяется свойствами:

```
property WantReturns : Boolean;
property WantTabs : Boolean;
```

Установленное в `true` свойство `WantReturns` разрешает ввод символа перевода строки. Переведя в `true` свойство `WantTabs`, мы дадим разрешение на ввод символа табуляции в текстовые данные.

Редактор расширенного текстового формата *TRichEdit*

Среди всех текстовых редакторов из состава VCL наиболее впечатляющими возможностями обладает элемент управления `TRichEdit`. Класс представляет собой логическое развитие редактора многострочного текста с поддержкой форматирования RTF (rich-text format). Таким образом, на основе `TRichEdit` можно создать достаточно сложный текстовый редактор, по своим возможностям не уступающий редактору WordPad из состава Microsoft Windows.

Существенная часть методов и свойств класса нам уже хорошо знакома, т. к. она унаследована от `TCustomEdit` и `TCustomMemo`. Поэтому нам следует остановиться только на ключевых особенностях, внесенных классом `TCustomRichEdit`.

Как и в случае с обычным многострочным редактором `TMemo`, текстовые строки хранятся в свойстве

```
property Lines: TStrings;
```

Однако подход к обработке текстовых данных у `TRichEdit` принципиально другой. У редактора с расширенными возможностями есть две ключевые особенности, благодаря которым он приобретает такие способности, о которых не мог и мечтать обычный `TMemo`. *Во-первых*, редактор способен различать абзацы текста и применять к ним отдельные правила форматирования. Такой результат достигается благодаря классу `TParaAttributes`, он отвечает за описание атрибутов абзаца. *Во-вторых*, редактор умеет хранить сведения об индивидуальных характеристиках части текста внутри абзаца, этот функционал возложен на класс `TTextAttributes`.

Доступ к текущему абзацу текста обеспечивает свойство

```
property Paragraph: TParaAttributes;
```

Потенциал по форматированию абзаца достаточно высок, редактор умеет размещать и выравнивать текст, строить нумерованные списки, устанавливать метки табуляции и т. д.

> ***Внимание!***
>
> Важно помнить, что `TRichEdit` владеет одним-единственным объектом `TParaAttributes`, соответственно свойство `Paragraph` указывает лишь на текущий абзац (абзац, содержащий курсор ввода). Для форматирования других абзацев необходимо осуществить их перебор.

Все уникальные возможности редактора активизируются при посредничестве свойства

```
property PlainText: Boolean; //по умолчанию false
```

Если свойство установлено в `true`, то текст интерпретируется как обычный неформатированный. В противном случае вы получите все преимущества расширенного текстового формата.

Среди множества методов класса `TRichEdit` особое внимание стоит уделить встроенной функции поиска фрагмента текста:

```
function FindText(const SearchStr: string; StartPos, Length: Integer;
                  Options: TSearchTypes): Integer;
type TSearchType = (stWholeWord, stMatchCase);
```

В качестве параметров передаются: искомый фрагмент текста `SearchStr`; место в тексте, с которого начинается поиск `StartPos`; место, до которого производится поиск `StartPos+Length`. Опции поиска настраиваются в параметре `Options`, где флаг `stMatchCase` указывает на то, что поиск производится с учетом регистра символов. А флаг `stWholeWord` — что поиск ведется только целыми словами (другими словами, если вы ищете текст "**метр**", то поиск не будет остановлен на слове "пара**метр**"). В случае успеха метод возвращает позицию первого символа найденного фрагмента в тексте, иначе результатом станет –1.

Отметим наличие метода

```
procedure Print(const Caption: string);
```

позволяющего организовать простейший способ вывода на печать содержимого редактора. Перед отправкой текста на печать следует определить параметры страницы, для этого предназначено свойство

```
property PageRect: TRect;
```

Форматирование абзаца — класс *TParaAttributes*

Класс `TParaAttributes` определяет основные свойства абзаца текста. Выравнивание текста в абзаце определяет свойство

```
property Alignment: TAlignment;
type TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

В абзаце предусмотрена возможность назначения отступов текста. Для этого предоставлены три свойства, отвечающие за установку абзацного отступа, левой и правой границ абзаца.

```
property FirstIndent: Longint; //отступ первой строки абзаца
property LeftIndent: Longint;  //левая граница абзаца
property RightIndent: Longint; //правая граница абзаца
```

Во всех трех свойствах единицей измерения выступает пиксел.

Форматирование текста в виде маркированного списка обеспечивается установкой в состояние `nsBullet` свойства

```
property Numbering: TNumberingStyle;
type TNumberingStyle = (nsNone, nsBullet);
```

Позиции табуляции внутри абзаца устанавливаются в массиве `Tab`. Помните, что массив не вставляет непосредственно символы табуляции (это осуществляется нажатием клавиши <Tab>), а просто расставляет места предполагаемых табуляторов.

```
property Tab[Index: Byte]: Longint;
```

Количество табуляторов в абзаце доступно в свойстве:

```
property TabCount: Integer;
```

Класс `TParaAttributes` не богат собственными методами. Отмечу наличие метода "формат по образцу":

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

Метод назначает абзацу такие же параметры, как и у источника `Source`.

Атрибуты текста — класс *TTextAttributes*

В отличие от обычного класса `TMemo` расширенный текстовый редактор способен настраивать текстовые атрибуты каждого отдельного слова и даже символа в абзаце. Такой функциональностью владеет инкапсулированный в состав компонента класс `TTextAttributes`. Класс производит комплекс настроек текстовых атрибутов с возможностью последующего их применения к выделенному тексту. Свойства класса текстовых атрибутов созвучны свойствам класса шрифтов `TFont`. На базе этого класса функционируют два свойства:

```
property DefAttributes: TTextAttributes;
property SelAttributes: TTextAttributes;
```

Первое свойство (атрибуты по умолчанию) доступно только в период выполнения приложения. Оно описывает характеристики шрифта, устанавливаемые по умолчанию для вновь вводимого текста. Второе свойство (выделенные атрибуты) возвращает или устанавливает атрибуты выделенного текста или части текста, в которой в данный момент находится курсор ввода (листинг 17.5).

Листинг 17.5. Управления атрибутами шрифта

```
public
  function SelectedTextAttributes: TTextAttributes;
  procedure SetFontSize(NewSize : byte);//установка высоты шрифта
end;
. . .
function TfrmMain.SelectedTextAttributes: TTextAttributes;
begin
  //получаем атрибуты выделенного текста
  if RichEdit1.SelLength>0 then Result:=RichEdit1.SelAttributes
  else
    Result:=RichEdit1.DefAttributes;
end;

procedure TfrmMain.SetFontSize(NewSize : byte);
begin
  //в параметре NewSize передается новая высота шрифта
  SelectedTextAttributes.Size := NewSize;
end;
```

Особенности обработки событий

Компонент `TRichEdit` обладает рядом специфичных обработчиков событий. Например, при любых изменениях в области выделенного текста вызывается событие

```
property OnSelectionChange: TNotifyEvent;
```

Допустим, что для настройки параметров текста в компоненте `RichEdit1` используется ряд элементов управления: в комбинированном списке `cbFonts:TComboBox` хранятся названия гарнитур доступных шрифтов; в компоненте `UpDownFontSize:TUpDown` содержится размер шрифта; в комбинированном списке выбора цвета `cbColor:TColorBox` — цвет шрифта. В рамках события `OnSelectionChange()` мы можем синхронизировать состояние этих элементов управления с параметрами текста, расположенного под курсором ввода (листинг 17.6).

Листинг 17.6. Установка элементов управления в актуальное состояние

```
procedure TfrmMain.RichEdit1SelectionChange(Sender: TObject);
begin
   //шрифты
    cbFonts.Text := RichEdit1.SelAttributes.Name;
    UpDownFontSize.Position:=RichEdit1.SelAttributes.Size;
    cbColor.Selected:= RichEdit1.SelAttributes.Color;
...
   //положение каретки
   StatusBar1.Panels[0].Text:=
     Format('%d:%d',[RichEdit1.CaretPos.X,RichEdit1.CaretPos.Y]);
end;
```

Так как событие `OnSelectionChange()` генерируется не только в момент редактирования текста, но и при простейшем перемещении текстового курсора внутри текста, оно позволит отслеживать все параметры абзаца и атрибуты текста. Об этом свидетельствует экранный снимок редактора текста RTF, построенного на базе компонента `TRichEdit` (рис. 17.4).

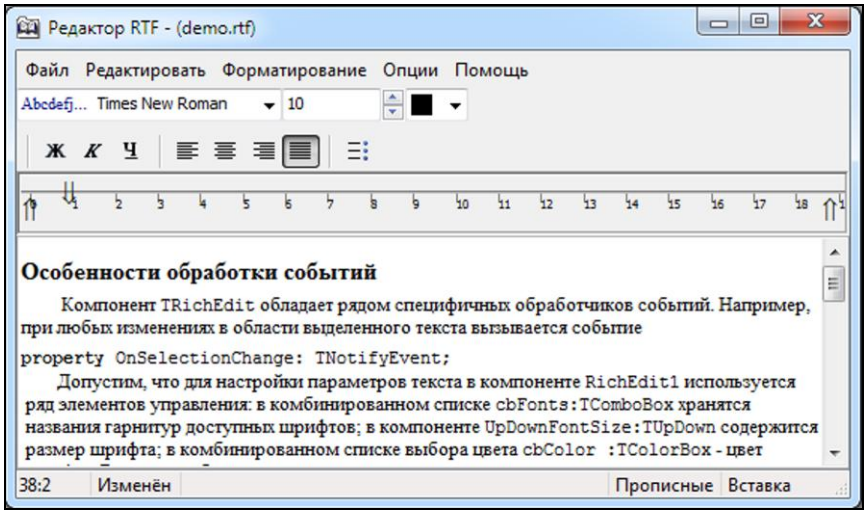


Рис. 17.4. Экранный снимок редактора на основе `TRichEdit`

В компоненте предусмотрена реакция на попытку изменения защищенного текста (текста с атрибутом `Protected`, установленным в `true`):

```
property OnProtectChange: TRichEditProtectChange;
type TRichEditProtectChange = procedure(Sender: TObject;
          StartPos, EndPos: Integer; var AllowChange: Boolean) of object;
```

Параметры `StartPos` и `EndPos` проинформируют нас об участке текста, в котором была предпринята попытка редактирования. Разрешение (`true`) или запрет (`false`) на изменения определяется программистом в формальном параметре `AllowChange`.

При наступлении противоречий между размерами редактора и объемом содержащегося в нем текста возникает событие

```
property OnResizeRequest: TRichEditResizeEvent;
type TRichEditResizeEvent =
      procedure(Sender: TObject; Rect: TRect) of object;
```

Событие уведомит программиста об оптимальном размере редактора в параметре `Rect`.

При уничтожении экземпляра редактора `TRichEdit` последний проверяет, не находится ли в буфере обмена его текст. В этом случае возникает событие:

```
property OnSaveClipboard: TRichEditSaveClipboard;
type TRichEditSaveClipboard = procedure(Sender: TObject;
  NumObjects, NumChars: Integer; var SaveClipboard: Boolean) of object;
```

В параметре `NumObjects` окажется количество объектов буфера, в `NumChars` — число символов. Для очистки буфера передайте в параметр `SaveClipboard` значение `false`.

ГЛАВА 18

Кнопки и компоненты выбора значений

Интересно, считал ли кто-нибудь, сколько раз за день нам приходится нажимать различные кнопки? Будь это кнопка пульта дистанционного управления, кнопка вызова лифта или выключателя света в настольной лампе, по нажатию происходило какое-то событие: переключались каналы телевизора, приезжал лифт и загорался свет. Перекочевав из повседневной жизни на экраны компьютеров, кнопка превратилась в самый популярный элемент интерфейса современного приложения. Именно благодаря кнопке формируется доступный и интуитивно понятный даже начинающему пользователю способ управления программой. Мы уже не раз задействовали кнопку `TButton` в наших примерах. Однако список кнопок не исчерпывается классом `TButton` — это только начало. Стремление к удобствам, наглядности восприятия, повышению функциональной нагрузки на компонент породило широкий спектр элементов управления, зачастую внешне лишь отдаленно напоминающих обычную кнопку, однако очень схожих по характеру решаемых ими задач.

Все изучаемые в первой части главы кнопки берут начало от базового класса `TWinControl`, подробно рассмотренного нами в *главе 13*, и наследуют объявленные в нем свойства и методы. Вместе с тем иерархическое дерево (рис. 18.1) элементов управления кнопок представлено несколькими самостоятельными ветвями, вносящими в компоненты свои, зачастую принципиальные особенности. Поэтому далее будут рассмотрены только характерные для каждого элемента управления свойства, методы и обработчики событий. А из материала, пройденного ранее, вспомним главное — основным событием, связанным с любой из кнопок, является событие-щелчок:

```
property OnClick: TNotifyEvent;
```

Событие может быть инициировано щелчком кнопки мыши, нажатием клавиш или вызвано программным образом.

Кнопка *TButton*

Элемент управления `TButton` — самая обычная кнопка, родившаяся вместе с первыми версиями Windows и до сего дня не утратившая своих твердо занимаемых позиций. В первую очередь поведение кнопки определяется свойством

```
property Style: TButtonStyle;//по умолчанию bsPushButton
type TButtonStyle = (bsPushButton,  //стандартная кнопка
                     bsCommandLink, //кнопка — команда
                     bsSplitButton);//кнопка с выпадающим меню
```

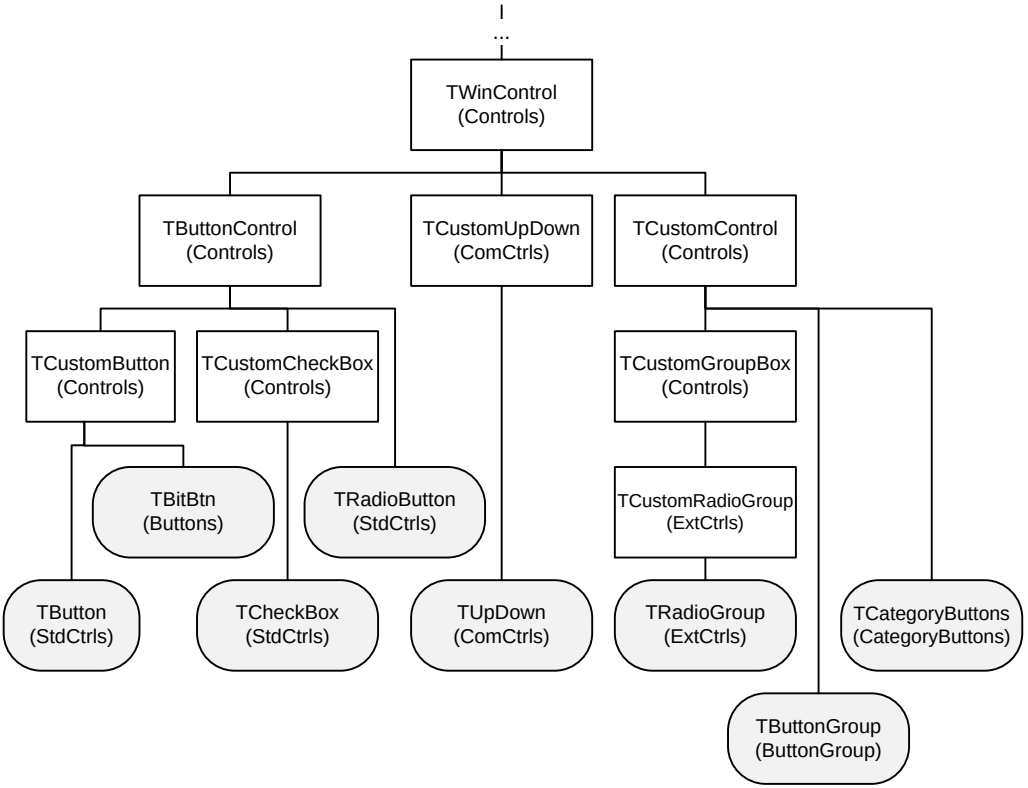


Рис. 18.1. Классы кнопок и компонентов выбора значений в иерархии VCL

По умолчанию стиль установлен в состояние `bsPushButton`. Это соответствует обычной кнопке, которую можно нажать указателем мыши. Стили `bsCommandLink` и `bsSplitButton` доступны, только начиная ОС с Windows Vista.

Если для кнопки выбран командный стиль работы, то стоит обратить внимание на свойство

```
property CommandLinkHint: string;
```

В нем содержится пояснительный текст, который подлежит выводу на экран в дополнительной строке, расположенной немного ниже основного заголовка кнопки.

При назначении стиля `bsSplitButton` к кнопке с помощью свойства

```
property DropDownMenu : TPopupMenu;
```

следует подключить всплывающее меню (компонент `TPopupMenu`). Щелчок по правой части кнопки приводит к выпадению из кнопки меню, с помощью которого пользователь сможет выбрать интересующее его действие (рис. 18.2). Особо отметим, что щелчок сопровождается генерацией события

```
property OnDropDownClick: TNotifyEvent;
```

Это событие можно задействовать для обновления элементов меню.

Очередная особенность кнопки `TButton` проявляется при работе с модальными окнами. Задачей такого окна является ожидание выбора пользователем какого-то варианта дальней-

ших действий приложения. Простейшим примером служит диалог подтверждения удаления файла из Проводника Windows. Проводник не продолжит выполнение поставленной задачи до тех пор, пока пользователь не примет решения (нажмет кнопку удаления файла или кнопку отмены), при этом переключение к другим окнам этого приложения будет заблокировано.

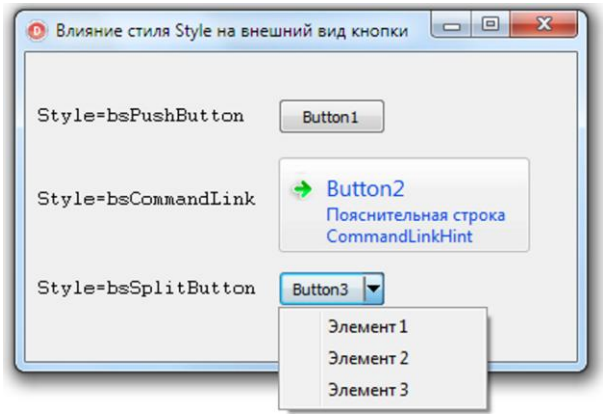


Рис. 18.2. Внешний вид кнопок `TButton` с разными стилями `Style`

В классе опубликованы три свойства, предназначенные для настройки поведения кнопки при работе с модальными окнами. Нажатие кнопки передает модальному окну (диалогу) значение, определенное в ее свойстве:

```
property ModalResult: TModalResult;
type TModalResult = Low(Integer)..High(Integer);
```

Это значение называется *модальным результатом*. Более подробно о модальных окнах (в терминологии Delphi — модальных формах) мы уже говорили в *главе 15*.

Хорошим правилом считается размещение на модальной форме двух кнопок, реагирующих на нажатие клавиш <Enter> и <Esc>. При нажатии клавиши <Enter> принимается вариант действий, по умолчанию предлагаемый модальной формой. При нажатии клавиши отмены — вариант отклоняется. Для этого предназначены два свойства:

```
property Default: Boolean;
property Cancel: Boolean;
```

При установке `Default` в состояние `true` кнопка реагирует на нажатие клавиши <Enter>. Присвоив `true` свойству `Cancel`, мы обеспечим реакцию на нажатие клавиши <Esc>.

В ранних версиях Delphi кнопка `TButton` не обладала возможностью выводить на своей поверхности рисунок, о назначении кнопки пользователь судил лишь по тексту в ее заголовке `Caption`. Сравнительно недавно кнопка (а точнее, все потомки `TButtonControl`) была усовершенствована и приобрела способность отображать на своей поверхности значок (табл. 18.1).

Одновременно с выходом в свет ОС Microsoft Vista в распоряжении кнопки появилось свойство

```
property ElevationRequired: Boolean;//по умолчанию false
```

Перевод свойства в состояние `true` приводит к появлению на поверхности кнопки изображения щита, которое должно подсказать пользователю, что для обращения к кнопке ему следует обзавестись правами администратора.

Таблица 18.1. *Отображение значков на поверхности кнопки* `TButton`

Свойство	Описание
`property Images: TCustomImageList;`	Свойство для подключения коллекции изображений (`TImageList`) для значков
`property ImageIndex: TImageIndex;`	Индекс основного значка
`property PressedImageIndex: TImageIndex;`	Индекс значка, соответствующего нажатой кнопке
`property DisabledImageIndex: TImageIndex;`	Индекс значка для отключенной кнопки (`Enabled=false`)
`property HotImageIndex: TImageIndex;`	Индекс значка для кнопки, над которой находится указатель мыши
`property StylusHotImageIndex: TImageIndex;`	Индекс значка для кнопки, над которой находится электронное перо
`property ImageAlignment: TImageAlignment;`	Расположение значка относительно заголовка кнопки
`property ImageMargins: TImageMargins;`	Определяет точную позицию значка на поверхности кнопки

Кнопка с рисунком *TBitBtn*

Кнопка `TBitBtn` предназначена для применения на формах, работающих в модальном режиме. Поэтому самое примечательное свойство элемента управления отвечает за модальный результат, передаваемый окну, и отображаемый на поверхности кнопки пояснительный значок:

```
property Kind: TBitBtnKind;
type TBitBtnKind = (bkCustom, bkOK, bkCancel, bkHelp, bkYes, bkNo,
                    bkClose, bkAbort, bkRetry, bkIgnore, bkAll);
```

Каждому возможному значению `TBitBtnKind` поставлен в соответствие модальный результат (`ModalResult=bkCustom` результат `0`, `ModalResult=bkOk` результат `mrOk`, ... , `ModalResult=bkAll` результат `mrAll`). Для большей наглядности (рис. 18.3) кнопка одновременно снабжается значком.

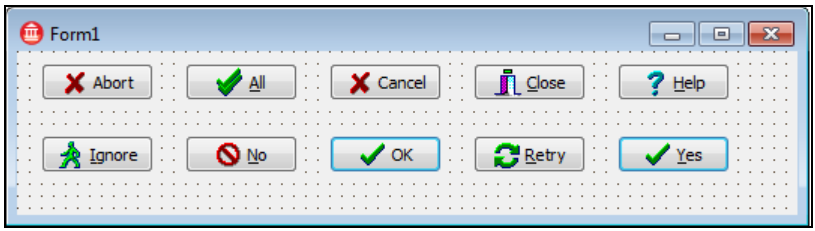


Рис. 18.3. Внешний вид кнопок `TBitBtn` с разными состояниями свойства `Kind`

Если вас не устраивает назначаемое по умолчанию изображение, то воспользуйтесь свойством

```
property Glyph: TBitmap;
```

Загрузите в свойство файл с небольшой картинкой, и кнопка покажет новое изображение. Имейте в виду, что кнопка одновременно может поддерживать до четырех изображений, каждое из которых соответствует определенному состоянию (кнопка не нажата; не активна; в момент нажатия; нажата).

Количество подключаемых значков определяется свойством

```
property NumGlyphs: TNumGlyphs;
type TNumGlyphs = 1..4;
```

Все картинки должны быть одна за другой "склеены" в единый файл.

По желанию разработчика местоположение рисунка на рабочей поверхности кнопки изменяется

```
property Layout: TButtonLayout; //по умолчанию blGlyphLeft
type TButtonLayout =
                (blGlyphLeft, blGlyphRight, blGlyphTop, blGlyphBottom);
```

Расстояние от края кнопки до границы значка определяется свойством

```
property Margin: Integer;
```

Дистанция между рисунком и текстом заголовка кнопки изменяется в свойстве

```
property Spacing: Integer;
```

Свойство, отвечающее за стиль кнопки

```
property Style: TButtonStyle;
type TButtonStyle = (bsAutoDetect, bsWin31, bsNew);
```

Это не что иное, как наследие середины 1990-х (времен перехода с Windows 3.1 на Win32), и предназначено для обеспечения обратной совместимости проектов различных версий Delphi.

Кнопка-флажок *TCheckBox*

Элемент управления `TCheckBox` (флажок) полезен в тех случаях, когда программе требуется узнать мнение пользователя по тому или иному вопросу. В своем выборе пользователь должен быть предельно лаконичен — обычно ему предоставлены только два варианта ответа: "да" (компонент отмечен флажком) или "нет" (флажок снят). Текущее состояние компонента оценивается свойством

```
property State: TCheckBoxState;
type TCheckBoxState = (cbUnchecked, //флажок не установлен
                       cbChecked,   //флажок установлен
                       cbGrayed);   //неопределенное состояние
```

Программист может запретить переводить флажок в неопределенное состояние. Для этого должно быть установлено в `false` свойство

```
property AllowGrayed: Boolean;//по умолчанию false
```

В случае если кнопка выбрана (State=cbChecked), свойство

```
property Checked: Boolean;
```

принимает значение true, в остальных случаях (State=cbChecked или cbGrayed) значение равно false.

Кнопки изменения значения *TUpDown*

Элемент управления TUpDown представляет собой объединение из двух кнопок — счетчик. Основной задачей элемента управления является пошаговое изменение (увеличение или уменьшение) целого значения. Текущее значение описывается ключевым для компонента свойством

```
property Position: SmallInt;
```

При щелчке по соответствующей кнопке значение свойства увеличивается или уменьшается с шагом, определенным в свойстве

```
property Increment: Integer; //по умолчанию 1
```

Диапазон допустимых значений, которые может принимать свойство Position, ограничивается свойствами

```
property Min: SmallInt; //минимальное значение
property Max: SmallInt; //максимальное значение
```

Если вы допускаете циклическую смену значения Position, установите в true свойство

```
property Wrap: Boolean;
```

Теперь при достижении максимального значения следующий щелчок по кнопке приращения присваивает свойству Position значение из свойства Min, и наоборот.

При установке в true свойства

```
property ArrowKeys: Boolean;
```

компонент TUpDown (находясь в фокусе ввода) получит возможность реагировать на нажатие клавиш управления курсором.

Элемент управления не обладает своим заголовком, но способен отображать текущее значение Position в ассоциированном с ним объекте. Связь с другим оконным элементом определяется свойством

```
property Associate: TWinControl;
```

В роли связанного объекта может выступать любой наследник класса TWinControl, но обычно в качестве такового применяют строку ввода (например, TEdit) либо статический текст (например, TStaticText).

Для повышения наглядности отображаемых цифровых значений предусмотрена возможность визуального разделения тысяч при отображении свойства Position в ассоциированном с элементом управления объекте. Для этого присвойте значение true свойству

```
property Thousands : Boolean;
```

Расположение компонента TUpDown слева или справа от связанного объекта определяется в свойстве

```
property AlignButton: TUDAlignButton; //слева udLeft или справа udRight
```

Кнопки элемента управления могут быть сориентированы по горизонтали или вертикали:

```
property Orientation: TUDOrientation;// udHorizontal или udVertical
```

Помимо классического для всех кнопок события `OnClick()` кнопка класса `TUpDown` может похвастаться парой специфичных событий: `OnChanging()` и `OnChangingEx()`. Оба события вызываются перед изменением свойства `Position` компонента.

```
property OnChanging: TUDChanginEvent;
type TUDChangingEvent = procedure(Sender: TObject;
                                  var AllowChange: Boolean) of object;
```

Управляя параметром `AllowChange`, программист разрешает (`true`) или запрещает (`false`) изменять состояние свойства `Position`. Еще более богатыми возможностями обладает расширенная версия события

```
property OnChangingEx: TUDChanginEventEx;
type TUDChangingEventEx = procedure (Sender: TObject;
                          var AllowChange: Boolean; NewValue: SmallInt;
                          Direction: TUpDownDirection) of object;
```

Здесь появились дополнительные параметры: новое потенциальное значение `NewValue` и направление изменения значения `Direction` (`updNone`, `updUp`, `updDown`).

Кнопка выбора *TRadioButton*

Кнопка выбора `TRadioButton` — переключатель — никогда не применяется сама по себе, ей всегда требуется хотя бы один однотипный напарник. Сгруппированные вместе кнопки начинают напоминать кнопки переключения диапазонов старого радиоприемника — в группе одновременно может быть выбрана только единственная кнопка.

> ***Замечание***
>
> Проще всего объединить кнопки выбора в группу, разместив их на панели `TPanel` или на панели группировки `TGroupBox`.

За состояние кнопки отвечает свойство

```
property Checked: Boolean;
```

Если кнопка выбрана, то свойство возвратит значение `true`, если нет — `false`.

С точки зрения оформления элемента управления незнакомым может показаться только свойство

```
property Alignment: TLeftRight;
type TLeftRight = taLeftJustify..taRightJustify;
```

Оно определяет, с какой стороны от кнопки будет располагаться поясняющая надпись из свойства `Caption`.

Группа переключателей *TRadioGroup*

Группа переключателей `TRadioGroup` представляет собой симбиоз панели группировки и кнопок выбора, но реализованных в рамках единого элемента управления. По умолчанию компонент не содержит ни одной кнопки. Для того чтобы группа переключателей стала

владельцем кнопок, ни в коем случае не надо тянуться к палитре компонентов, чтобы найти на ней кнопку TRadioButton. Вместо этого обращаемся к свойству

```
property Items: TStrings;
```

Во время проектирования добавление или удаление кнопок осуществляется в специальном редакторе — каждая вновь введенная строка добавляет на панель новую кнопку. Во время выполнения программы настройка списка осуществляется в соответствии с правилами работы с классом TStrings *(см. главу 10)*.

Для идентификации выбранной кнопки предназначено свойство

```
property ItemIndex: Integer;
```

Это свойство возвращает порядковый номер выбранной кнопки в списке Items. Если выбрана первая кнопка, то ее индекс равен 0, следующая кнопка — 1, последняя — Count-1.

Группа переключателей позволяет размещать кнопки в несколько колонок. Число колонок определяется свойством:

```
property Columns: Integer;
```

Стоит помнить об ограничении — количество колонок не должно превышать 16.

Группа кнопок *TButtonGroup*

Когда пользовательский интерфейс приложения предполагает наличие большого числа кнопок, то не следует упускать из виду элемент управления TButtonGroup, способный создавать группу кнопок. Для редактирования перечня кнопок проще всего обратиться к контекстному меню компонента, выбрав в нем пункт **Items Editor**. Щелчок по элементу меню вызовет встроенный редактор, позволяющий добавлять, удалять и изменять последовательность кнопок (рис. 18.4).

Отдельная кнопка описывается классом TGrpButtonItem, экземпляры кнопки хранятся в коллекции элементов TGrpButtonItems, доступ к которой осуществляется благодаря посредничеству свойства

```
property Items: TGrpButtonItems;
```

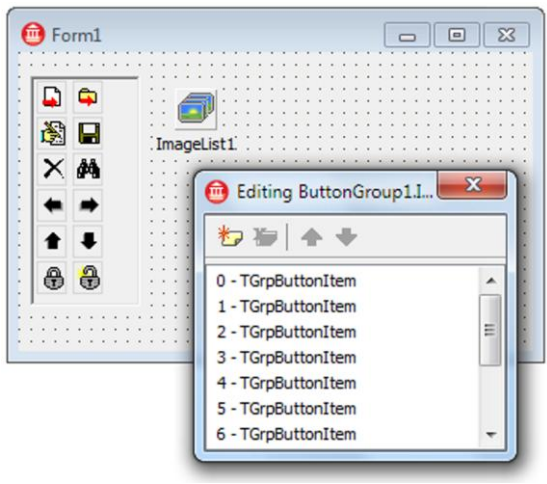


Рис. 18.4. Экранный снимок компонента TButtonGroup с редактором коллекции кнопок

Коллекция кнопок `TGrpButtonItems` является прямым наследником класса `TCollection` (класс рассмотрен в *главе 10*), поэтому в табл. 18.2 упомянуты лишь наиболее важные свойства и методы класса `TGrpButtonItems`.

Таблица 18.2. *Основные свойства и методы коллекции* `TGrpButtonItems`

Свойства и методы	Описание
`function Add: TGrpButtonItem;`	Добавление в коллекцию новой кнопки. Функция возвращает ссылку на экземпляр созданной кнопки
`function AddItem(Item: TGrpButtonItem; Index: Integer): TGrpButtonItem;`	Добавление новой кнопки в коллекцию `Item`, позиция кнопки определяется параметром `Index`. Функция возвращает ссылку на экземпляр созданной кнопки
`function Insert(Index: Integer): TGrpButtonItem;`	Вставка новой кнопки в позицию `Index` коллекции. Функция возвращает ссылку на экземпляр созданной кнопки
`property ButtonGroup: TButtonGroup;`	Ссылка на элемент управления, которому принадлежит коллекция
`property Count: Integer;`	Число кнопок в коллекции
`property Items[Index: Integer]: TGrpButtonItem;`	Обращение к отдельной кнопке по ее индексу
`procedure Clear;`	Полная очистка коллекции
`procedure Delete(Index: Integer);`	Удаление кнопки с индексом `Index`

Отдельная кнопка группы `TGrpButtonItem` обладает скромным набором свойств, наиболее существенные из них упомянуты в табл. 18.3.

Таблица 18.3. *Основные свойства кнопки* `TGrpButtonItem`

Свойства	Описание
`property Index: Integer;`	Индекс кнопки в коллекции
`property Caption:string;`	Заголовок кнопки
`property Data: TCustomData;`	Дополнительные данные, связанные с кнопкой
`property ImageIndex: TImageIndex;`	Индекс пиктограммы кнопки
`property ButtonGroup: TButtonGroup;`	Ссылка на элемент управления `TButtonGroup`, которому принадлежит кнопка
`property Collection: TGrpButtonItems;`	Ссылка на коллекцию (свойство `Items` группы), которой принадлежит кнопка

Наполнив коллекцию кнопками, программисту стоит уделить внимание опциям элемента управления

```
property ButtonOptions: TGrpButtonOptions;//по умолчанию gboShowCaptions
```

Опций всего четыре:

```
type TGrpButtonOptions = set of
   (gboAllowReorder, //пользователю разрешено переупорядочивать кнопки
    gboFullSize,     //устанавливаются максимальные размеры кнопок
    gboGroupStyle,   //кнопки наследуют стиль группы
    gboShowCaptions);//отображать заголовки кнопок
```

Режим `gboFullSize` весьма полезен для группы кнопок, работающих совместно с панелями категорий (элементом управления `TCategoryPanelGroup`). В этом случае можно построить пользовательский интерфейс, схожий с палитрой компонентов Delphi (рис. 18.5). Щелчок по заголовку панели раскрывает панель, на которой надо расположить нашего нового знакомого — элемент управления `TButtonGroup`.

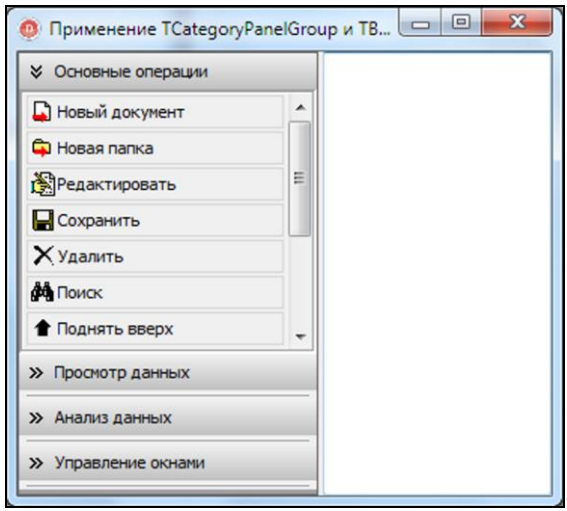


Рис. 18.5. Совместное применение `TButtonGroup` и `TCategoryPanelGroup`

Внешний вид группы кнопок существенно улучшится, если к элементу управления подключить контейнер с изображениями, для этого понадобится свойство

```
property Images: TCustomImageList;
```

Коллекция изображений должна быть заполнена значками, которые ассоциируются с отдельными кнопками. Для этого на уровне кнопки объявлено свойство `ImageIndex`.

> ***Замечание***
>
> Если логика работы приложения будет позволять пользователю буксировать группу кнопок (drag and dock), то стоит обратить внимание на свойство `DragImageList`, предназначенное для подключения набора значков, которые отображаются во время буксировки. Индекс значка назначается свойством `DragIndex`.

Геометрические размеры кнопок определяются парой свойств:

```
property ButtonHeight: Integer;//по умолчанию 24
property ButtonWidth: Integer; //по умолчанию 24
```

Для обращения к кнопке из программного кода можно воспользоваться свойством

```
property ItemIndex: Integer;
```

Кнопка с указанным индексом подлежит выделению.

Если геометрические размеры элемента управления не позволяют одновременно отобразить все кнопки, то для быстрого перемещения к кнопке с заданным индексом следует вызвать метод

```
procedure ScrollIntoView(const Index: Integer);
```

Перечень ключевых событий группы кнопок предложен в табл. 18.4.

Таблица 18.4. *Основные события группы кнопок* `TButtonGroup`

События	Описание
`property OnButtonClicked: TGrpButtonEvent;` `type TGrpButtonEvent = procedure(Sender: TObject; Index: Integer) of object;`	Щелчок по одной из кнопок группы, в параметре `Index` возвращается индекс кнопки
`property OnHotButton: TGrpButtonEvent;`	Генерируется во время перемещения указателя мыши над кнопками, в параметре `Index` — индекс кнопки
`property OnReorderButton: TGrpButtonReorderEvent;` `type TGrpButtonReorderEvent = procedure(Sender: TObject; OldIndex, NewIndex: Integer) of object;`	Переупорядочивание кнопок в коллекции (выбран стиль `gboAllowReorder`). Старый и новый индексы кнопки передаются в параметры `OldIndex` и `NewIndex`
`property OnBeforeDrawButton: TGrpButtonDrawEvent;` `type TGrpButtonDrawEvent = procedure(Sender: TObject; Index: Integer; Canvas: TCanvas; Rect: TRect; State: TButtonDrawState) of object;`	Событие вызывается перед началом операции графического вывода кнопки
`property OnDrawButton: TGrpButtonDrawEvent;`	Прорисовка кнопки
`property OnDrawIcon: TGrpButtonDrawIconEvent;` `type TGrpButtonDrawIconEvent = procedure(Sender: TObject; Index: Integer;Canvas: TCanvas; Rect: TRect; State: TButtonDrawState; var TextOffset: Integer) of object;`	Прорисовка значка кнопки
`property OnAfterDrawButton: TGrpButtonDrawEvent;`	Вызывается по завершению графического вывода кнопки

Категории кнопок *TCategoryButtons*

Мы подошли к изучению наиболее интересного элемента управления — кнопок, упорядоченных по категориям — `TCategoryButtons`. По своим возможностям `TCategoryButtons` ничем не уступает таким серьезным конкурентам, как группа кнопок (компонент `TButtonGroup`), размещенная на панели категорий (компонент `TCategoryPanelGroup`). Только на этот раз все задачи решаются не за счет двух, а силами единственного компонента — наш новый знакомый позволяет создавать отдельные панели (которые мы станем называть категориями) и формировать в них группы кнопок.

Конструктивно элемент управления `TCategoryButtons` представляет собой симбиоз нескольких коллекций и хранящихся в этих коллекциях элементов. Отдельная панель категорий реализуется классом `TButtonCategory` и содержится в коллекции панелей `TButtonCategories`. В свою очередь, каждая панель категорий содержит коллекцию кнопок,

описываемую классом `TButtonCollection`, отдельная кнопка этой коллекции представлена классом `TButtonItem`.

Доступ к коллекции категорий обеспечивает свойство

```
property Categories: TButtonCategories;
```

Щелчок по свойству `Categories` вызывает окно редактора панелей категорий, позволяющего добавлять, удалять и упорядочивать панели. В свою очередь, выделив в редакторе панель `TButtonCategory`, вы сможете найти в Инспекторе объектов свойство

```
property Items:TButtonCollection;
```

Благодаря такой многоходовой операции вы получите доступ к коллекции кнопок `TButtonCollection` этой панели (рис. 18.6).

Внешний вид и поведение панелей категорий и принадлежащих им кнопок в первую очередь определяется свойством

```
property ButtonOptions: TCatButtonOptions;
```

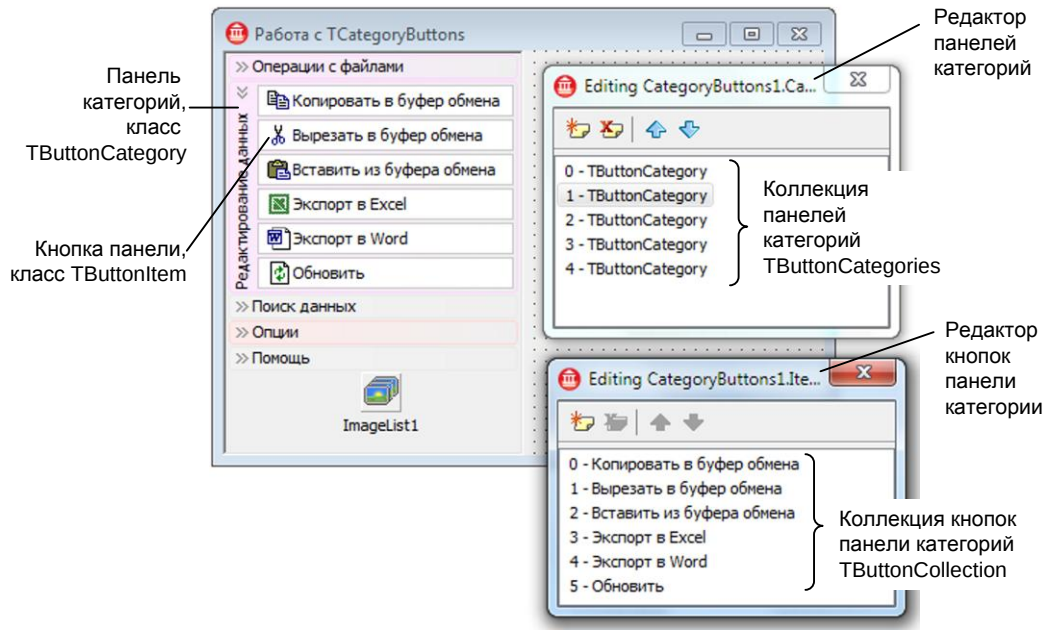


Рис. 18.6. Внешний вид элемента управления `TCategoryButtons`

Перечень доступных опций описывается множеством `TCatButtonOptions`.

```
type TCatButtonOptions = set of (
    boAllowReorder, //пользователю разрешено переупорядочивать кнопки
    boAllowCopyingButtons, //разрешено копировать кнопки
    boFullSize,            //полноразмерные кнопки
    boGradientFill,        //используется градиентная заливка
    boShowCaptions,        //отображать заголовки
    boVerticalCategoryCaptions,//вертикальная ориентация заголовков
    boBoldCaptions, //заголовки выводятся полужирным шрифтом
```

```
    boUsePlusMinus, // "-" — свернуть, "+" — развернуть панель
    boCaptionOnlyBorder);{влияет на вывод заголовка
                          при градиентной заливке}
```

Если в состав опций не входит опция `[boFullSize]`, то размеры кнопок устанавливаются свойствами

```
property ButtonHeight: Integer; //по умолчанию 24
property ButtonWidth: Integer;  //по умолчанию 24
```

Для того чтобы кнопки смогли отображать на своей поверхности значки, к элементу управления с помощью свойства

```
property Images: TCustomImageList;
```

следует присоединить набор изображений — компонент `TImageList`.

Текущая панель категорий доступна благодаря свойству

```
property CurrentCategory: TButtonCategory;
```

Элемент управления владеет информацией о том, какая из кнопок находится в фокусе ввода

```
property FocusedItem: TBaseItem;
```

и какая кнопка выбрана пользователем

```
property SelectedItem: TBaseItem;
```

Так как основным посредником между элементом управления и пользователем выступает мышь, большой перечень методов компонента посвящен получению ссылки на панель или кнопку по экранным координатам указателя мыши. Например, доступ к панели по координатам (`X`, `Y`) предоставляет функция

```
function GetCategoryAt(X, Y: Integer): TButtonCategory;
```

Передав координаты (`X`, `Y`) в метод

```
function GetButtonAt(X, Y: Integer;
                     Category: TButtonCategory = nil): TButtonItem;
```

мы получим ссылку на кнопку. При необходимости с помощью параметра `Category` можно конкретизировать, какой именно панели категорий принадлежит кнопка.

Наиболее универсален метод

```
procedure GetTargetAt(X, Y: Integer; var TargetButton: TButtonItem;
                      var TargetCategory: TButtonCategory);
```

Он позволяет одновременно идентифицировать кнопку и панель категорий.

При решении обратной задачи — получении экранных координат определенной панели или кнопки — следует обратиться к методам

```
function GetCategoryRect(const Category: TButtonCategory;
                         ButtonOnly: Boolean = False): TRect;
function GetButtonRect(const Item: TBaseItem): TRect;
```

Функции возвращают прямоугольные координаты указанного элемента.

По умолчанию предусмотрено горизонтальное размещение кнопок на поверхности панели, при желании кнопки можно повернуть на 90° с помощью свойства

```
property ButtonFlow: TCatButtonFlow;//по умолчанию cbfHorizontal
type TCatButtonFlow = (cbfVertical, cbfHorizontal)
```

Элемент управления `TCategoryButtons` обладает внушительным перечнем обработчиков событий, основные из них собраны в табл. 18.5.

Таблица 18.5. *Основные события кнопок категорий* `TCategoryButtons`

События	**Описание**
Выбор панели или кнопки	
property OnButtonClicked: TCatButtonEvent;	Щелчок по кнопке
property OnHotButton: TCatButtonEvent;	Появление указателя мыши над кнопкой
property OnCategoryClicked: TCatButtonCategoryEvent;	Щелчок по панели категорий
property OnCategoryCollapase: TCategoryCollapseEvent;	Свертывание панели категорий
property OnSelectedItemChange: TCatButtonEvent;	Пользователем выбран другой элемент
property OnSelectedCategoryChange: TCatButtonCategoryEvent;	Смена текущей панели, генерируется вне зависимости от свертывания или разворачивания панели
Редактирование заголовков панелей и кнопок	
property OnEditing: TCatButtonEditingEvent;	Генерируется в момент начала изменения текста в заголовке
property OnEdited: TCatButtonEditedEvent;	Редактирование текста в заголовке завершено
property OnCancelEdit: TCatButtonCancelEditEvent;	Отмена изменений в заголовке
Изменение положения панелей и кнопок (включена опция `boAllowReorder`)	
property OnReorderButton: TCatButtonReorderEvent;	Пользователь изменил положение кнопки
property OnReorderCategory: TCategoryReorderEvent;	Пользователь изменил положение панели
Копирование кнопки (включена опция `boAllowCopyingButtons`)	
property OnCopyButton: TCatButtonCopyEvent;	Копирование кнопки (например, в момент перетаскивания кнопки)
События управления графическим выводом	
property OnBeforeDrawButton: TCatButtonDrawEvent;	Начало перерисовки
property OnDrawButton: TCatButtonDrawEvent;	Вывод кнопки
property OnDrawIcon: TCatButtonDrawIconEvent;	Прорисовка изображения
property OnDrawText: TCatButtonDrawEvent;	Вывод текста
property OnAfterDrawButton: TCatButtonDrawEvent;	Завершение перерисовки

При проектировании современных приложений значительное внимание уделяется созданию привлекательному внешнему виду его элементов управления. Подтверждение тому — целых пять обработчиков событий (см. табл. 18.5), специализирующихся на управлении графическим выводом. Ко всему прочему у `TCategoryButtons` имеется целый арсенал

свойств и методов, оказывающих прямое влияние на оформление панелей и кнопок элемента управления. За определение цвета поверхности кнопки отвечают свойства

```
property RegularButtonColor: TColor; //кнопка в обычном состоянии
property SelectedButtonColor: TColor; //кнопка выбрана
property HotButtonColor: TColor; //над кнопкой находится указатель мыши
```

Управление градиентной заливкой осуществляют свойства

```
property BackgroundGradientColor: TColor; //фоновый цвет градиента
property BackgroundGradientDirection: TGradientDirection;//направление
property GradientDirection: TGradientDirection;//направление заливки
```

В табл. 18.6—18.9 представлены справочные сведения о работающих в составе элемента управления `TCategoryButtons` коллекциях и элементах коллекций.

***Таблица 18.6.** Основные характеристики коллекции панелей `TButtonCategories`*

Свойства и методы	Описание
`function Add: TButtonCategory;`	Создание и добавление в коллекцию панели
`function AddItem(Item: TButtonCategory; Index: Integer): TButtonCategory;`	Добавление в позицию `Index` коллекции панели `Item`
`function Insert(Index: Integer): TButtonCategory;`	Создание и добавление в позицию `Index` панели
`property Items[Index: Integer]: TButtonCategory;`	Доступ к отдельной панели по ее индексу
`function ItemIndex(const Caption: string): Integer;`	Возвращает индекс панели по ее заголовку
`property Count: Integer;`	Число панелей в коллекции
`property VisibleCount: Integer;`	Число видимых панелей
`procedure Delete(Index: Integer);`	Удаление панели с индексом `Index`
`procedure Clear;`	Очистка коллекции
`property CategoryButtons: TCategoryButtons;`	Ссылка на элемент управления, которому принадлежит коллекция

***Таблица 18.7.** Основные характеристики экземпляра панели категорий `TButtonCategory`*

Свойства	Описание
`property Caption: string;`	Текстовый заголовок категории
`property Data: TCustomData;`	Дополнительные данные категории
`property Collapsed: Boolean;`	В состоянии `true` панель категории свернута, `false` — развернута
`property Items: TButtonCollection;`	Доступ к коллекции кнопок данной категории
`property CategoryButtons: TCategoryButtons;`	Ссылка на элемент управления, которому принадлежит панель категории
`property Color: TColor;`	Цвет панели

Таблица 18.7 *(окончание)*

Свойства	Описание
`property GradientColor: TColor;`	Цвет градиента
`property TextColor: TColor;`	Цвет текста

Таблица 18.8. *Основные характеристики коллекции кнопок* `TButtonCollection`

Свойства и методы	Описание
`function Add: TButtonItem;`	Создание новой кнопки `TButtonItem` и добавление ее в коллекцию
`function AddItem(Item: TButtonItem; Index: Integer): TButtonItem;`	Добавление кнопки `Item` в коллекцию в позицию `Index`
`function Insert(Index: Integer): TButtonItem;`	Создание новой кнопки `TButtonItem` и вставка ее в коллекцию в позицию `Index`
`property Items[Index: Integer]: TButtonItem;`	Доступ к кнопке коллекции по ее индексу
`property Count: Integer;`	Число кнопок в коллекции
`procedure Delete(Index: Integer);`	Удаление кнопки с индексом `Index`
`procedure Clear;`	Полная очистка коллекции
`property Category: TButtonCategory;`	Ссылка на панель категорий `TButtonCategory`, которой принадлежит коллекция

Таблица 18.9. *Основные характеристики кнопки* `TButtonItem`

Свойства	Описание
`property Caption: string;`	Заголовок кнопки
`property ImageIndex: TImageIndex;`	Индекс отображаемого значка
`property CategoryButtons: TCategoryButtons;`	Ссылка на элемент управления
`property Category: TButtonCategory;`	Ссылка на панель, которой принадлежит кнопка

ГЛАВА 19

Меню приложения

Без всякого преувеличения можно сказать, что меню приложения выступает наиболее важной частью пользовательского интерфейса современной программы для Windows. По сравнению с большинством элементов управления, меню представляет одно из наиболее удачных дизайнерских решений программистов — его достоинство заключается в том, что, практически не занимая места на форме, меню способно предоставить в распоряжение пользователя весь функционал программы.

Различают два типа меню: главное (`TMainMenu`) и контекстное (`TPopupMenu`). Главное меню размещается сразу под заголовком формы и, как правило, на самом верхнем уровне содержит пункты **Файл**, **Правка**, **Окно** и **Помощь**. При выборе пункта верхнего уровня из него "выпадают" связанные с ним элементы нижнего уровня. В процессе проектирования программист обладает возможностью размещать на форме сколько угодно компонентов главного меню, однако после запуска приложение сможет отобразить только одно главное меню. Ограничений на численность контекстных меню у приложения нет. В отличие от главного меню, контекстное возникает на экране только после щелчка правой кнопки мыши по форме или элементу управления. Программисты стараются не перегружать этот тип меню избыточными пунктами и по негласной договоренности заполняют его операциями, относящимися именно к конкретному элементу управления, которому принадлежит это меню.

Компоненты меню вы обнаружите на странице стандартных (**Standard**) элементов управления палитры компонентов Delphi. С точки зрения иерархии наследования, как главное, так всплывающее меню берут начало от уже изученного нами класса `TComponent` (рис. 19.1). По сути оба элемента управления представляют собой контейнеры, обеспечивающие хранение и доступ к отдельным пунктам меню — элементам, построенным на основе класса `TMenuItem`. Компоненты `TMainMenu` и `TPopupMenu` снабжены специализированным редактором, значительно упрощающим процесс построения меню. Для вызова редактора меню достаточно дважды щелкнуть левой кнопкой мыши по компоненту меню (рис. 19.2) или выбрать свойство `Items` в Инспекторе объектов. Для создания пункта меню следует выделить пустую область курсором мыши и в поле `Caption` Инспектора объектов присвоить пункту название. Элементы меню второго уровня способны обладать подменю, для создания которого достаточно, щелкнув правой кнопкой мыши, в контекстном меню выбрать пункт **Create Submenu**. При необходимости можно переупорядочить пункты внутри меню — перетаскивая их мышью. Для удаления лишнего пункта просто нажмите клавишу <Del>.

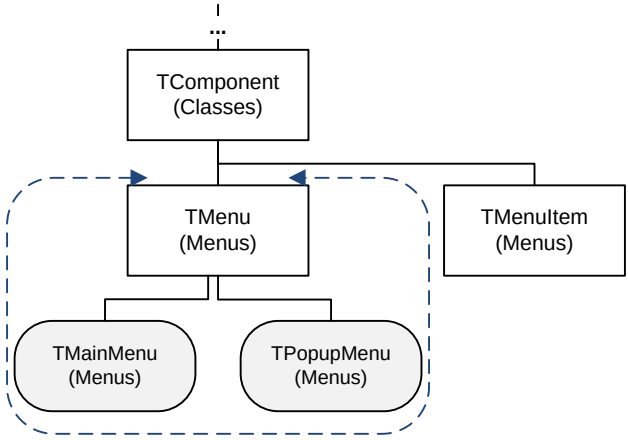


Рис. 19.1. Меню в иерархии VCL

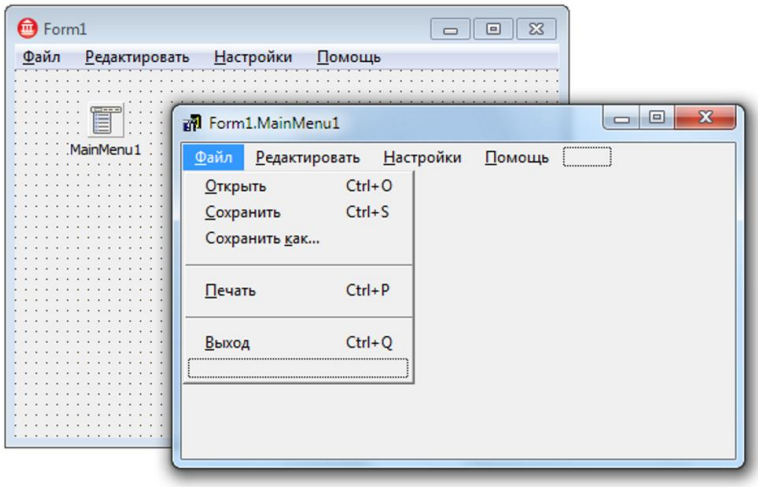


Рис. 19.2. Главное меню приложения и встроенный редактор меню

Опорный класс меню *TMenu*

Класс `TMenu` является каркасом для главного (`TMainMenu`) и контекстного (`TPopupMenu`) меню. Основное назначение класса — служить хранилищем для элементов меню — экземпляров класса `TMenuItem`. С этими целями в нем реализовано свойство

```
property Items: TMenuItem;
```

благодаря которому мы получаем доступ к тому или иному элементу меню. Для обращения к элементу из кода программы нам потребуется указать соответствующий ему индекс, например `MyItem:=MainMenu1.Items.Item[0]`.

При изменении состава заключенных в компонент элементов меню, при загрузке меню в память и при изменении влияющих на структуру меню свойств возникает событие

```
property OnChange: TMenuChangeEvent;
type TMenuChangeEvent = procedure(Sender: TObject; Source: TMenuItem;
                                  Rebuild: Boolean) of object;
```

Здесь `Source` указывает на элемент меню, чьи свойства изменяются. Если при этом параметр `Rebuild` возвращает значение `true`, то это свидетельствует о том, что осуществляются кардинальные изменения, связанные с удалением или созданием элементов меню.

По умолчанию меню старается помочь программисту при определении "горячих" клавиш, ускоряющих доступ к тому или иному элементу меню

```
property AutoHotkeys: TMenuAutoFlag; //по умолчанию maAutomatic
```

Если свойство установлено в `true`, то в заголовок `Caption` элемента меню будет автоматически подставлен символ амперсанда `&`. Иногда это мешает, в особенности если в программе создаются динамические пункты меню.

Как главное, так и контекстное меню позволяют сопоставлять с каждым пунктом меню отдельный значок. Для этого с помощью свойства

```
property Images: TCustomImageList;
```

следует подключить к компоненту коллекцию изображений `TImageList`.

Главное меню *TMainMenu*

Класс `TMainMenu` построен на основе опорного класса `TMenu` и специализируется на хранении и предоставлении доступа к элементарным пунктам меню (реализуемых из класса `TMenuItem`).

Из особенностей компонента главного меню стоит особо отметить механизм объединения. Объединение меню применяется:

- при работе с приложениями с интерфейсом MDI *(см. главу 15)*;
- при разработке приложения, обладающего несколькими окнами с индивидуальными меню;
- при внедрении объектов OLE (например, с применением компонента `TOleContainer`), в этом случае при старте сервера автоматизации последний вставляет свои элементы меню в главное меню приложения.

Последовательно рассмотрим все три направления использования механизма слияния меню. Первый случай — отображение в главном меню главной формы приложения MDI пунктов меню с названиями открытых дочерних окон. В этом случае благодаря объединению меню пользователь получает удобную возможность обращения к открытым дочерним окнам. Как не странно это покажется на первый взгляд, но для решения этой задачи надо воспользоваться не свойствами или методами класса `TMainMenu`, а свойством `WindowMenu` главной формы нашего проекта. Этим свойством определяется пункт меню `TMenuItem` (входящий в состав элементов главного меню), к которому будет "пристроен" список имен дочерних форм.

Если разрабатываемое приложение строится на основе интерфейса SDI, то для автоматического присоединения меню подчиненной формы к меню главной проверьте, чтобы свойство главного меню

```
property AutoMerge: Boolean; //по умолчанию false
```

было установлено в состояние `true`.

Если требуется полностью контролировать процесс слияния меню, то вместо услуг свойства AutoMerge стоит обратиться к методам

```
procedure Merge(Menu: TMainMenu);
procedure Unmerge(Menu: TMainMenu);
```

Эти процедуры предназначены для присоединения и отсоединения пунктов меню из текста программы. В параметре Menu следует передавать ссылку на главное меню проекта.

Контекстное меню *TPopupMenu*

Подавляющая часть построенных на основе класса TControl визуальных элементов управления обладает правом показа контекстного (всплывающего) меню. Для этого элементы оснащены свойством, предназначенным для подключения к ним компонента TPopupMenu

```
property PopupMenu: TPopupMenu;
```

В свою очередь компонент TPopupMenu также в состоянии идентифицировать "хозяина" при помощи своего свойства

```
property PopupComponent: TComponent;
```

Если это меню разделяется двумя и более компонентами, то я не возьмусь предсказать, на кого укажет данное свойство, если меню еще ни разу не вызывалось на экран. Если же меню отображалось, то в свойстве окажется ссылка на элемент управления, последнего воспользовавшегося контекстным меню.

По умолчанию всплывающее меню появляется рядом со своим владельцем после щелчка правой кнопкой мыши по его поверхности, а координаты вывода меню определяются текущим местоположением указателя мыши. Единственное, на что можно повлиять в этом случае, — так это определить, что мы предпочтем увидеть меню левее, правее или точно по центру относительно все того же указателя мыши.

```
property Alignment: TPopupAlignment; ;
type TPopupAlignment = (paLeft, paRight, paCenter);
```

При желании от автоматического вывода всплывающего меню на экран можно и отказаться. Для этого требуется установить в false его свойство

```
property AutoPopup: Boolean;
```

С этого момента управление показом меню переключается в ручной режим. Теперь для вызова меню потребуется вспомнить о существовании процедуры

```
procedure Popup(X, Y: Integer);
```

Управление отображением контекстного меню с помощью этого метода имеет одно существенное преимущество. В этом случае мы получаем право определить место вывода меню, указав экранные координаты X и Y явным образом.

В момент вывода меню на экран происходит событие

```
property OnPopup: TNotifyEvent;
```

которое зачастую применяется для последней настройки пунктов этого меню — управления их свойствами Visible, Enabled и Checked.

Процесс отображения пунктов меню на экране можно максимально осовременить, включив модные визуальные эффекты анимации:

```
property MenuAnimation: TMenuAnimation;
type TMenuAnimations = (maLeftToRight, maRightToLeft,
                        maTopToBottom, maBottomToTop, maNone);
```

Элемент меню *TMenuItem*

По большому счету главное меню и контекстное меню просто являются хранилищами элементов `TMenuItem`, поэтому вне зависимости от того, с каким типом меню вы собираетесь работать, основным помощником станет элемент меню — объект класса `TMenuItem`.

Как и большинство уже изученных элементов управления, пункт меню обладает заголовком, определяемым свойством

```
property Caption: string;
```

Но по сравнению со всеми остальными компонентами Delphi, у свойства `Caption` элемента меню есть существенные особенности. Во-первых, если в заголовок будет введен одинединственный символ "тире" (-), то пункт меню превратится в разделитель. Обратите внимание на рис. 19.2. Здесь между элементами **Сохранить как...** и **Печать** проведена сплошная горизонтальная черта. Это и есть пункт меню "разделитель". Он не способен нести функциональную нагрузку и реагировать на щелчок пользователя, его задача — улучшение наглядности приложения. Вторая особенность свойства `Caption` — возможность определения клавиш-акселераторов, ускоряющих доступ пользователя к пункту меню при одновременном нажатии <Alt> и клавиши-акселератора. На экране компьютера символ, соответствующий клавише-акселератору, выводится с подчеркиванием. Для назначения акселератора во время набора заголовка меню необходимо воспользоваться символом `&`. Амперсанд устанавливается перед символом, который программист предполагает сделать акселератором: "&Файл" или "&Открыть".

Помимо акселераторов, с каждым пунктом меню можно связать так называемые быстрые клавиши. Отличие быстрых клавиш от клавиш-акселераторов заключается в том, что выбор акселератора только позволит добраться до необходимого пункта меню, а нажатие комбинации быстрых клавиш заставит приложение выполнить сопоставленный с ними пункт меню. Быстрые клавиши определяются свойством

```
property ShortCut : TShortCut;
```

Если по какой-то причине потребуется сменить быстрые клавиши во время выполнения программы, то для этой цели разработана специальная функция, преобразующая комбинацию клавиш к виду `TShortCut`:

```
function ShortCut(Key: Word; Shift: TShiftState): TShortCut;
```

Здесь первым параметром передается код символа, а вторым — код нажатой служебной клавиши:

```
MenuItem1.ShortCut := ShortCut(Word('V'), [ssCtrl]);
```

Стоит подчеркнуть, что `ShortCut()` не является методом экземпляра меню `TMenuItem`. Это самостоятельная функция, она наряду с десятком других входит в обойму функций поддержки меню и объявлена в модуле `Menus`.

Для выяснения принадлежности элемента меню родительскому меню вызывают метод

```
function GetParentMenu: TMenu;
```

Для исключения ошибки назначения одинаковых акселераторов для различных пунктов меню устанавливайте в автоматический режим (maAutomatic) свойство

```
property AutoHotkeys: TMenuItemAutoFlag; //по умолчанию maAutomatic
type TMenuItemAutoFlag = (maAutomatic, maManual, maParent);
```

В этом случае, перед тем как меню будет отображено на экране, Delphi выявит и отключит дубликаты акселераторов. То же самое можно сделать, вызвав метод

```
function RethinkHotkeys: Boolean;
```

Если функция вернула true, то это признак того, что были найдены и исправлены ошибки.

Основное назначение пункта меню — среагировать на щелчок пользователя, в этом он практически ничем не отличается от обычных кнопок. Соответственно ничем не отличается и ключевой для элемента TMenuItem обработчик события

```
property OnClick: TNotifyEvent;
```

Умеющим работать с Windows API программистам полезно знать, что в момент щелчка по элементу меню в адрес окна-владельца меню отправляется сообщение WM_COMMAND. Для того чтобы оконная процедура была в состоянии выяснить, по какому именно пункту меню был произведен щелчок, в качестве старшего параметра сообщения передается уникальный числовой идентификатор элемента TMenuItem, который хранится в свойстве

```
property Command: Word; //только для чтения
```

Хотя у пункта меню не предусмотрена реакция на двойной щелчок, но, тем не менее, такой поступок пользователя совсем не исключается. Если произведен двойной щелчок по элементу меню, обладающему подменю, то в списке дочерних пунктов меню Delphi постарается найти элемент, чье свойство

```
property Default: Boolean;
```

установлено в true. Если такой элемент существует, то далее будет вызван его обработчик OnClick(). Другими словами, это вызываемый по умолчанию пункт.

Элемент меню в виде флажка

При необходимости отдельный пункт меню можно превратить в элемент управления, напоминающий флажок (TCheckBox) или кнопку выбора (TRadioButton). Таким чудесным возможностям TMenuItem в первую очередь обязан свойству

```
property Checked: Boolean;
```

Установив это поле в состояние true, мы увидим "галочку" слева от заголовка. Этим самым пункт меню сигнализирует нам, что он отмечен. Контролируя это свойство (листинг 19.1), программист получит превосходную возможность применять конструкцию if..then..else в обработчике события OnClick().

Листинг 19.1. Щелчок по элементу меню

```
procedure TForm1.MenuItem1Click(Sender: TObject);
begin
  if MenuItem1.Checked=true then ...  //операция 1
                            else ...; //операция 2
end;
```

Для того чтобы пункт меню при щелчке по нему автоматически помечался галочкой, следует перевести в `true` свойство:

```
property AutoCheck: Boolean;
```

в противном случае придется делать это вручную внутри события `OnClick()`. В простейшем случае это будет всего-навсего одна строка кода

```
MenuItem1.Checked:= NOT MenuItem1.Checked;
```

инвертирующая предыдущее состояние пункта меню.

Элементы меню в виде группы выбора

Для того чтобы научиться превращать несколько пунктов меню в группу выбора, рассмотрим крохотный пример (рис. 19.3). Создайте новый проект, разместите на нем компонент `TMainMenu`. Создайте пункт меню верхнего уровня с заголовком "Цвет формы" и четыре подчиненных ему пункта:

- элемент с заголовком "Стандартный цвет". Имя элемента — `miStandardColor`, свойство `tag=0`;
- элемент "Красный", `tag=1`;
- элемент "Синий", `tag=2`;
- элемент "Зелёный", `tag=3`.

Наша задача — по щелчку по пункту меню перекрашивать форму в соответствующий цвет. Теперь одновременно выделите все четыре только что созданных пункта меню и в Инспекторе объектов найдите свойство

```
property GroupIndex: Byte;
```

Это свойство предназначено для создания логических групп пунктов меню. По умолчанию каждый вновь создаваемый элемент `TMenuItem` не входит ни в одну группу (`GroupIndex=0`), но если атрибуту `GroupIndex` двух (или более) пунктов меню присвоить значение, отличное от нуля, то мы получим возможность объединять элементы в некоторые товарищества, что мы сейчас собственно и сделаем. Введите в это свойство любое отличное от нуля неотрицательное число, например 1.

Следующим шагом по превращению четырех пунктов меню в группу выбора будет установка в `true` свойства:

```
property RadioItem: Boolean;
```

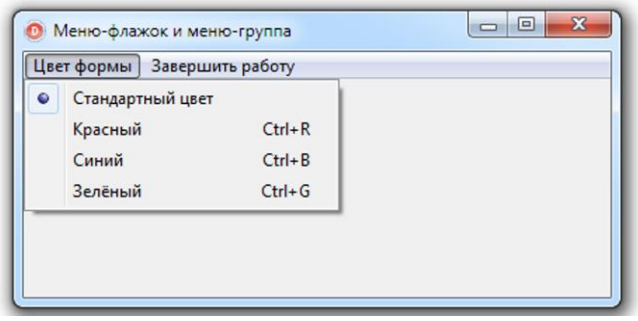


Рис. 19.3. Создание группы выбора из элементов меню

Перевод свойства RadioItem в значение true приведет к тому, что пункт меню станет вести себя аналогично компоненту TRadioButton. Для того чтобы выяснить, не отмечен ли элемент меню флажком, надо обратиться к уже знакомому нам свойству Checked, а Delphi позаботится о том, чтобы в одной группе не могло быть помечено более одного элемента TMenuItem одновременно. И в завершении убедитесь, что свойство AutoCheck всех четырех элементов установлено в true.

Нам осталось описать щелчок по любому из пунктов меню, например по miStandard (листинг 19.2), и сделать это событие общим для всех оставшихся пунктов меню, входящих в группу выбора.

Листинг 19.2. Работа с меню в режиме кнопок выбора

```
procedure TForm1.miStandardColorClick(Sender: TObject);
begin
  {красный — tag=1, синий — tag=2, зеленый — tag=3}
  case TMenuItem(Sender).Tag of
    1: Form1.Color:=clRed;
    2: Form1.Color:=clGreen;
    3: Form1.Color:=clBlue;
    else Form1.Color:=clBtnFace;
  end;
end;
```

Родительские и дочерние элементы меню

Каждый дочерний пункт меню знает, кому он принадлежит и свой индекс в списке родительского элемента:

```
property Parent: TMenuItem; //родитель
property MenuIndex: Integer;//индекс
```

Единственное исключение при определении родительского пункта возникает, если элемент меню является пунктом самого верхнего уровня. Свойство MenuIndex сослужит отличную службу, если необходимо переупорядочить пункты меню. Например, для перемещения элемента меню в начало списка присвойте этому свойству нулевое значение.

Если у элемента есть подчиненные пункты меню, то доступ к ним мы получим, обратившись к свойству Items родительского элемента.

```
property Items[Index: Integer]: TMenuItem;
```

Здесь хранится список всех дочерних элементов меню, нам надо только выбрать его порядковый индекс.

Информацию о числе дочерних элементов меню предоставит свойство

```
property Count: Integer; //только для чтения
```

Если родительский пункт меню захочет выяснить индекс пункта меню в своем списке Items, то для этой цели стоит обратиться к методу

```
function IndexOf(Item: TMenuItem): Integer;
```

Если меню не входит в подменю родительского элемента, то функция вернет –1.

Для поиска пункта меню по его заголовку применяйте функцию

```
function Find(ACaption: string): TMenuItem;
```

Если вызов метода завершился успехом, то он вернет ссылку на найденный пункт меню, иначе получаем пустышку — неопределенный указатель `nil`.

Присвоение элементам меню значков

Для придания пункту меню более изысканного вида рядом с заголовком разрешается расположить небольшую картинку. Вариантов решения этой задачи два. Простейший из них сводится к элементарной загрузке изображения в свойство

```
property Bitmap: TBitmap;
```

Второй вариант более рационален. Он заключается в подключении к родительскому компоненту `TMainMenu` (`TPopupMenu`) коллекции изображений — компонента `TImageList`. Если мы пошли по такому пути, то при помощи свойства

```
property ImageIndex: TImageIndex;
```

сможем выбирать любое изображение из коллекции по его индексу и отображать его слева от заголовка пункта меню.

Допускается назначить отдельную коллекцию картинок всем пунктам подменю. Для этого у владельца подменю надо найти свойство

```
property SubMenuImages: TCustomImageList;
```

Динамическое создание элементов меню

Как и все компоненты коллекции VCL, пункт меню вооружен своим конструктором `Create()` и деструктором `Destroy()`.

При создании нового экземпляра класса `TMenuItem` конструктор делает пункт меню видимым (`Visible=true`) и включенным (`Enabled=True`). В качестве владельца нового элемента `MenuItem` по возможности следует назначать пункт меню, к которому мы рассчитываем подключить новый элемент.

Операция присоединения нового пункта меню осуществляется одним из следующих методов:

```
procedure Add(Item: TMenuItem); overload;
procedure Add(const AItems: array of TMenuItem); overload;
procedure Insert(Index: Integer; Item: TMenuItem);
```

Первые два метода перегружаемые и поэтому называются одинаково — `Add()`. Разница между ними в том, что первый из них добавляет только один элемент `Item`, а второй способен подключить целый массив пунктов меню `AItems`. Если методы `Add()` присоединяют новые пункты меню к концу списка, то метод `Insert()` вставит новый пункт меню `Item` в позицию, определяемую параметром `Index`.

Предположим, что мы разрабатываем приложение, аналогичное по своим функциональным возможностям программе Блокнот из состава Windows. Нам необходимо научить наш проект запоминать, какие текстовые файлы обрабатывались последними, и при повторном запуске приложения создавать пункты меню, хранящие ссылку на эти файлы.

Перечень ссылок проще всего сохранить в обычном текстовом файле. В этот файл можно заносить обычную строку с путем к файлу после его чтения или сохранения. В момент следующего запуска приложения нам следует прочитать этот файл и динамически создать дополнительные пункты меню (листинг 19.3).

Листинг 19.3. Динамическое создание элементов меню

```
procedure TForm1.FormShow(Sender: TObject);
const rl='recentlinks.rec'; //имя файла со ссылками
var   sl:TStringList;
      i:integer;
      mi:TMenuItem;
begin
  MainMenu1.AutoHotkeys:=maManual; {откажемся от помощи
                                    в создании "горячих" клавиш}
  if FileExists(rl)=false then exit; //если нет файла со ссылками — выход
  try
    sl:=TStringList.Create;   //создали список строк
    sl.Duplicates:=dupIgnore; //запрет на дубликаты строк
    sl.LoadFromFile(rl);      //загрузим данные из файла
    for i :=0 to sl.Count-1 do //для каждой строки создаем TMenuItem
      if FileExists(sl.Strings[i])=true then //контроль за ссылкой
      begin
        mi:=TMenuItem.Create(MainMenu.Items[0]);
        mi.Caption:=sl.Strings[i];
        MainMenu.Items[0].Add(mi);
        {этой процедуры пока нет, см. листинг 19.4
        mi.OnClick:=RecentFilesMenuItemClick;//щелчок по меню }
      end;
  finally
    sl.Free; //список строк больше не нужен
  end;
end;
```

Приложение создаст ровно столько пунктов меню, сколько окажется текстовых строк с именами файлов в целевом файле `rl='recentlinks.rec'`, и присоединит их к самому первому элементу главного меню приложения `MainMenu.Items[0]`. Имя и путь к файлу заносятся в заголовки новых элементов меню. Для того чтобы щелчок по вновь созданному пункту меню загружал текстовый файл, нам придется написать еще одну процедуру, описывающую реакцию на событие `OnClick` (листинг 19.4).

Листинг 19.4. Динамическое создание элементов меню

```
private
  procedure RecentFilesMenuItemClick(Sender: TObject);
  //...
  procedure TForm1.RecentFilesMenuItemClick(Sender: TObject);
  begin
    Memo1.Lines.Clear;
    Memo1.Lines.LoadFromFile(TMenuItem(Sender).Caption);
  end;
```

Удаление элементов меню

Для удаления дочернего меню из списка нам опять понадобится его индекс и метод

```
procedure Delete(Index: Integer);
```

Альтернативный вариант удаления пункта меню предоставляет

```
procedure Remove(Item: TMenuItem);
```

Здесь вместо индекса требуется передать прямую ссылку на удаляемый пункт. Однако эти методы не уничтожают элемент меню, а лишь отнимают его у родительского компонента; для физического удаления пункта используйте метод `Free()`.

Наиболее кардинальное решение предлагает метод

```
procedure Clear;
```

Он расправится сразу со всеми дочерними пунктами меню, причем в этом случае явный вызов метода `Free()` не требуется — все пункты меню самостоятельно освободят занимаемую память.

Элементы-разделители

В меню, перенасыщенном командами, пункты меню формируют столбцы неоправданно больших размеров. В таких случаях весьма полезным окажется свойство

```
property Break: TMenuBreak;
type TMenuBreak = (mbNone, mbBreak, mbBarBreak);
```

Если у одного из пунктов меню (находящемуся примерно в середине столбца) установить это свойство в `mbBreak`, то следующие за ним элементы меню на экране монитора создадут еще один столбец. Если свойство принимает значение `mbBarBreak`, то столбцы разделятся вертикальной чертой. Появление нового столбца — только визуальный эффект, принадлежность элементов меню в этом случае не меняется.

Для визуального объединения нескольких пунктов меню в группу или для отделения одного элемента меню от другого программисты часто применяют пункты-разделители. Для этого во время разработки программы достаточно в свойство `Caption` поместить один-единственный символ "тире" (-). Во время выполнения программы задачу вставки линий-разделителей решают две пары функций:

```
function InsertNewLineBefore(AItem: TMenuItem): Integer;
function InsertNewLineAfter(AItem: TMenuItem): Integer;
```

и

```
function NewTopLine: Integer;
function NewBottomLine: Integer;
```

Метод `InsertNewLineBefore()` вставит разделитель перед пунктом меню `AItem`, а метод `InsertNewLineAfter()` — после. Пара функций `NewTopLine` и `NewBottomLine` обычно используется в случаях создания подменю во время выполнения программы. Все методы возвращают индекс элемента-сепаратора. При вставке разделителей программным способом разработчики Delphi рекомендуют устанавливать в автоматический режим (`maAutomatic`) свойство

```
property AutoLineReduction: TMenuItemAutoFlag;
TMenuItemAutoFlag = (maAutomatic, maManual, maParent);
```

В этом случае ответственность за удаление лишних разделителей возьмет на себя Delphi. Например, под ненужными разделителями Delphi понимает следующие подряд два разделителя, появление разделителя на самой первой или самой последней позиции подменю и т. п. Однако если вы отказались от автоматической чистки разделителей, то для борьбы с ненужными пунктами меню превосходно приспособлен метод

```
function RethinkLines: Boolean;
```

Если функция успешно справится с этой задачей, то она возвратит значение `true`.

Если пункт меню является обыкновенной разделительной линией, то при вызове метода

```
function IsLine: Boolean;
```

последний вернет значение `true`.

Особенности прорисовки пункта меню

Элемент меню способен реагировать на четыре события. Безусловно, основным из них считается щелчок `OnClick()` — свидетельство выбора этого пункта меню пользователем, и с ним мы уже знакомы. Все остальные обработчики предоставляют дополнительный функционал по прорисовке этого элемента меню. Простейший из них

```
property OnDrawItem: TMenuDrawItemEvent;
type TMenuDrawItemEvent = procedure (Sender: TObject; ACanvas: TCanvas;
                          ARect: TRect; Selected: Boolean) of object;
```

Здесь: `ACanvas` — холст пункта меню; `ARect` — координаты границ холста, доступные для перерисовки; параметр `Selected` сигнализирует, выбран данный пункт меню или нет.

> ***ЗАМЕЧАНИЕ***
>
> Для активации событий перерисовки следует установить в состояние `true` свойство `OwnerDraw` родительского компонента (владельца пункта меню).

В качестве примера использования `OnDrawItem()` могу предложить строки кода из листинга 19.5.

Листинг 19.5. Индивидуальный графический вывод элемента меню

```
procedure TForm1.MenuItem1DrawItem(Sender: TObject; ACanvas: TCanvas;
                                   ARect: TRect; Selected: Boolean);
begin
  with ACanvas do
  begin
    FillRect(aRect);
    if Selected=True then Font.Style:=Font.Style+[fsUnderline]
                     else Font.Style:=Font.Style-[fsUnderline];
    TextOut(aRect.Left,aRect.Top, (Sender as TMenuItem).Caption);
  end;
end;
```

Пункт меню проверяет, выделен он пользователем или нет. В первом случае заголовок пункта меню подчеркивается, иначе выводится обычным шрифтом.

Второй способ прорисовки пункта меню обладает более богатыми возможностями.

```
property OnAdvancedDrawItem: TAdvancedMenuDrawItemEvent;
type TAdvancedMenuDrawItemEvent = procedure (Sender: TObject;
       ACanvas: TCanvas; ARect: TRect; State: TOwnerDrawState) of object;
```

Расширенные возможности обеспечиваются наличием параметра `State`, сигнализирующего текущее состояние данного пункта меню. Теперь мы можем узнать не только о том, выделен этот пункт меню или нет, но проконтролировать, отмечен ли он "галочкой", активен или пассивен и т. д.

```
type TOwnerDrawState = set of (odSelected, odGrayed, odDisabled,
    odChecked, odFocused, odDefault, odHotLight, odInactive, odNoAccel,
    odNoFocusRect, odReserved1, odReserved2, odComboBoxEdit);
```

И наконец, третий обработчик события, косвенно связанный с прорисовкой пункта меню, решает задачу по динамическому изменению размеров пункта меню.

```
property OnMeasureItem: TMenuMeasureItemEvent;
type TMenuMeasureItemEvent = procedure (Sender: TObject;
                ACanvas: TCanvas; var Width, Height: Integer) of object;
```

Естественно, в этом случае ключевыми параметрами будут ширина `Width` и высота `Height` пункта меню.

ГЛАВА 20

Управление приложением с помощью команд

Пользовательский интерфейс хорошо продуманного приложения позволяет пользователю компьютера прийти к одному и тому же результату абсолютно разными путями. Например, для сохранения проекта Delphi программист может остановить свой выбор, по крайней мере, на одном из трех возможных вариантов действий: выбрать пункт главного меню **File | Save All**; щелкнуть по соответствующей кнопке на панели инструментов; нажать сопоставленную операции сохранения проекта комбинацию быстрых клавиш <Shift>+<Ctrl>+<All>. Наличие трех способов сохранения проекта не означает, что создатели IDE Delphi во время написания среды проектирования трижды повторили одни и те же строки кода. Такое решение было бы нелепым. На самом деле, для сохранения проекта была создана одна-единственная процедура, вызов которой осуществляется несколькими способами. В простейшем случае программистам Embarcadero было достаточно описать щелчок по пункту меню и затем подключить к этому событию все остальные элементы управления.

Обратите внимание на еще одну особенность поведения элементов управления визуальной среды проектирования Delphi. После полного сохранения проекта отвечающий за эту операцию элемент меню и кнопка элементов управления перестают быть активными — это свидетельствует о том, что все изменения в исходном коде успешно отправлены на жесткий диск компьютера. Но как только программист внесет малейшее исправление в листинг программы, соответствующие элементы управления вновь перейдут в режим готовности выполнить свою задачу — сохранить изменения в коде.

Элементы пользовательского интерфейса современного приложения должны не просто обладать способностью вызывать ту или иную процедуру, но и уметь отражать текущую обстановку. Для достижения поставленной цели в состав VCL были введены командные объекты, основу которых составляет класс `TBasicAction`. В этой главе мы познакомимся с наиболее распространенной реализацией команды — классом `TAction`.

Командный объект `TAction` создается и хранится в специализированном контейнере, в роли которого имеет право выступать список команд `TActionList` или менеджер команд `TActionManager`. Список команд `TActionList` обычно задействуется в приложениях, интерфейс которых строится на основе обычных компонентов меню (`TMainMenu` и `TPopupMenu`) и стандартных элементах управления (кнопок `TButton`, панелей инструментов `TToolBar` и т. п.). Более совершенный менеджер команд `TActionManager` обладает всеми возможностями своего коллеги и кроме этого способен взаимодействовать со специализированными командными панелями: инструментальной панелью `TActionToolBar` и панелями меню `TActionMainMenuBar` и `TPopupActionBar` (рис. 20.1).

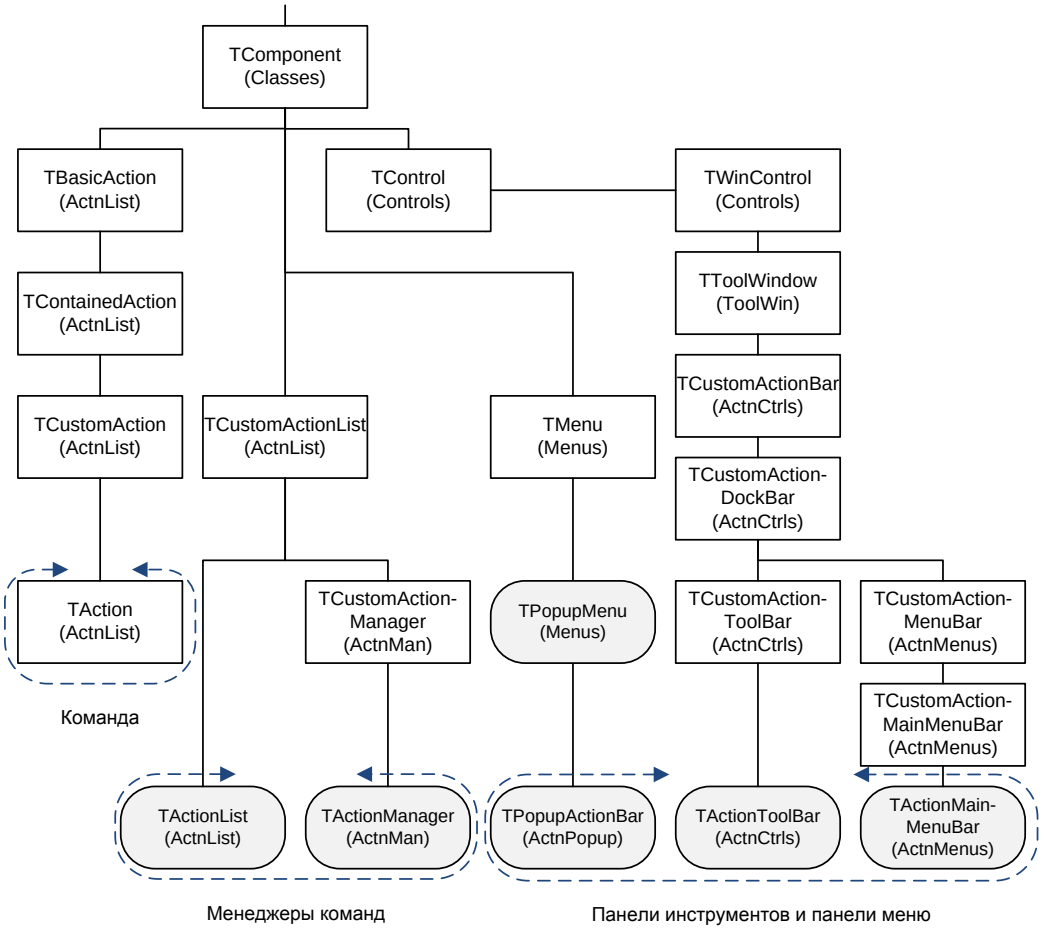


Рис. 20.1. Командный объект, менеджеры команд и панели команд в иерархии VCL

Команда *TAction*

Наиболее востребованный тип командного объекта создается на основе класса `TAction`. Кроме решения основной задачи, поставленной перед командой (открытие файла, сохранение данных, вызов метода и т. п.), команда выполняет ряд дополнительных заданий:

- централизация управляющего кода в рамках события `OnExecute()`;
- предоставление связанным с командой элементам управления права вызова управляющего кода;
- установка подключенных к команде элементов управления в актуальное состояние.

Для того чтобы сразу ощутить все преимущества, которые приобретает приложение, разработанное на основе командных объектов, рассмотрим небольшой пример. Создайте новый проект и на его главной форме расположите следующие элементы управления:

- список команд `TActionList` (или менеджер команд `TActionManager`);
- главное меню `TMainMenu`;

- контекстное меню `TPopupMenu`;
- три кнопки любого типа, например `TButton` или кнопки с инструментальной панели `TToolBar`;
- контейнер изображений `TImageList`;
- многострочный редактор `TMemo`.

Наша задача — написать код, позволяющий копировать (или вырезать) текст из многострочного редактора в буфер обмена и вставлять текст из буфера в редактор.

Дважды щелкните по менеджеру команд и трижды нажмите кнопку **New Action** в появившемся редакторе. Настройте свойства тройки новых команд:

- `Name='acCopy'; Caption='Копировать'; ShortCut='Ctrl+C';`
- `Name='acCut'; Caption='Вырезать'; ShortCut='Ctrl+X';`
- `Name='acPaste'; Caption='Вставить'; ShortCut='Ctrl+V'.`

Загрузите в контейнер изображений три файла с графическими образами, отражающими операции копирования, вырезания и вставки текста из буфера обмена. Воспользовавшись свойством `ImageList` компонента `ActionList1`, подключите контейнер изображений к менеджеру команд, затем повторите операцию подключения контейнера для компонентов меню. С этого момента вы получаете возможность сопоставить пиктограмму с каждой из команд (рис. 20.2).

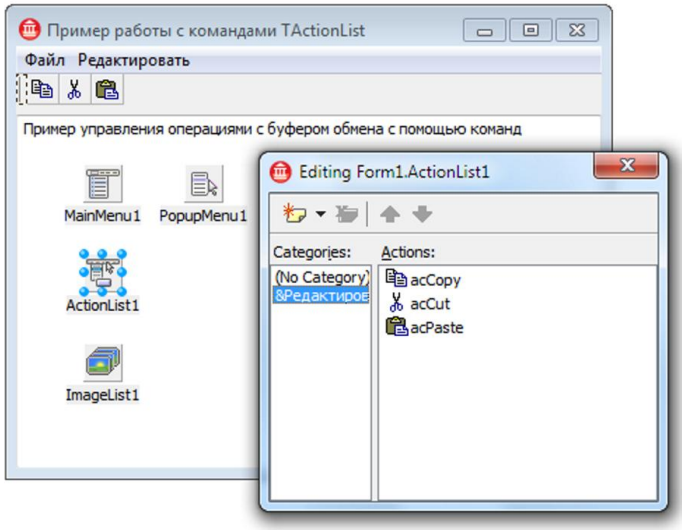


Рис. 20.2. Внешний вид приложения с открытым редактором списка команд

Мы подошли к самому главному — опишем управляющий код в событиях `OnExecute()` каждой из команд (листинг 20.1).

Листинг 20.1. Выполнение команд, событие `OnExecute()`

```
procedure TForm1.acCopyExecute(Sender: TObject);
begin
  Memo1.CopyToClipboard; //копируем текст в буфер
end;
```

```
procedure TForm1.acCutExecute(Sender: TObject);
begin
  Memo1.CutToClipboard; //вырезаем текст в буфер
end;

procedure TForm1.acPasteExecute(Sender: TObject);
begin
  Memo1.PasteFromClipboard; //забираем текст из буфера
end;
```

Создайте три пустых элемента в главном и всплывающем меню проекта и, воспользовавшись свойством `Action`, подключите пункты меню к командам. Сразу после установления связи между командой и пунктом меню с последним происходят чудесные метаморфозы. Заголовок, всплывающая подсказка, значок пункта меню переключаются на показ соответствующих свойств ассоциированного командного объекта. Но самое главное в том, что с этого момента щелчок по пункту меню станет вызывать метод `OnExecute()` связанной с ним команды. Таким образом, вместо "размазывания" управляющего кода по многочисленным элементам управления пользовательского интерфейса нашего приложения он сосредотачивается в руках командного объекта, а все подключенные к нему элементы управления получают право на вызов этого кода.

> ***ЗАМЕЧАНИЕ***
>
> Командные объекты могут быть легко подключены к большинству элементов управления VCL, необходимым условием подключения является наличие у элемента управления (пункта меню, кнопки) свойства `Action`.

Нам осталось сделать последний важный штрих — научить наши команды оценивать окружающую обстановку. Например, команды копирования и вырезания текста в буфер обмена не могут быть активны до тех пор, пока в текстовом редакторе не выделен ни один символ. Команда вставки текста из буфера обмена в поле редактирования должна активироваться только в том случае, если в буфере находится текстовая строка. Для установки командных объектов в актуальное состояние предназначены их события `OnUpdate()` (листинг 20.2).

Листинг 20.2. Установка команд в актуальное состояние, событие `OnUpdate()`

```
uses clipbrd; //модуль для работы с буфером обмена
//...
procedure TForm1.acCopyUpdate(Sender: TObject);
begin
  acCopy.Enabled:=(Memo1.SelLength>0);
end;

procedure TForm1.acCutUpdate(Sender: TObject);
begin
  acCut.Enabled:=(Memo1.SelLength>0);
end;

procedure TForm1.acPasteUpdate(Sender: TObject);
begin
  acPaste.Enabled:= (Clipboard.HasFormat(CF_TEXT))
                    and (Clipboard.AsText<>#0);
end;
```

Демонстрационный проект готов. Запустите приложение и проверьте поведение пунктов меню и кнопок.

Связь с элементом управления

Теперь мы знаем, что любой командный объект умеет взаимодействовать со стандартными элементами управления, берущими свое начало от класса `TControl`. В первую очередь к таким элементам относятся пункты меню `TMenuItem` и различного рода кнопки (`TButton`, `TSpeedButton`, `TToolButton` и т. д.). Кроме того, в стенах Embarcadero подготовлен ряд специфичных командных панелей (`TActionMainMenuBar`, `TPopupActionBar` и `TActionToolBar`), предназначенных только для совместной работы с командами.

Внешним признаком того, что элемент управления не прочь связать свою судьбу с командным объектом, служит наличие свойства

```
property Action : TBasicAction;
```

Свойство доступно в Инспекторе объектов, так что для подключения элемента управления к команде программисту достаточно пару раз щелкнуть кнопкой мыши. Визуально появление связи отразится на свойствах `Caption`, `Checked`, `Enabled`, `HelpContext`, `Hint`, `ImageIndex`, `ShortCut` и `Visible` нашего элемента управления. Содержимое перечисленных полей обновится в соответствии со значениями в аналогичных свойствах командного объекта `TAction`. Но самое главное, что с этого момента элемент управления получит право на вызов метода `Execute()` команды. Как правило, для этого автоматически переопределяется обработчик события `OnClick()`. Так что теперь щелчок по кнопке или пункту меню вызовет событие `OnExecute()` командного объекта.

Выполнение команды

Ключевым методом любого командного объекта выступает функция

```
function Execute : Boolean;
```

Метод вызывается автоматически в момент щелчка по элементу управления, связанному с командой. Выполнение команды сопровождается каскадом весьма схожих по функциональной нагрузке событий у имеющих отношение к командному объекту элементов управления (табл. 20.1).

***Таблица 20.1.** Генерирование событий при выполнении команды*

Источник	Событие	Описание
`TActionList` или `TActionManager`	`property OnExecute : TActionEvent;`	Событие генерируется у списка или менеджера команд в зависимости от того, кому из них принадлежит команда
`TApplication`	`property OnActionExecute : TActionEvent;`	Событие вызывается, если команда не была обработана на уровне `TActionList` или `TActionManager`
`TAction`	`property OnExecute : TNotifyEvent;`	Событие произойдет, если команда не была обработана в двух предыдущих событиях

Первые два представленные в табл. 20.1 события описываются процедурой

```
type TActionEvent = procedure(Action: TBasicAction;
                              var Handled: Boolean) of object;
```

В первом параметре `Action` окажется ссылка на команду, чей метод `Execute()` сейчас вызывается. Манипулируя параметром `Handled`, мы сможем разрешить (`false`) или запретить (`true`) дальнейшее выполнение команды.

Установка в актуальное состояние

В перечне событий команды второе по значимости место занимает событие обновления статуса команды

```
property OnUpdate : TNotifyEvent;
```

Благодаря `OnUpdate()` мы сможем установить команду (и, конечно же, связанные с ней элементы управления) в актуальное, отвечающее текущей обстановке состояние. Важно знать, что событие обновления вызывается именно в тот момент, когда связанный с командой элемент готовится к выводу на экран. Благодаря этому команда способна контролировать текущую обстановку с минимальным расходом системных ресурсов, такое решение называется *обновлением по требованию* (update-on-demand approach).

Связь команды с контейнером

Командный объект должен храниться в специальном компоненте-контейнере `TActionList` или `TActionManager`. Для идентификации контейнера, которому принадлежит команда, предназначено свойство

```
property ActionList : TCustomActionList;
```

Индекс команды в списке контейнера находится в свойстве

```
property Index : Integer;
```

Исключительно для удобства программиста во время визуального проектирования команды могут группироваться по категориям. Название категории присваивается свойству

```
property Category : String;
```

Анализируя хранящиеся в этом свойстве текстовые данные, контейнеры `TActionList` и `TActionManager` способны быстро отфильтровать команды определенной категории, тем самым значительно упростив поиск требуемой команды. Кроме того, благодаря свойству `Category` менеджер команд способен создавать группы пунктов меню в командных меню `TActionMainMenu`.

Менеджеры команд

В составе VCL имеются два компонента-контейнера, специализирующихся на обслуживании командных объектов. Это список команд `TActionList` и менеджер команд `TActionManager`. Оба невизуальных компонента берут свое начало от класса `TCustomActionList`, именно он ответственен за создание, хранение и за уничтожение команд (рис. 20.1). В период разработки проекта заполнение контейнеров осуществляется с помощью специального редактора, позволяющего манипулировать командами. Редактор вызывается двойным щелчком левой кнопкой мыши по контейнеру или из его контекстного меню.

Общие черты менеджеров команд

Общие свойства и методы менеджеров команд опубликованы в базовом классе `TCustomActionList`.

Число хранимых в контейнере команд выясняется благодаря свойству

```
property ActionCount : Integer;
```

Доступ к коллекции команд обеспечивает свойство

```
property Actions[Index: Integer]: TContainedAction;
```

Обращение к отдельной команде производится по ее индивидуальному индексу (начиная от 0 и заканчивая `ActionCount-1`).

Контейнеры команд обладают некоторым перечнем прав по отношению к своим подопечным. Например, контейнер способен отдавать приказ на выполнение принадлежащей ему команды.

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

Выполнение метода приведет к генерации события `OnExecute()`. Функция вернет `true` в случае, если в рамках вызванного обработчика события команда была успешно отработана.

Ко всему прочему контейнер способен запретить всем принадлежащим ему командам реагировать на запросы пользователя. Для этого предназначено свойство

```
property State : TActionListState;//по умолчанию asNormal
type TActionListState = (asNormal, asSuspended, asSuspendedEnabled);
```

В момент создания свойству присваивается значение `asNormal`. Это нормальное состояние, разрешающее содержащимся в контейнере командам реагировать на действия пользователя. В состоянии `asSuspended` и `asSuspendedEnabled` все принадлежащие элементу управления команды перестают выполнять запросы пользователя. Разница между последними двумя состояниями в оказываемом влиянии на свойство `Enabled` компонентов `TAction`. Так перевод `TActionList` в состояние `asSuspended` или `asSuspendedEnabled` просто запрещает компонентам `TAction` выполнять запросы пользователя. В дополнение к этому состояние `asSuspendedEnabled` устанавливает все свойства команд `Enabled` в `true`, это требуется при разработке командных панелей (например, `TActionToolBar`).

Еще одно заслуживающее внимание свойство класса `TCustomActionList` подключит к контейнеру набором изображений

```
property Images: TCustomImageList;
```

Соединившись с коллекцией изображений, мы сможем сопоставить с каждой командой отдельный значок. Для этого достаточно указать индекс понравившейся нам картинки в свойстве `ImageIndex` командного объекта.

Потомки `TCustomActionList` унаследовали четыре базовых события, которые вы обнаружите в табл. 20.2.

***Таблица 20.2.** Основные события `TActionList` и `TActionManager`*

Событие	Описание
`property OnExecute : TActionEvent;` `type TActionEvent = procedure (Action: TBasicAction; var Handled: Boolean) of object;`	Событие генерируется у списка (или менеджера) команд при обращении к команде. Параметр `Action` хранит ссылку на команду. Параметр `Handled` позволяет прервать выполнение команды

Таблица 20.2 *(окончание)*

Событие	Описание
property OnChange: TNotifyEvent;	Генерируется при любом изменении в списке команд
property OnStateChange: TNotifyEvent;	Вызывается сразу за событием OnChange(), уведомляет об изменении состояния менеджера
property OnUpdate: TActionEvent;	Событие для централизованной установки всех команд в актуальное состояние. Генерируется во время простоя приложения. В его рамках программист переводит команды в состояние, соответствующее сложившейся обстановке

Список команд *TActionList*

Список команд TActionList — невизуальный элемент управления, который вы найдете на странице **Standard** палитры компонентов. Компонент обычно используют при проектировании приложений малой степени сложности с незначительным числом командных объектов и построенных на основе обычных компонентов меню и стандартных кнопок. Все свойства, методы и обработчики событий списка команд унаследованы от родительского класса TCustomActionList, и поэтому нам уже хорошо знакомы.

По умолчанию список команд Actions пуст. Для заполнения списка командами дважды щелкните по компоненту левой кнопкой мыши. В результате такого действия на экран компьютера выводится окно специализированного редактора (см. рис. 20.2). Для добавления команды TAction следует вызвать пункт **New Action**, а для добавления стандартных команд из заранее подготовленного перечня выбираем пункт **New Standard Action...**. Команды могут группироваться по определяемым программистом категориям. Выбор в левом списке редактора той или иной категории отфильтрует только команды указанной группы. При желании можно отказаться от фильтрации команд, для этого в списке категорий **Categories** надо выбрать строку **All Actions**.

> ***ЗАМЕЧАНИЕ***
>
> Список команд TActionList предназначен для совместной работы с обычными меню (TMainMenu и TPopupMenu) и со стандартными элементами управления, например кнопками TButton.

Менеджер команд *TActionManager*

Как и обычный список команд, менеджер TActionManager построен на основе класса TCustomActionList, благодаря этому он умеет все, чему научен его коллега: создавать, хранить и управлять командными объектами приложения. Но на этом перечень возможностей менеджера не исчерпывается. Изюминка компонента заключена в том, что он способен оказать существенную помощь в проектировании интерфейса приложения, с популярными "выплывающими" меню, скрывающимися неиспользуемыми элементами, сохранением данных о текущих настройках в файл, сменой цветовых схем и многим, многим другим. Результат может превзойти все ожидания, в особенности если менеджер будет работать во взаимном согласии со специально разработанными для него командными панелями: инструментальной панелью TActionToolBar; командными меню TActionMainMenuBar и TPopupActionBar.

Вопрос взаимодействия менеджера команд с командными панелями тщательно продуман. Каждый раз, в момент подключения очередной панели к менеджеру создается новый скрытый экземпляр класса TActionBarItem, сформированный объект станет играть роль посредника между менеджером и настоящей панелью. Все посредники хранятся в коллекции

```
property ActionBars : TActionBars;
```

С помощью этого свойства менеджер команд позволяет обратиться к элементу коллекции TActionBarItem, а через него к подключенной к менеджеру панели команд

```
ActionManager1.ActionBars[0].ActionBar.SetFocus;
```

> ***Замечание***
>
> В период разработки проекта подключение командной панели к коллекции ActionBars осуществляется автоматически в тот момент, когда программист перетащит на эту панель самый первый командный объект.

В недрах компонента скрывается еще одна коллекция командных панелей.

```
property DefaultActionBars : TActionBars;
```

Это эталонные панели, состав команд которых был настроен программистом во время визуального проектирования. При необходимости эталон понадобится для восстановления всех панелей к своему первозданному состоянию.

Командные панели

Несмотря на все свои достоинства, менеджер команд далеко не всемогущ. Его величие заканчивается ровно в том месте, где требуется проявить умение общаться с оператором компьютера. Так как TActionManager не является визуальным компонентом, то после запуска проекта он превращается в невидимку. Оттого, даже узурпировав бразды правления приложением, без помощи третьих лиц менеджер не сумеет предоставить свои услуги пользователю. Поэтому вокруг менеджера вьется целый рой прислуги в лице командных панелей TActionToolBar, TActionMainMenuBar и панели контекстного меню TPopupActionBar.

Для того чтобы спрятанная в недрах менеджера команд команда появилась на панели команд, в недрах проекта Delphi происходят два важных действия.

На первом шаге формируется специальный объект — экземпляр класса TActionClientItem. Этот объект создается автоматически в момент, когда программист мышью перетащит команду из менеджера на поверхность командной панели. Таким образом, экземпляр класса TActionClientItem играет роль представителя команды на панели. Однако объект TActionClientItem — это еще не визуальный элемент управления, а всего лишь невидимый для пользователя экземпляр коллекции. Для того чтобы на командной панели появилась реальная кнопка или элемент меню, по которым пользователь сможет щелкнуть кнопкой мыши, необходима еще одна операция.

На втором шаге на командной панели автоматически создается визуальный элемент управления. В качестве прототипа элемента управления (в зависимости от типа командной панели) обычно выступает класс TStandardButtonControl или TStandardMenuButton. При определении характеристик командного элемента управления панель руководствуется данными, хранимыми в соответствующих элементах коллекций TActionClientItem.

Замечание

В особых случаях программист имеет право подменить создаваемый по умолчанию на командной панели пункт меню или кнопку собственным вариантом элемента управления — потомком класса `TCustomActionControl`.

Класс *TActionClientItem*

Экземпляр класса `TActionClientItem` появляется на свет в момент переноса команды из менеджера `TActionManager` на поверхность командной панели. Обратите внимание на то, что наш новый знакомый не является компонентом, ведь в его цепочке наследования даже нет и намека на класс `TComponent`. Но в данном случае в этом и нет необходимости, ведь `TActionClientItem` играет роль посредника между командой и соответствующей ей кнопкой (или элементом меню) на командной панели.

В функциональные обязанности `TActionClientItem` входит:

- организация связи между командой и полосой команд;
- хранение подчиненных элементов;
- индивидуальная настройка основных интерфейсных характеристик команды;
- ведение статистики обращений пользователя к команде.

Первый пункт из перечня обязанностей выполняется за счет двух свойств. Это ссылка на соответствующую `TActionClientItem` команду

```
property Action: TBasicAction;
```

и ссылка на командную панель, которой он принадлежит

```
property ActionBar: TCustomActionBar;
```

Умение хранить дочерние элементы пригодится при организации различного рода подменю. Это весьма ценное качество унаследовано от предка `TActionClientItem` — класса `TActionClient`.

Список элементов доступен благодаря свойству

```
property Items: TActionClients;
```

Список элементов, входящих в состав контекстного меню (всплывающего по щелчку правой кнопкой мыши):

```
property ContextItems: TActionClients;
```

Для отображения подчиненных элементов используется отдельная командная панель:

```
property ChildActionBar : TCustomActionBar;
```

Ее создание производится автоматически.

Роль `TActionClientItem` в проекте Delphi проиллюстрирована рис. 20.3. Командный элемент `acAlignLeft`, отвечающий за выравнивание текста в редакторе при переносе на командную панель `TactionToolBar`, "превратился" в кнопку, а при размещении на панели главного меню `TActionMainMenuBar` "обернулся" элементом меню. Во время визуального проектирования выбор элемента управления на командной панели сопровождается отображением в Инспекторе объектов свойств соответствующего экземпляра класса `TActionClientItem`.

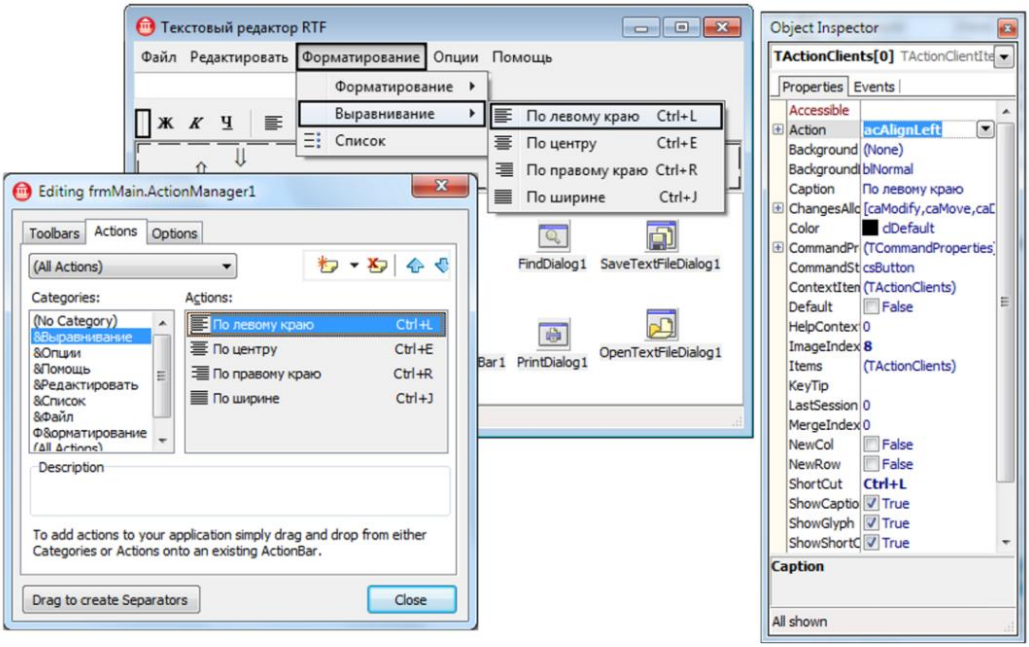


Рис. 20.3. Место класса `TActionClientItem` в проектах Delphi

Замечание

Класс создаваемого на командной панели элемента управления в первую очередь зависит от типа панели. Если речь идет о панелях меню, то обычно создается элемент меню `TStandardMenuItem`, если же мы работаем с инструментальной панелью `TActionToolBar`, то формируется кнопка `TStandardButtonControl`. Именно эти элементы управления и увидит пользователь на панелях во время работы программы.

Основные свойства и методы `TActionClientItem`, определяющие внешний вид элемента управления на командной панели, представлены в табл. 20.3.

***Таблица 20.3.** Внешний вид элемента управления на командной панели*

Свойство	Описание
property `Background: TPicture;`	Фоновый рисунок
property `BackgroundLayout: TBackgroundLayout;`	Особенности вывода на экран изображения, описанного в свойстве `Background`
property `Caption: String;`	Текстовый заголовок
property `Color: TColor;`	Цвет фона
property `ImageIndex: Integer;`	Номер значка
property `ShortCut: TShortCut;`	Быстрые клавиши для обращения к команде
property `ShowCaption: Boolean;`	Разрешение (`true`) или запрет (`false`) показа заголовка
property `ShowGlyph: Boolean;`	Разрешение (`true`) или запрет (`false`) показа значка
property `ShowShortCut: Boolean;`	Разрешение (`true`) или запрет (`false`) показа подсказки быстрых клавиш

Таблица 20.3 *(окончание)*

Свойство	Описание
`function HasGlyph : Boolean;`	Проверка наличия значка
`function HasBackground : Boolean;`	Проверка наличия фонового рисунка

Рассмотрим последнюю (четвертую по счету) обязанность о `TActionClientItem` и обсудим, каким образом этот объект анализирует частоту обращения пользователя к команде.

Дату последнего вызова команды мы получим из свойства

```
property LastSession: Integer;
```

Общее количество вызовов команды хранится в свойстве

```
property UsageCount: Integer;
```

Для программного перевода элемента в разряд неиспользуемых присвойте данному свойству значение –1. Индикатором того, будет система показывать или скрывать элемент (в виду его редкого применения), служит свойство

```
property CheckUnused: Boolean;
```

Это ни в коем случае не аналог свойства `Visible`, потому что пользователь по-прежнему сохраняет возможность обратиться к этому элементу, только теперь его надо "выудить" из списка неиспользуемых.

Опорный класс командных панелей *TCustomActionBar*

В роли системообразующего класса для всех командных панелей выступает класс `TCustomActionBar` (см. рис. 20.1). В первую очередь командная панель представляет собой хранилище для уже знакомых нам экземпляров класса `TActionClientItem` — объектов, обеспечивающих связь с реальными командами. Доступ к коллекции осуществляется посредством свойства

```
property ActionClients: TActionClients;
```

Так как элемент коллекции `TActionClientItem` не способен превратиться в самостоятельный элемент управления, то во время старта приложения в соответствии каждому элементу `TActionClientItem` автоматически создается специальный элемент управления (чаще всего `TStandardMenuItem` или `TStandardMenuButton`). Программист может обратиться к любому из этих компонентов с помощью свойства

```
property ActionControls[const Index: Integer]: TCustomActionControl;
```

Для этого требуется передать индекс интересующего нас элемента управления.

Контакт между командной панелью и менеджером команд устанавливается автоматически в момент размещения на панели команды, принадлежащей менеджеру. После этого ссылка на ассоциированный с панелью менеджер окажется в свойстве

```
property ActionManager: TCustomActionManager;
```

По умолчанию расположенные на панели команды упорядочиваются слева направо. За организацию размещения отвечает свойство

```
property Orientation: TBarOrientation; //по умолчанию boLeftToRight
type TBarOrientation = (boLeftToRight, boRightToLeft, boTopToBottom, boBottomToTop);
```

Кроме того, панель способна упорядочить команды справа налево, сверху вниз и снизу вверх.

Все панели умеют располагать команды в несколько колонок и рядов. Такая способность весьма кстати, когда команды не помещаются на панели в одну линию. В этом случае "лишние" команды автоматически переносятся в новый ряд. Для большей наглядности колонки и ряды целесообразно разделять сепаратором

```
property VertSeparator: Boolean; //вертикальная разделительная полоса
property HorzSeparator: Boolean; //горизонтальная разделительная полоса
```

Для включения разделительной полосы переведите соответствующее свойство в `true`.

Панель главного меню *TActionMainMenuBar*

Панель `TActionMainMenuBar` специализируется на представлении команд в формате главного меню. Основу компонента составляют свойства и методы, унаследованные от `TCustomActionBar`. Вклад всех последующих в иерархии наследования классов значительно скромнее и в первую очередь нацелен на придание меню команд привлекательного внешнего вида. Например, за анимацию меню отвечает свойство

```
property AnimationStyle: TAnimationStyle;
type TAnimationStyle = (asNone, asDefault, asUnFold, asSlide, asFade);
```

При размещении компонента на форме по умолчанию устанавливается значение `asDefault`. В этом случае за способ развертывания пунктов меню на экране будет отвечать система. Значение `asNone` отключит анимацию, это самый щадящий режим не требующих серьезных затрат системных ресурсов. Оставшиеся три значения включат развертывание (`asUnFold`), скольжение (`asSlide`) или выпадение от невидимого к видимому элементу `asFade`.

Продолжительность эффектов анимации задается свойством

```
property AnimateDuration: Integer;//по умолчанию 150 миллисекунд
```

Для включения тени достаточно установить в `true` свойство

```
property Shadows: Boolean;
```

Промежуток времени, спустя который будут отображены и невидимые (по причине редкого обращения к ним пользователя) пункты меню, определяется свойством

```
property ExpandDelay: Integer; //по умолчанию 4000 миллисекунд
```

Для того чтобы выяснить, осуществляется ли в данный момент работа с меню, целесообразно обратиться к свойству

```
property InMenuLoop: Boolean;
```

Свойство возвратит `true` в том случае, когда элементы меню находятся в развернутом состоянии.

Меню приложения часто содержит подменю. Иногда важно выяснить, является ли активный в текущий момент пункт меню подчиненным относительно меню более высокого уровня. Доступ к корневому меню обеспечивает свойство

```
property RootMenu: TCustomActionMenuBar;
```

Собственных методов у компонента совсем немного, мы ограничимся упоминанием только процедуры, применяемой для закрытия развернутого меню

```
procedure CloseMenu;
```

Личных обработчиков событий у TActionMainMenuBar несколько больше (табл. 20.4).

Таблица 20.4. *События* TActionMainMenuBar

Событие	Описание
`property OnEnterMenuLoop: TNotifyEvent;`	Генерируется перед развертыванием меню
`property OnExitMenuLoop: TExitMenuEvent;` `type TExitMenuEvent = procedure(Sender : TCustomActionMenuBar; Cancelled : Boolean) of object;`	Вызывается после закрытия меню
`property OnPopup: TMenuPopupEvent;` `type TMenuPopupEvent = procedure(Sender: TObject; Item: TCustomActionBarControl) of object;`	Вызов контекстного меню
`property OnGetPopupClass: TGetPopupClassEvent;` `type TGetPopupClassEvent = procedure(Sender: TObject; var PopupClass: TCustomPopupClass) of object;`	Событие позволяет подменить создаваемый по умолчанию экземпляр всплывающего меню классом, разработанным программистом

При построении приложений с интерфейсом MDI хорошую помощь окажет свойство

```
property WindowMenu: String;
```

Если в него передать название (содержимое Caption) пункта меню верхнего уровня, то к нему начнут автоматически подключаться дочерние окна приложения.

Инструментальная панель *TActionToolBar*

Инструментальная командная панель отвечает за представление команд из связанного с ней менеджера в виде кнопок. Все ключевые свойства и методы компонентов вобрал от TCustomActionBar.

Контекстное командное меню *TPopupActionBar*

В отличие от остальных командных панелей контекстное командное меню создается не на основе абстрактного класса TCustomActionBar, а из обыкновенного всплывающего меню TPopupMenu (рис. 20.1). Поэтому 99% свойств и методов элемента управления нам уже знакомы по материалам предыдущей главы. Из особенностей компонента нам стоит упомянуть свойство, отвечающее за стиль меню

```
property Style: TActionBarStyle;
```

Кроме того, интерес представляет событие, генерирующееся в момент создания каждого из командных пунктов меню.

```
property OnGetControlClass: TGetControlClassEvent;
```

В заключение разговора о TPopupActionBar осталось вспомнить, что для подключения контекстного меню к какому-либо элементу управления следует воспользоваться свойством PopupMenu этого элемента.

Настройка интерфейса во время выполнения приложения, диалог *TCustomizeDlg*

Несмотря на то, что менеджер команд является невизуальным компонентом, тяжело преувеличить его вклад в формирование пользовательского интерфейса приложения. Тому подтверждение — уникальный механизм динамического изменения состава команд приложения во время его выполнения. Для того чтобы предоставить обычному пользователю право управлять интерфейсом приложения, программисту достаточно воспользоваться услугами компонента `TCustomizeDlg` и написать несколько строк кода.

Компонент `TCustomizeDlg` представляет собой специализированный диалог, позволяющий изменять состав команд на панелях. После подключения к менеджеру с помощью свойства

```
property ActionManager: TCustomActionManager;
```

диалог самостоятельно получает все сведения об имеющихся в распоряжении менеджера команд.

Для обращения к диалогу следует воспользоваться методом

```
procedure Show;
```

В результате на экране появится диалоговое окно, позволяющее перемещать команды между панелями, изменять размер пиктограмм, вставлять новые разделители, реконструировать меню, сохранять или сбрасывать внесенные изменения. Для этого в компоненте `TCustomizeDlg` инкапсулирован уже знакомый нам редактор менеджера команд. Вполне естественно, что версия запускаемого во время выполнения приложения редактора несколько упрощена, и в ней не разрешается создавать новую и удалять существующую команду, также редактор запрещает создавать и удалять командную панель.

С процессом отображения диалога `TCustomizeDlg` связаны два события:

```
property OnShow: TNotifyEvent; //показ диалогового окна
property OnClose: TNotifyEvent;//закрытие диалогового окна
```

Единственное, чего не умеет диалог `TCustomizeDlg`, — это сохранять внесенные пользователем изменения. Поэтому при рестарте приложения все изменения теряются и интерфейс командных панелей возвращается к первозданному состоянию. Для того чтобы научить приложение помнить настройки оператора, потребуется помощь менеджера команд (компонент `TActionManager`). В простейшем случае нам достаточно в свойстве:

```
property FileName: TFileName; //по умолчанию не заполнено = #0
```

определить имя файла, в который менеджер сможет сохранять пользовательские настройки. Если это свойство не пустое, то в указанный файл, во время завершения работы приложения, автоматически будет заноситься информация о текущем состоянии командных панелей. Во время старта приложения менеджер самостоятельно обратится к конфигурационному файлу и извлечет из него ранее сохраненные данные. Если по какой-то причине нас перестали удовлетворять хранящиеся в конфигурационном файле данные, то для очистки этого файла вызывается метод

```
procedure ResetUsageData;
```

После этого пользовательский интерфейс восстанавливается к тому виду, в котором он находился на момент первого старта приложения. Вместо кардинального `ResetUsageData()`

допускается вызывать "терапевтический" метод восстановления первоначального состояния отдельной панели

```
procedure ResetActionBar(Index : Integer);
```

Метод восстановит все настройки панели с порядковым номером `Index` на момент самого первого запуска приложения.

Отдельная группа методов менеджера обеспечивает процесс сохранения конфигурационных данных в определенный ресурс. Такая возможность пригодится при проектировании приложения, в котором каждый пользователь получит право производить индивидуальные для себя настройки интерфейса. Пара методов позволяет сохранять текущие настройки в файл и загружать их из файла

```
procedure SaveToFile(const FileName: string);
procedure LoadFromFile(const FileName: string);
```

Аналогичные действия допустимо осуществлять с областью памяти

```
procedure SaveToStream(Stream: TStream);
procedure LoadFromStream(Stream: TStream);
```

Рассмотрим пример, демонстрирующий порядок сохранения пользовательских настроек в любом приложении на основе менеджера команд. Для повторения примера нам понадобится проект, в состав которого войдут:

- менеджер команд `TActionManager`;
- диалог настроек `TCustomizeDlg`;
- командное меню `TActionMainMenuBar`;
- панель инструментов `TActionToolBar`.

Заполните менеджер команд любыми командами (для ускорения процесса можете воспользоваться командами из стандартного набора) и создайте две обязательные команды: `acCustomize:TAction`, предназначенную для вызова диалога настроек, и `acReset:TAction`, возвращающую интерфейс приложения к первоначальному состоянию. Разместите эти команды на панелях.

Объявим глобальную переменную `ConfigFile:TFileName` (в ней мы сохраним имя конфигурационного файла) и опишем события `OnCreate()` и `OnClose()` главной формы проекта так, как предложено в листинге 20.3.

Листинг 20.3. Подготовка приложения к работе и завершение работы

```
var ConfigFile : TFileName;//глобальная переменная
...
procedure TForm1.FormCreate(Sender: TObject);
var UserName : array[0..24] of char;
    NameSize : Word;
begin
  CustomizeDlg1.ActionManager:=ActionManager1;//подключим диалог
  NameSize:=SizeOf(UserName);//узнаем размер, отводимый для хранения имени
  GetUserName(UserName,NameSize);//узнаем имя пользователя
  {определяем имя пользовательского файла настроек}
  ConfigFile:=ExtractFilePath(Application.ExeName)+UserName+'.dat';
```

```
  if FileAge(ConfigFile)<>-1
  then ActionManager1.LoadFromFile(ConfigFile); //загружаем настройки
end;

procedure Tform1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ActionManager1.SaveToFile(ConfigFile);//сохраняем настройки
end;
```

Во время создания главной формы мы формируем имя конфигурационного файла `ConfigFile`, которое формируется путем сцепления имени пользователя и пути к исполняемому файлу приложения. Если файл с таким именем существует, то мы его загружаем. В момент завершения работы приложения в файл `ConfigFile` сохраняются все изменения.

Теперь нам осталось научить программу вызывать диалог настроек и обнулять все настройки. Для этого мы применяем подготовленные заранее команды `acCustomize` и `acReset` (листинг 20.4).

Листинг 20.4. Команды настройки пользовательского интерфейса

```
procedure TForm1.acCustomizeExecute(Sender: TObject);
begin
  CustomizeDlg1.Show; //вызов диалога настройки команд
end;

procedure TForm1.acResetExecute(Sender: TObject);
var i : integer;
begin
  for i:=0 to ActionManager1.ActionBars.Count-1 do
              ActionManager1.ResetActionBar(i);
  ActionManager1.ResetUsageData; //сброс настроек
end;
```

Мы научили приложение подстраивать свой интерфейс под пожелания пользователя. К сожалению, мы получили англоязычное окно диалога, однако при необходимости совсем несложно переписать исходный код диалога `TCustomizeDialog`, который вы найдете в программном модуле `CustomizeDlg`, и перевести интерфейс диалога на любой язык (рис. 20.5).

Редактор "горячих" клавиш *THotKey*

На последних страницах главы нам предстоит рассмотреть элемент управления, не имеющий прямого отношения к командам. Однако его помощь может оказаться полезной при построении профессиональных приложений как на основе команд, так и на базе изученных в предыдущей главе компонентов-меню.

Для того чтобы сделать наше приложение максимально удобным для эксплуатации, можно предоставить пользователю возможность самостоятельно выбирать комбинацию "горячих" клавиш для того или иного пункта меню. Наиболее просто это сделать с помощью редактора "горячих" клавиш `THotKey`.

На первый взгляд редактор "горячих" клавиш представляет собой обычную строку ввода. Однако первое впечатление обманчиво — в цепочке наследования элемента управления нет и намека на характерный для всех текстовых редакторов класс `TCustomEdit`. Вместо этого в число предков `THotKey` вошли классы `TCustomHotKey` и `TWinControl`.

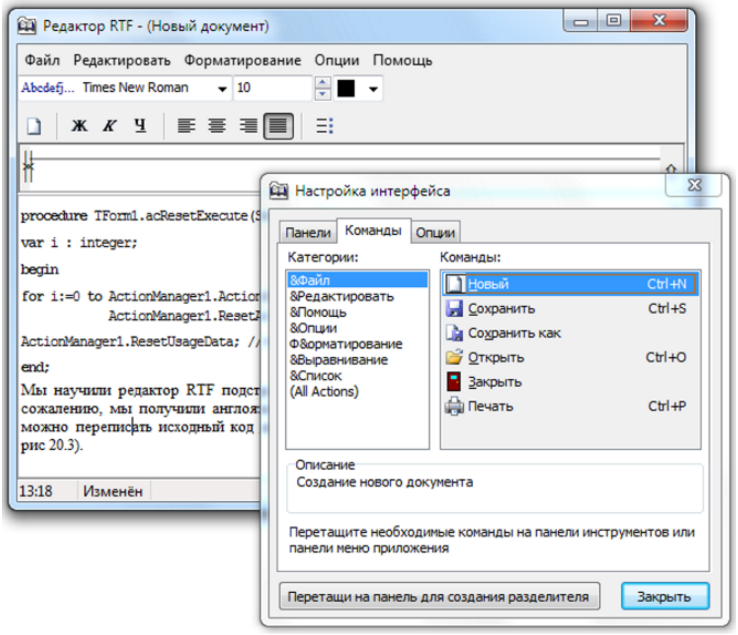


Рис. 20.4. Настройка пользовательского интерфейса с помощью `TCustomizeDialog`

Порядок работы с редактором очень прост. При получении фокуса ввода компонент переходит в режим ожидания нажатия пользователем комбинации клавиш. Дождавшись действия пользователя, редактор "горячих" клавиш запоминает выбранную комбинацию в свойстве

```
property HotKey: TShortCut;//по умолчанию <Alt>+<A>
```

и отображает ее текстовый эквивалент в своем окне.

По умолчанию в свойстве `HotKey` только что размещенного на форме редактора "горячих" клавиш сохранена комбинация клавиш <Alt>+<A>. При желании можно изменить выбранный по умолчанию модификатор <Alt>, направив в свойство

```
property Modifiers: THKModifiers; //по умолчанию hkAlt
```

любое другое значение из перечня

```
type THKModifier = (hkShift, hkCtrl, hkAlt, hkExt);
```

Зачастую при назначении "горячих" клавиш имеет смысл ограничить фантазию пользователя, запретив ему использовать некоторые модификаторы. Константы запрещенных модификаторов следует передать в свойство

```
property InvalidKeys: THKInvalidKeys; //по умолчанию [hcNone, hcShift]
type THKInvalidKey = (hcNone, hcShift, hcCtrl, hcAlt, hcShiftCtrl,
                hcShiftAlt, hcCtrlAlt, hcShiftCtrlAlt);
```

Завершая разговор о компоненте `THotKey`, упомянем о самом главном событии редактора "горячих" клавиш:

```
property OnChange: TNotifyEvent;
```

Событие вызывается в момент изменения свойства `HotKey` элемента управления.

ГЛАВА 21

Списки

Элементы-управления "списки" предназначены для хранения набора текстовых строк и обеспечения выбора одной или нескольких из них пользователем. На рис. 21.1 представлен фрагмент дерева классов VCL с семью компонентами — списками. Обратите внимание на ряд особенностей рассматриваемых в этой главе элементов управления. Во-первых, все компоненты наследуют характеристики оконного класса `TWinControl`. Во-вторых, опорным родительским классом всех списков выступает `TCustomListControl`. В-третьих, элементы управления "списки" разделяются на три группы: простые списки, комбинированные списки и список просмотра, представленный единственным компонентом `TListView`.

> ***Замечание***
>
> Ключевым свойством рассматриваемых в этой главе списков является уже знакомое нам свойство `Items`, основанное на классе `TStrings` (за исключением `TListView`) и подробно рассмотренное в *главе 10*.

Простые и комбинированные списки предназначены для решения одних и тех же задач — предоставление пользователю возможности выбора своих элементов. Схожесть задач объясняет схожесть построения компонентов. Самое существенное различие между элементами управления в том, что обычный список представляет собой окно, в котором одновременно отображаются все его элементы, если из-за ограниченного размера окна текстовые элементы не могут быть выведены на экран полностью, то окно списка снабжается полосой прокрутки. Комбинированный список немного сложнее, фактически он представляет собой объединение двух элементов управления: собственно списка и строки ввода. Строка ввода по своим возможностям практически повторяет класс `TEdit`, но с существенной доработкой — щелчок по кнопке, расположенной в правой части строки, разворачивает список, в котором пользователь осуществляет свой выбор. После выбора список сворачивается. Благодаря этому комбинированный список занимает значительно меньше места на форме, чем его "коллега" — обычный список (рис. 21.2).

> ***Замечание***
>
> Наиболее существенное функциональное отличие между списками и комбинированными списками в том, что компоненты, входящие в группу списков, позволяют пользователю одновременно выбирать несколько своих строковых элементов. Комбинированные списки такой сервис не предоставляют.

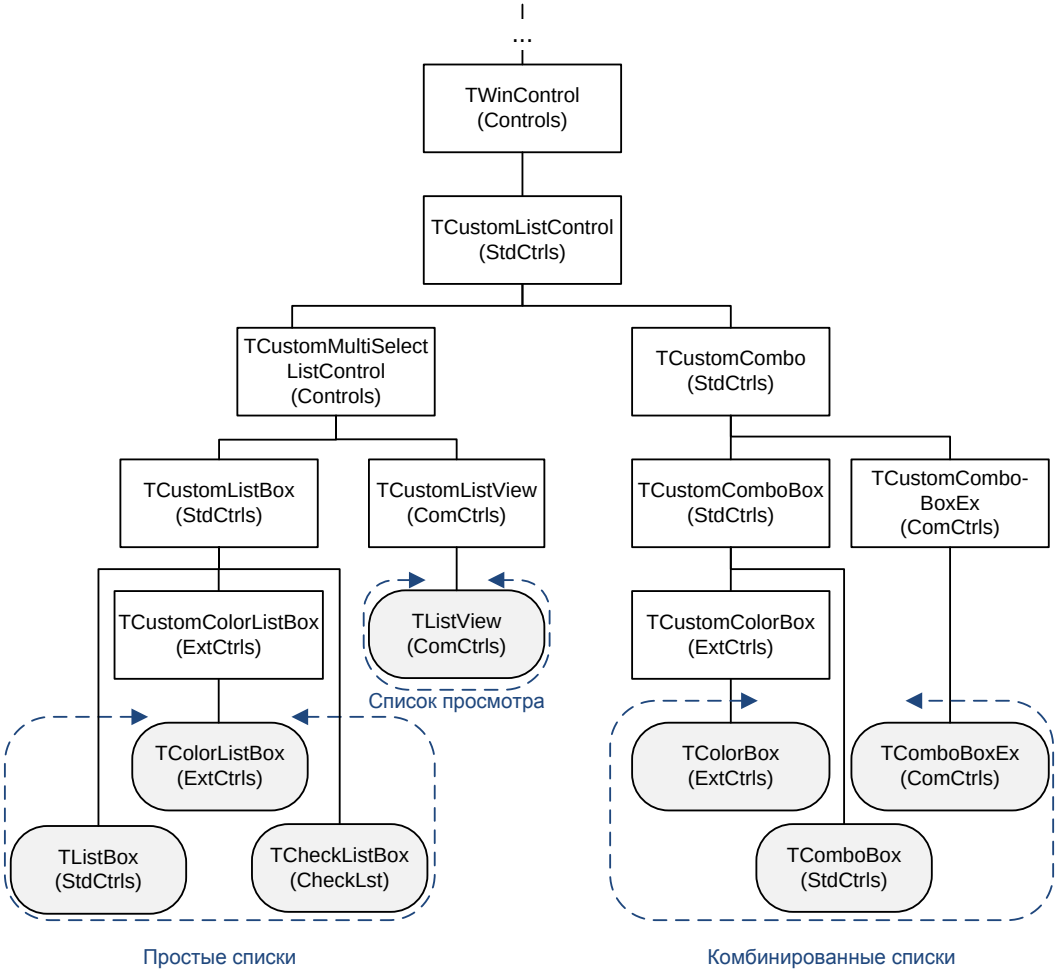


Рис. 21.1. Место компонентов-списков в иерархии классов VCL

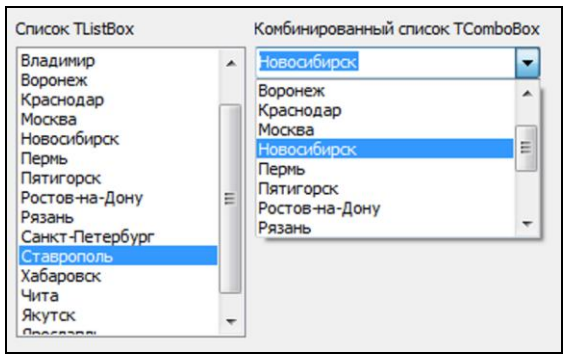


Рис. 21.2. Внешний вид списка `TListBox` *(слева)* и комбинированного списка `TComboBox` *(справа)*

Опорный класс списков *TCustomListControl*

Для всех изучаемых в этой главе элементов управления точкой отсчета выступает класс `TCustomListControl`. В нем сосредоточены объявления методов, задающих общие правила поведения всей плеяды компонентов-списков.

Для добавления нового элемента в список проще всего воспользоваться методом

```
procedure AddItem(Item: String; AObject: TObject);
```

Заметьте, что, кроме текстового описания элемента `Item`, процедура позволяет добавить к списку ссылку на любой объект `AObject`. Если объект отсутствует, то вместо ссылки передается неопределенный указатель `nil` (листинг 21.1).

Листинг 21.1. Заполнение списка данными

```
with ListBox1 do
begin
  Clear;
  AddItem('Владивосток',nil);
  AddItem('Казань',nil);
  //...
  Sorted:=True;
end;
```

О числе хранящихся в списке элементов проинформирует метод

```
function GetCount: Integer;
```

Индекс текущего элемента списка (последнего элемента, выбранного пользователем) доступен благодаря свойству

```
property ItemIndex: Integer;
```

Если ни один из элементов списка не выбран, то свойство возвратит значение –1. Это свойство позволяет не только считывать номер выбранного элемента, но и назначать его из кода программы.

Для удаления выделенного элемента следует вызывать один из методов:

```
procedure DeleteSelected;
procedure ClearSelection;
```

Последний метод предназначен для работы в списках, позволяющих пользователю одновременно выбирать несколько элементов.

Полная очистка списка осуществляется благодаря процедуре

```
procedure Clear;
```

Поведение метода

```
procedure SelectAll;
```

зависит от того, к какому компоненту он был применен. Если элемент управления принадлежит группе комбинированных списков, то процедура просто выделит весь текст в строке. Если же речь идет об обычном списке, то будут выделены все его элементы (при условии, что его свойство `Multiselect` установлено в `true`).

Выделенные элементы могут быть скопированы и переданы в другой список. Для этого предназначена процедура

```
procedure CopySelection(Destination: TCustomListControl);
```

Второй вариант развития событий — выделенные элементы переносятся в другой список.

```
procedure MoveSelection(Destination: TCustomListControl);
```

Общие черты списков, список *TListBox*

Основное функциональное отличие простого списка от его комбинированного собрата в том, что обычный список допускает одновременный выбор нескольких строк, а комбинированный — нет. Такому поведению списки обязаны классам `TCustomMultiSelectListControl` и `TCustomListBox`, несущим прямую ответственность за поддержку множественного выбора.

Для того чтобы компонент-список разрешил пользователю производить одновременный выбор нескольких строк, следует перевести в режим `true` свойство

```
property MultiSelect: Boolean; //по умолчанию false
```

О числе выделенных элементов проинформирует свойство

```
property SelCount: Integer;
```

Чтобы для одновременного выбора нескольких элементов списка пользователь мог прибегнуть к услугам клавиш <Ctrl> и <Shift>, проконтролируйте установку `true` свойства

```
property ExtendedSelect: Boolean; //по умолчанию true
```

Для того чтобы выяснить, выбран элемент списка или нет, следует обратиться к свойству

```
property Selected[Index: Integer]: Boolean;
```

Свойство возвращает значение `true`, если элемент с порядковым номером `Index` выбран пользователем. Предложенный в листинге 21.2 пример просматривает весь перечень элементов списка и "склеивает" из названий выделенных пользователем элементов строку `s`.

Листинг 21.2. Сбор названий выбранных элементов

```
var s:string;
    i:integer;
begin
  for i:=0 to ListBox1.GetCount-1 do
     if ListBox1.Selected[i]=true then
        s:=s+'-'+ListBox1.Items.Strings[i];
end;
```

На уровне `TCustomListBox` объявлены очень важные методы и свойства. В первую очередь это ключевое для всех списков свойство

```
property Items: TStrings;
```

предоставляющее доступ к элементам списка в формате класса `TStrings`.

Списки способны самостоятельно упорядочивать свои текстовые элементы по возрастанию. Для этого следует перевести в `true` свойство

```
property Sorted:boolean; //по умолчанию false
```

Свойство

```
property Count: Integer;
```

расскажет программисту о числе элементов списка.

> ***Внимание!***
>
> Если выбран виртуальный (`lbVirtual`) стиль списка (см. табл. 21.1), то свойство `Count` позволит программисту определить количество виртуальных элементов списка.

Список может содержать сотни и тысячи элементов, которые даже при самом большом желании пользователя не могут быть одновременно отображены в окне списка. Для того чтобы выяснить индекс самого верхнего видимого элемента списка, следует обратиться к свойству

```
property TopIndex: Integer;
```

Свойство доступно и в режиме записи, оно позволит изменять перечень видимых элементов.

При необходимости элемент управления может проинформировать о порядковом номере элемента по его экранным координатам. Для этого следует воспользоваться методом

```
function ItemAtPos(Pos: TPoint; Existing: Boolean): Integer;
```

При отсутствии элемента списка в точке `Pos` характер возвращаемого значения определяется параметром `Existing`: `true` — результат функции равен –1; `false` — возвращается индекс последнего элемента. Этот метод обычно работает в содружестве с событиями, реагирующими на манипуляции мышью (листинг 21.3).

Листинг 21.3. Обращение к элементу списка по его координатам (`X`, `Y`)

```
procedure TForm1.ListBox1MouseMove(Sender: TObject; Shift: TShiftState;
                                   X, Y: Integer);
var s : string;
    Index : integer;
begin
  Index:=ListBox1.ItemAtPos(Point(X,Y), true);
  if Index<>-1 then s:=ListBox1.Items.Strings[Index];
end;
```

Существенное влияние на внешний вид и особенности поведения списков оказывает стиль элемента управления:

```
property Style: TListBoxStyle;//по умолчанию lbStandard
```

Варианты стилей предложены в табл. 21.1.

Таблица 21.1. *Стили списков* `TListBoxStyle`

Стиль	Описание	Высота элементов
`lbStandard`	Стандартный стиль	Постоянная
`lbOwnerDrawFixed`	Разрешен программный доступ для прорисовки элементов списка	Постоянная

Таблица 21.1 *(окончание)*

Стиль	Описание	Высота элементов
lbOwnerDrawVariable	Программный доступ для прорисовки элементов списка, допустима индивидуальная настройка высоты каждого элемента	Индивидуальная
lbVirtual	Стиль виртуального списка, элементы списка не заносятся в свойство Items, а создаются динамически во время обработки события OnData(). Число элементов назначается свойством Count. Становятся доступными обработчики события OnDataObject() и OnDataFind(), предназначенные для обслуживания связанных с элементами объектов	Постоянная
lbVirtualOwnerDraw	Аналогично lbVirtual, но с возможностью программной перерисовки элементов списка	Постоянная

ЗАМЕЧАНИЕ

К нюансам применения того или иного стиля мы еще вернемся при обсуждении специфичных для списков обработчиков событий (см. табл. 21.2).

Для получения координат прямоугольной области, содержащей элемент списка с индексом Index, используйте метод

```
function ItemRect(Index: Integer): TRect;
```

Полученные координаты могут оказаться полезными при программировании операций прорисовки.

Отказавшись от нулевого значения в свойстве

```
property Columns: Integer;//по умолчанию 0
```

можно разместить элементы списка в несколько колонок. Если список разбит на несколько колонок, то стоит вспомнить о существовании свойства, знающего (а при необходимости и определяющего) логическую ширину окна просмотра

```
property ScrollWidth: Integer; //по умолчанию 0
```

Если значение свойства превышает клиентскую ширину окна, то это приводит к автоматическому появлению горизонтальной полосы прокрутки.

Если текстовые элементы списка могут содержать символ табуляции, то стоит настроить размер отступа

```
property TabWidth: Integer;
```

В качестве единицы измерения принят пиксел.

За автоматическую подстройку высоты списка отвечает свойство:

```
property IntegralHeight: Boolean;
```

По умолчанию оно установлено в false и допускает "обрезку" последнего элемента списка.

Внешний вид границ элемента управления определяется свойством

```
property BorderStyle: TBorderStyle;
```

Замедление перебора элементов списка

По умолчанию при переходе от одного элемента списка к другому с помощью клавиш список моментально изменяет индекс выбранного элемента `ItemIndex`. Иногда это нецелесообразно, например, когда текстовый элемент содержит длинное название и код нашей программы не успевает его обработать. В таком случае можно несколько замедлить процесс смены индекса выбранного элемента. Для этого следует перевести в `false` свойство

```
property AutoComplete: Boolean; //по умолчанию true
```

Временной интервал отсрочки смены индекса `ItemIndex` назначается в свойстве

```
property AutoCompleteDelay: Cardinal; //по умолчанию 500 миллисекунд
```

Особенности обработки событий

Переопределив стиль списка в состояние `lbOwnerDrawFixed` или `lbOwnerDrawVariable` и слегка "поколдовав" над исходным кодом, вполне реально добиться весьма неординарных результатов при выводе списка на экран. Для этого применяется обработчик события

```
property OnDrawItem: TDrawItemEvent;
type TDrawItemEvent = procedure(Control: TWinControl; Index: Integer;
                        Rect: TRect; State: TOwnerDrawState) of object;
```

Это событие вызывается каждый раз, когда списку необходимо перерисовать свой элемент. Здесь: `Control` — ссылка на элемент управления, вызвавший событие; `Index` — индекс выводимого на экран элемента; `Rect` — координаты прямоугольной области, ограничивающей элемент; `State` — состояние элемента списка, характеризуемое множеством значений `TOwnerDrawState`.

```
type TOwnerDrawState = set of (odSelected, {выбран}
                               odGrayed,   {серый}
                               odDisabled, {заблокирован}
                               odChecked,  {отмечен флажком}
                               odFocused );{в фокусе ввода}
```

Создайте новый проект и разместите на его форме следующий перечень элементов управления.

- Строка ввода `TEdit` или `TLabeledEdit`. Пользователь укажет в строке путь к интересующей его папке с метафайлами (wmf-файлами). Например, это может быть каталог C:\Program Files\Microsoft Office\CLIPART\PUB60COR\, в котором хранятся метафайлы из состава программного пакета Microsoft Office.
- Кнопка `TButton`. Щелчок по кнопке даст команду на сбор данных о доступных файлах.
- Список `TListBox1`. С помощью свойства `Anchors` (или `Align`) сделайте так, чтобы список занял все свободное клиентское пространство формы. Переведите стиль списка в состояние `lbOwnerDrawFixed` и увеличьте высоту элемента `ItemHeight` до 50 пикселов.

Наша работа начнется с описания щелчка по единственной кнопке (листинг 21.4).

Листинг 21.4. Сбор метафайлов в список

```
procedure TForm1.Button1Click(Sender: TObject);
var s:string;
begin
  ListBox1.Clear; //очищаем список
```

```
  //создаем маску для сбора wmf-рисунков
  s:=IncludeTrailingBackslash(TRIM(Edit1.text));
  if Edit1.text<>s then Edit1.text:=s;
  s:=s+'*.wmf'#0;
  //заполняем список именами доступных файлов
  ListBox1.Perform(LB_DIR, DDL_READWRITE, INTEGER(@s[1]));
end;
```

В результате выполнения кода из листинга 21.4 в элементах списка окажутся названия файлов с расширением emf.

Вторым этапом программирования станет описание события `OnDrawItem()`, которое позволит нам сформулировать наши требования к прорисовке каждого элемента списка (листинг 21.5).

Листинг 21.5. Событие перерисовки элементов списка

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var OffSet   : Integer;
    WMFRect  : TRect;
    WMF      : TMetafile;
begin
  with (Control as TCustomListBox).Canvas do
  begin
    FillRect(Rect); //заливка прямоугольника текущей кистью
    WMF:=TMetafile.Create; //создание экземпляра метафайла
    WMF.LoadFromFile(Edit1.text+ListBox1.Items[Index]); //загрузка
    WMFRect:=Bounds(Rect.Left+1,Rect.Top+1,
                    Rect.Bottom-Rect.Top-2,Rect.Bottom-Rect.Top-2);
    StretchDraw(WMFRect, WMF);     //прорисовка изображения
    Offset:=Rect.Bottom-Rect.top+6;//расчет отступа текста
    TextOut(Rect.Left+Offset,    {вывод текста с именем файла}
      Rect.Top+(Rect.Bottom - Rect.top) div 2,ListBox1.Items[index]);
    WMF.Free; //освобождение ресурсов метафайла
  end;
end;
```

Основная часть кода уже написана, можно смело запустить проект и проверить его работоспособность (рис. 21.3).

При использовании стиля `lbOwnerDrawVariable` высота каждой строки задается программистом в обработчике события

```
property OnMeasureItem: TMeasureItemEvent;
type TMeasureItemEvent = procedure(Control: TWinControl;
                         Index: Integer; var Height: Integer) of object;
```

Обработчик манипулирует тремя параметрами: `Control` — ссылка на список, `Index` — порядковый номер элемента и `Height` — новая высота элемента.

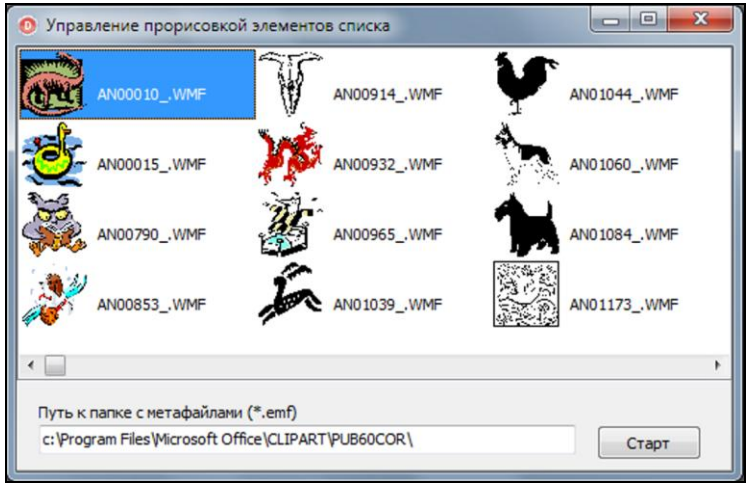


Рис. 21.3. Управление процессом прорисовки элементов списка `TListBox`

Если список работает в виртуальном режиме (`Style=lbVirtual` или `lbVirtualOwnerDraw`), то его элементы формируются не благодаря свойству `Items`, а виртуально — в момент обработки события `OnData()` (табл. 21.2).

Таблица 21.2. *События обслуживания виртуальных элементов списка*

Событие	Описание
`property OnData: TLBGetDataEvent;` `type TLBGetDataEvent = procedure(Control: TWinControl; Index: Integer; var Data: string) of object;`	Событие создания виртуальных элементов списка. Здесь: `Control` — ссылка на список; `Index` — порядковый номер виртуального элемента и `Data` — текст элемента списка
`property OnDataObject: TLBGetDataObjectEvent;` `TLBGetDataObjectEvent = procedure(Control: TWinControl; Index: Integer; var DataObject: TObject) of object;`	Событие понадобится в том случае, когда с виртуальным элементом с индексом `Index` необходимо связать внешний объект. Ссылка на объект передается в параметр `DataObject`.
`property OnDataFind: TLBFindDataEvent;` `TLBFindDataEvent = function(Control: TWinControl; FindString: string): Integer;`	Осуществляет поиск индекса виртуального элемента по его имени `FindString`

Замечание

Виртуальный стиль работы поддерживают классы `TListBox`, `TCheckListBox` и `TListView`. Список `TColorListBox` не способен создавать виртуальные элементы.

Пример создания виртуального списка с использованием элемента управления `TListBox` предложен в листинге 21.6.

Внимание!

После перевода списка в виртуальный режим работы необходимо определить число виртуальных элементов. Для этого предназначено свойство `Count`.

Листинг 21.6. Создание виртуальных элементов списка в событии OnData()

```
procedure TForm1.FormShow(Sender: TObject);
begin
  ListBox1.Style:=lbVirtual; //виртуальный стиль
  ListBox1.Count:=10;        //в списке 10 элементов
end;

procedure TForm1.ListBox1Data(Control: TWinControl; Index: Integer;
  var Data: string);
begin
  Data:='Виртуальный элемент '+IntToStr(Index);
end;
```

Список с флажками выбора *TCheckListBox*

Элемент управления TCheckListBox объединил достоинства списка и элемента-флажка. Каждая строка списка снабжена индивидуальным флажком со свойствами, напоминающими отдельный элемент управления TCheckBox. Флажок может быть установлен в одно из трех состояний

```
property State[Index: Integer]: TCheckBoxState;
type TCheckBoxState = (cbUnchecked, //флажок снят
                       cbChecked,   //элемент отмечен флажком
                       cbGrayed);   //неопределенное состояние
```

Параметр Index не что иное, как индекс элемента в списке.

```
property AllowGrayed: Boolean;
```

Если в программе нет необходимости устанавливать флажок в промежуточное состояние, то вместо свойства State достаточно применять свойство

```
property Checked[Index: Integer]: Boolean;
```

Внедрение флажков упростило процесс одновременного выбора нескольких элементов списка и позволяет программисту не обращаться к свойствам MultiSelect и ExtendedSelect.

Программист получил возможность "отключить" любой элемент списка, установив в false свойство

```
property ItemEnabled[Index: Integer]: Boolean;
```

Для флажка это аналог установки его состояния State в cbGrayed.

У TCheckListBox имеется еще один альтернативный способ выделения строки:

```
property Header[Index: Integer]: Boolean;
```

Установив это свойство элемента в true, вы превратите его в заголовок.

Допустим, что в вашем списке хранится некоторый упорядоченный перечень текстовых строк, тогда для динамического создания заголовков можно воспользоваться строками кода, приведенными в листинге 21.7.

На рис. 21.4 представлен внешний вид списка с динамически созданными заголовками.

Листинг 21.7. Динамическое создание заголовков в списке с флажками

```
var i:integer;
begin
  with CheckListBox1 do
  begin
    Items.BeginUpdate; //начало обновления списка
    Sorted:=false;     //временное отключение сортировки
    for i:=Byte('A') to Byte('Я') do //заголовки по 1 букве алфавита
      begin
        AddItem(ANSICHAR(i),nil);//вставка в конец списка
        Header[Count-1]:=True;   //признак заголовка
      end;
    Sorted:=True;    //вновь сортируем список
    Items.EndUpdate; //конец обновления
  end;
end;
```

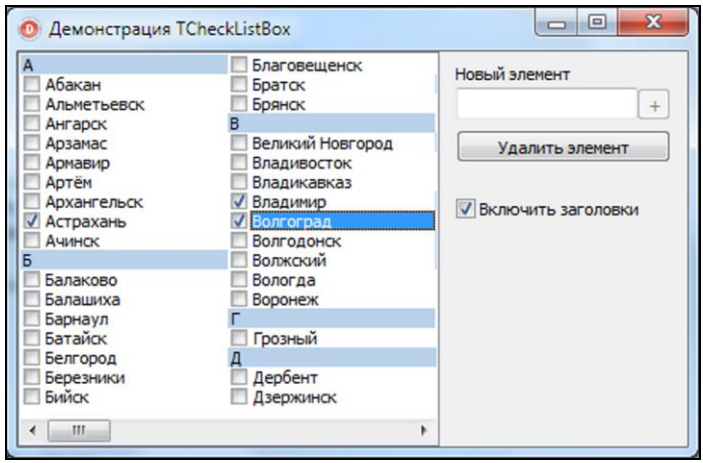


Рис. 21.4. Список `TCheckListBox` с элементами-заголовками

Цвет шрифта заголовка устанавливается свойством:

```
property HeaderColor: TColor;
```

Цвет заливки строки заголовка:

```
property HeaderBackgroundColor: TColor;
```

Перечень обработчиков событий повторяет события `TListBox` и дополнен обработчиком события щелчка по флажку:

```
property OnClickCheck: TNotifyEvent;
```

Список выбора цвета *TColorListBox*

Как и следует из названия, список выбора цвета `TColorListBox` специализируется на предоставлении пользователю возможности выбора одного из своих разноцветных элементов.

С точки зрения программной реализации, список выбора цвета TColorListBox также построен на основе набора строк TStrings, каждый элемент которого хранит текстовое описание имени и значение цветовой константы. Значения констант используются для определения цвета кисти при прорисовке элементов списка скрытым от программиста методом DrawItem().

Доступ к цветовым элементам компонента производится при посредничестве свойства

```
property Colors[Index: Integer]: TColor;
```

Каждому цвету сопоставлено его название (имя константы или англоязычное название цвета)

```
property ColorNames[Index: Integer]: string;
```

Заполнение списка данными осуществляется автоматически, особенности поведения и состав включаемых в список цветов в первую очередь определяется стилем

```
property Style: TColorBoxStyle;
```

Доступные стили описаны в табл. 21.3, по умолчанию активированы стили cbStandardColors, cbExtendedColors и cbSystemColors.

Таблица 21.3. *Описание доступных стилей* `TColorBoxStyles`

Стиль	Описание
cbStandardColors	В состав элементов входят 18 стандартных чистых цветов: clBlack, clMaroon, clGreen, clOlive, clNavy, clPurple, clTeal, clGray, clSilver, clRed, clLime, clYellow, clBlue, clFuchsia, clAqua, clLtGray, clDkGray и clWhite
cbExtendedColors	Дополнительные цвета: clMoneyGreen, clSkyBlue, clCream и clMedGray
cbSystemColors	Системные цвета, соответствующие установкам в Панели управления Windows
cbIncludeNone	Допускается отключение цвета clNone, опция работает только совместно с cbSystemColors
cbIncludeDefault	Список цветов включает цвет по умолчанию
cbCustomColor	Первый элемент списка отображает пользовательский цвет
cbPrettyNames	Для обозначения цвета в элементах списка вместо именованных констант выводятся обычные названия цветов
cbCustomColors	Разрешает выбирать пользовательские цвета

Проявив совсем немного выдумки, можно научить список выбора цвета показывать только те цвета, которые нужны пользователю. Листинг 21.8 демонстрирует порядок назначения трех пользовательских цветов. Для этого в список передается наименование элемента и соответствующее ему значение цвета.

Листинг 21.8. Создание пользовательских цветов в TColorListBox

```
with ColorListBox1 do
  begin
   Style:=[cbCustomColors];
   AddItem('Красный',TObject($0000FF));
   AddItem('Зеленый',TObject($00FF00));
   AddItem('Синий',  TObject($FF0000));
   Selected:=$0000FF;
  end;
```

Значение выбранного пользователем цвета возвращает свойство

```
property Selected: TColor; //по умолчанию clBlack
```

Это же свойство может применяться для программной установки текущего цвета.

Кроме того, пара свойств отвечает за цвет по умолчанию

```
property DefaultColorColor: TColor; //по умолчанию clBlack
```

и значение "отсутствующего" цвета

```
property NoneColorColor: TColor;    //по умолчанию clBlack
```

Разговор об особенностях элемента управления закончим упоминанием события

```
property OnGetColors: TLBGetColorsEvent;
TLBGetColorsEvent = procedure(Sender: TCustomColorListBox;
                              Items: TStrings) of object;
```

Данное событие генерируется в момент инициализации компонента и может применяться для заполнения списка пользовательскими цветами. Цветовые значения передаются в параметр `Items` элемента управления.

Комбинированные списки, *TComboBox*

Ключевое отличие комбинированного списка от его обычного "коллеги" отражено в названии класса. Слово *Combo* говорит о том, что все потомки класса `TCustomCombo` представляют собой комбинацию списка и строки ввода. Строка ввода позволяет нам ввести новый текстовый элемент либо выбрать уже существующий из имеющегося списка. Список выпадает из строки ввода по щелчку на кнопке в правой части строки. Еще одна особенность комбинированных списков в том, что у них не предусмотрена возможность выбора нескольких элементов одновременно. Впрочем, в этом нет и необходимости, ведь результат выбора должен отобразиться в единственной строке ввода.

Разработчики опорного для всех комбинированных списков класса `TCustomCombo` стараются не афишировать тот факт, что физически элемент управления состоит более чем из одного окна. Поэтому дескрипторы окон списка и строки ввода спрятаны среди полей класса, а доступ к ним возможен только с помощью свойств

```
property ListHandle: HWnd; //дескриптор окна списка
property EditHandle: HWnd; //дескриптор окна строки ввода
```

которые, в свою очередь, скрываются от нас в защищенной секции `protected`. Знания об этом могут пригодиться при проектировании собственных версий комбинированных списков.

> ***ЗАМЕЧАНИЕ***
>
> Создатели Delphi при разработке обычных и комбинированных списков приложили максимум усилий, чтобы при работе с ними можно было использовать единый стиль программирования. Благодаря этому, названия и назначение большинства свойств и методов этих классов идентичны.

В основном задача опорного класса `TCustomCombo` сводится к адаптации уже знакомых нам по обычным спискам свойств и методов к решению задач, характерных для комбинированных списков и в первую очередь для наиболее популярного у программистов элемента управления `TComboBox` (табл. 21.4). Например, методы занимающиеся выделением отдель-

ных элементов в обычных списках, в комбинированных списках переориентированы на выделение символов текста в строке ввода. Это объясняется тем, что комбинированные списки не позволяют одновременно выделять несколько строк.

Таблица 21.4. *Общие свойства и методы комбинированных списков*

Свойства и методы	Описание
Управление данными в списке	
`procedure AddItem(Item: string; AObject: TObject);`	Добавление нового элемента в список
`procedure Clear;`	Удаление всех элементов списка
`procedure CopySelection(Destination: TCustomListControl);`	Копирует выбранный элемент списка в другой список
`procedure DeleteSelected;`	Удаляет выбранный пользователем элемент из списка
`function GetCount: Integer;`	Возвращает число элементов списка
`property Items: TStrings;`	Предоставляет доступ к элементам списка
Управление строкой ввода	
`property Text:string;`	Доступ к тексту в строке ввода
`property CharCase: TEditCharCase;` `type TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);`	Ограничение регистра вводимых символов
`procedure SelectAll;`	Выделяет весь текст в строке ввода списка
`procedure ClearSelection;`	Снимает выделение с текста
`property SelLength: Integer;`	Устанавливает длину выделенного текста
`property SelStart: Integer;`	Определяет начало выделения
`property SelText: string;`	Доступ к выделенному в строке тексту
`property AutoComplete: Boolean;`	Автоматическое завершение строки. При вводе пользователем текста в строку ввода осуществляется поиск наиболее подходящего значения из списка
Развертывание и свертывание списка	
`property AutoCloseUp: Boolean;`	Автоматическое свертывание списка
`property AutoDropDown: Boolean;`	Автоматическое раскрытие списка
`property DroppedDown: Boolean;`	В состоянии `true` указывает, что список развернут, `false` — свернут. Может применяться для программного управления выпадающим списком
`property DropDownCount: Integer;`	Число строк (по умолчанию 8), отображаемых в выпадающем списке

Во многом поведение комбинированного списка `TComboBox` определяется его стилем

```
property Style: TComboBoxStyle;//по умолчанию csDropDown
```

Особенности того или иного стиля представлены в табл. 21.5.

Таблица 21.5. *Стили комбинированного списка* `TComboBoxStyle`

Значение	Описание
`csDropDown`	Допускается ввод текста в строку ввода
`csSimple`	Одновременно доступны строка ввода и список. Чтобы увидеть список, достаточно увеличить значение свойства `Height` компонента
`csDropDownList`	Комбинированный список не допускает ввода текста в строку
`csOwnerDrawFixed`	Комбинированный список с возможностью программного управления прорисовкой его элементов. Строка ввода не допускает редактирование текста. Высота элементов определяется свойством `ItemHeight`
`csOwnerDrawVariable`	Стиль аналогичен `csOwnerDrawFixed`, но допускает индивидуальную настройку высоты каждого элемента списка

В ответ на выбор пользователем строки в выпадающем списке генерируется событие

```
property OnSelect: TNotifyEvent;
```

Изменение текста порождает событие

```
property OnChange: TNotifyEvent;
```

Процесс развертывания и сворачивания списка сопровождается событиями

```
property OnDropDown: TNotifyEvent; //список развертывается
property OnCloseUp: TNotifyEvent;  //список свертывается
```

Если комбинированный список допускает программное управление прорисовкой (стиль установлен в состояние `csOwnerDrawFixed` или `csOwnerDrawVariable`), то управление графическим выводом осуществляется в рамках события

```
property OnDrawItem: TDrawItemEvent;
```

Индивидуальная настройка высоты строк (стиль `csOwnerDrawVariable`) обеспечивает событие

```
property OnMeasureItem: TMeasureItemEvent;
```

Улучшенный комбинированный список *TComboBoxEx*

В дополнение к обычному комбинированному списку в состав библиотеки визуальных компонентов включен комбинированный список с расширенными возможностями `TComboBoxEx`. Наиболее существенное отличие элемента управления `TComboBoxEx` от своих "коллег" — наличие у него улучшенного списка элементов.

```
property ItemsEx: TComboExItems;
```

Теперь в роли элемента списка выступает не обычная текстовая строка, а объект — экземпляр класса `TComboExItem`. Элемент расширенного списка обладает некоторым перечнем свойств и методов, ключевые из них представлены в табл. 21.6.

> ***Замечание***
>
> В классе `TComboBoxEx` по-прежнему доступно привычное для всех списков свойство `Items`, позволяющее связывать со списком обычный набор текстовых строк.

Управление списком возлагается на коллекцию `TComboExItems`, ее основные характеристики представлены в табл. 21.7.

Таблица 21.6. *Основные свойства элемента списка — класса* `TComboExItem`

Свойство	Описание
`property Caption: string;`	Заголовок элемента
`property Data: TCustomData;`	Дополнительные данные, связанные с элементом
`property ImageIndex: TImageIndex;`	Индекс значка элемента
`property SelectedImageIndex: TImageIndex;`	Индекс значка для выбранного элемента
`property OverlayImageIndex: TImageIndex;`	Необязательный индекс черно-белой пиктограммы-маски для создания эффекта прозрачности
`property Indent: Integer;`	Величина отступа от левого края списка (единица измерения — 10 пикселов)

Таблица 21.7. *Управление элементами списка — класс* `TComboExItems`

Свойство/метод/событие	Описание
`function Add: TComboExItem;`	Добавление нового элемента
`function AddItem(const Caption: string; const ImageIndex, SelectedImageIndex, OverlayImageIndex, Indent: Integer; Data: TCustomData): TComboExItem;`	Добавление нового элемента с одновременным назначением заголовка, номеров значков и дополнительных данных
`function Insert(Index: Integer): TComboExItem;`	Вставка нового элемента в указанное место
`procedure Delete(Index: Integer);`	Удаление элемента по его индексу
`procedure Clear;`	Полная очистка списка
`property ComboItems[const Index: Integer]: TComboExItem;`	Доступ к отдельному элементу списка по его индексу
`property SortType: TListItemsSortType;`	Определение порядка сортировки элементов в списке
`procedure Sort;`	Инициация процесса сортировки
`property OnCompare: TListCompareEvent;`	Событие, позволяющее определять пользовательские особенности упорядочивания

С каждым элементом улучшенного списка можно сопоставить несколько пояснительных значков. Они заносятся в контейнер рисунков `TImageList` и подключаются к комбинированному списку с помощью свойства

```
property Images: TCustomImageList;
```

На особенности функционирования комбинированного списка с расширенными возможностями оказывают влияние стили компонента.

```
property Style: TComboBoxExStyle; //по умолчанию csExDropDown;
type TComboBoxExStyle = (
   csExDropDown, //выпадающий список с редактируемой строкой ввода
   csExSimple,   //строка ввода с заранее развернутым списком
   csExDropDownList); //список без строки ввода
```

```
property StyleEx: TComboBoxExStyles;// по умолчанию [];
type TComboBoxExStyleEx = (
 csExCaseSensitive,    //чувствительность к регистру
 csExNoEditImage,      //запрет на смену изображения
 csExNoEditImageIndent,//запрет на управление отступом Indent
 csExNoSizeLimit,      //снятие ограничения на размер
 csExPathWordBreak //путь к папке с файлами закрывается наклонной чертой
);
```

Кроме того, на особенности поведения комбинированного списка с расширенными возможностями оказывает свойство

```
property AutoCompleteOptions: TAutoCompleteOptions;//[acoAutoAppend];
```

Свойство определяет поведение элемента управления во время редактирования данных в строке ввода:

```
type TAutoCompleteOption = (
   acoAutoSuggest, //автоматический подбор текста из списка
   acoAutoAppend,  //включение режима автозаполнения
   acoSearch,      //добавление искомого элемента в список
   acoFilterPrefixes,{запрет стандартных префиксов, например, таких как
                      "www." и "http://" }
   acoUseTab, //использование клавиши <Tab> для выбора элементов
   acoUpDownKeyDropsList, //применение клавиш <Up> и <Down>
   acoRtlReading);
```

Из специфичных для элемента событий следует выделить пару событий:

```
property OnBeginEdit: TNotifyEvent;
property OnEndEdit: TNotifyEvent;
```

Они генерируются в момент начала и по завершению редактирования текста в строке.

Список просмотра *TListView*

Среди всех представленных в этой главе элементов управления самыми широкими возможностями обладает список просмотра `TListView`. Главный козырь списка — умение представлять свои элементы в виде значков с текстовыми подписями. Классическим примером использования графического списка на практике служит программа Проводник из комплекта Windows. В этой программе список позволяет отобразить большие или малые значки каталогов и файлов, вывести подробную информацию о них, осуществить сортировку и сделать многое другое.

Основу расширенного списка составляет его свойство

```
property Items: TListItems;
```

Это контейнер, в котором элемент управления хранит свои данные — элементы списка (экземпляры класса `TListItem`). Щелчок по свойству `Items` (или выбор пункта **Items Editor** в контекстном меню компонента) выводит окно редактора (рис. 21.5), который позволяет добавлять, редактировать и удалять элементы списка.

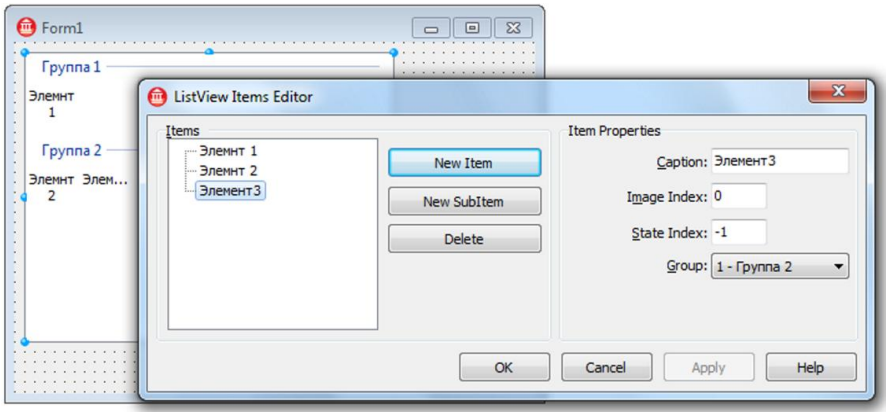


Рис. 21.5. Список `TListView` с активным редактором элементов

Стиль представления данных

В первую очередь внешний вид компонента `TListView` определяется текущим состоянием свойства, отвечающим за вид компонента:

```
property ViewStyle: TViewStyle;
```

Описание стилей вы найдете в табл. 21.8.

Таблица 21.8. *Стили представления данных* `TViewStyle`

Значение	Описание
`vsIcon`	Стиль "Крупные значки". Элементы списка отображаются в виде больших рисунков. Под значком выводится текстовый заголовок элемента. Элементы в списке упорядочены слева направо, сверху вниз
`vsSmallIcon`	Стиль "Маленькие значки". Значки элементов списка малого размера. Заголовок элемента размещен справа от картинки. Элементы в списке упорядочены слева направо, сверху вниз
`vsList`	Стиль "Список". Значки элемента маленького размера. Заголовок элемента размещен справа от картинки. Элементы в списке упорядочены сверху вниз, слева направо
`vsReport`	Стиль "Отчет". Информация об элементах списка представляется в многоколоночном виде. В первой колонке отображается малый значок элемента и его заголовок. Количество колонок определяется свойством `Columns`, порядок заполнения колонок задается в коде программы

Три свойства предназначены для организации взаимодействия между списком просмотра и набором картинок `TImageList`:

```
property LargeImages: TCustomImageList; //большие значки
property SmallImages: TCustomImageList; //маленькие значки
property StateImages: TCustomImageList; //значки состояния
```

Выбор отображаемого рядом с заголовком элемента `TListItem` значка осуществляется в соответствии со свойством

```
property ImageIndex: TImageIndex;   //основной значок
property OverlayIndex: TImageIndex; //значок маски
property StateIndex: TImageIndex;   //значок состояния
```

Если вид компонента `ViewStyle` установлен в `vsIcon` или `vsSmallIcon`, то способ упорядочивания значков также определяется свойством

```
property IconOptions: TIconOptions;
```

При желании программист способен научить расширенный список отображать рядом с элементом списка флажок (см. класс `TCheckBox`).

```
property Checkboxes: Boolean;
```

Но при этом свойство `ViewStyle` компонента не должно быть равным `vsIcon`.

Интересных визуальных эффектов можно добиться, установив в `true` свойство

```
property HotTrack: Boolean;
```

В этом случае элемент списка начнет реагировать на обычное перемещение курсора мыши над своей поверхностью. Время (в миллисекундах), проходящее с момента появления над элементом курсора мыши до момента его выделения, устанавливается в свойстве

```
property HoverTime: Integer;
```

Характер поведения элемента (указатель в виде руки, подчеркивание и т. п.) определяется состоянием свойства

```
property HotTrackStyles: TListHotTrackStyles;
type TListHotTrackStyle = (htHandPoint, htUnderlineCold, htUnderlineHot);
```

Особенности работы списка со стилем *vsReport*

При работе в режиме "Отчет" (стиль `vsReport`) данные графического списка отображаются в многоколоночном виде. Первая колонка компонента содержит значок и заголовок элемента. Состав остальных колонок зависит от пожеланий программиста и определяется в свойстве

```
property Columns: TListColumns;
```

Количество колонок и их заголовки могут быть настроены как во время визуального проектирования, так и во время выполнения программы.

Доступ к отдельной колонке организуется при посредничестве свойства

```
property Items[Index: Integer]: TListColumn; default;
```

Каждая колонка создается на основе класса `TListColumn`.

Общее количество колонок хранится в свойстве

```
property Count: Integer; //только для чтения
```

Для того чтобы отказаться от показа заголовков колонок, надо установить в `false` свойство

```
property ShowColumnHeaders: Boolean;
```

При необходимости можно запретить (`false`) или разрешить (`true`) показ линий, разделяющих колонки.

```
property GridLines: Boolean;
```

Если свойство `ViewStyle` установлено `vsReport`, то в свойстве

```
property VisibleRowCount: Integer;
```

вы можете узнать количество видимых строк.

Колонка *TListColumn*

В коллекции `TListColumns` списка `TListView` каждая колонка идентифицируется своим индексом:

```
property Index: Integer;
```

Колонки располагаются слева направо, нулевое значение индекса соответствует самой левой колонке.

Заголовок колонки текстом описывается свойством

```
property Caption : string;
```

Способ выравнивания текста внутри определяется свойством:

```
property Alignment : TAlignment;
```

Предусмотрено несколько способов назначения ширины колонки. Простейший из вариантов — положиться на Delphi и просто установить в `true` свойство:

```
property AutoSize : Boolean;
```

В этом случае горизонтальный размер колонки станет подстраиваться под ширину текста автоматически. Второй способ определения ширины колонки подчинен свойству

```
property Width : TWidth;
type TWidth = ColumnHeaderWidth..MaxInt;//см. табл. 21.9
```

***Таблица 21.9.** Определение ширины колонки*

Константа `TWidth`	Значение	Описание
`ColumnTextWidth`	`-1`	Горизонтальный размер подстраивается под ширину текста
`ColumnHeaderWidth`	`-2`	Размер определяется шириной заголовка колонки
	`0` и более	Все значения от 0 и выше определяют ширину колонки в пикселах

Ограничения на минимальный и максимальный размеры колонки задаются свойствами:

```
property MaxWidth : TWidth; //максимальный размер
property MinWidth : TWidth; //минимальный размер
```

Коллекция элементов списка *TListItems*

Нам уже известно о том, что все элементы графического списка `TListView` содержатся в свойстве `Items`, описываемом коллекцией `TListItems`. Уделим несколько минут обсуждению свойств и методов этой коллекции.

Для получения доступа к отдельному элементу списка надо воспользоваться свойством

```
property Item[Index: Integer]: TListItem;
```

В качестве параметра `Index` передается порядковый номер интересующего нас элемента.

Общее количество элементов в списке хранится в свойстве

```
property Count: Integer; //только для чтения
```

Индекс выбранного элемента хранится в свойстве

```
property ItemIndex: Integer;
```

Для получения сведений о самом первом видимом элементе воспользуйтесь свойством

```
property TopItem: TListItem;
```

но имейте в виду, что, во-первых, в процессе скроллинга верхний элемент меняется, и во-вторых, смысл использовать это свойство есть только в случае, когда свойство `ViewStyle` принимает значение `vsReport` или `vsList`.

Если свойство `ViewStyle` установлено в `vsIcon` или `vsSmallIcon`, то можно узнать экранные координаты левого верхнего угла расширенного списка.

```
property ViewOrigin: TPoint;
```

Если пользователь не дотрагивался до полос прокрутки, то свойство вернет точку с координатами (0, 0).

Каждый элемент знает своего владельца:

```
property Owner: TCustomListView;
```

Для добавления в список очередного элемента воспользуйтесь методом

```
function Add: TListItem;
```

или

```
function AddItem(Item: TListItem; Index: Integer = -1): TListItem;
```

Функция добавит элемент к концу списка и вернет на него ссылку.

Предусмотрена возможность вставки элемента `TListItem` в место, определенное индексом `Index`

```
function Insert(Index: Integer): TListItem;
```

Удаление элемента списка осуществляется методом

```
procedure Delete(Index: Integer);
```

Для удаления всего перечня элементов используйте метод

```
procedure Clear;
```

Как правило, операции, связанные со вставкой или удалением большого числа узлов, принято начинать методом

```
procedure BeginUpdate;
```

и завершать

```
procedure EndUpdate;
```

В таком случае исключаются лишние операции перерисовки поверхности компонента и, соответственно, сокращается расход драгоценных системных ресурсов компьютера.

Каждый элемент расширенного списка способен вернуть свой порядковый номер:

```
function IndexOf(Value: TListItem): Integer;
```

Если элемент не принадлежит расширенному списку, то функция возвратит –1.

Элемент списка *TListItem*

Визуально элемент `TListItem` представляет собой комбинацию значка и заголовка, а в случае когда расширенный список отображается в табличном виде (стиль `vsReport`), то элемент списка в дополнение ко всему выводит и набор строк.

Заголовок элемента вносится в свойство

```
property Caption: string;
```

К каждому элементу допускается подключить дополнительный набор строк:

```
property SubItems: TStrings;
```

Строки из набора `SubItems` выводятся в колонках компонента, только если свойство `ViewStyle` последнего установлено в `vsReport`.

Если элемент выбран пользователем, то его свойство

```
property Selected: Boolean;
```

устанавливается в `true`.

Находящийся в фокусе узел просигнализирует об этом, если у него установлено в `true` свойство

```
property Focused: Boolean;
```

Индекс элемента в списке хранится в свойстве

```
property Index : Integer;
```

Если вы активировали механизм группировки (свойство `Groups` компонента `TListView`), то каждый элемент может быть введен в определенную группу. Для этого надо передать идентификатор группы в свойство

```
property GroupID: Integer;
```

С любым элементом списка можно связать данные, для этого предназначен указатель

```
property Data: Pointer;
```

Если свойство `Checkboxes` графического списка установлено в `true` (компонент `TListView` превратился в симбиоз списка и флажка `TCheckBox`), то мы можем выяснить, включен или отключен флажок у конкретного элемента из его свойства

```
property Checked: Boolean;
```

Программист обладает возможностью указать место размещения любого из элементов в клиентской области расширенного списка. Для этого надо изменить значения свойств

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

или

```
property Position: TPoint;
```

Вместо вышеуказанных свойств, определяющих положение элемента, можно использовать методы

```
function GetPosition : TPoint;
procedure SetPosition(const Value: TPoint);
```

Первый метод указывает на текущую позицию элемента. Второй метод устанавливает положение объекта.

> ***Замечание***
>
> Изменить расположение элемента возможно, только если свойство `ViewStyle` родительского элемента управления `TListView` установлено в `vsIcon` или `vsSmallIcon`.

Редактирование заголовка элемента

Для перевода заголовка элемента `TListItem` в режим редактирования необходимо вызвать метод

```
function EditCaption: Boolean;
```

В случае если модификация заголовка невозможна, то функция вернет `false`.

Вызвав метод

```
procedure CancelEdit;
```

мы откажемся от редактирования и вернем текст заголовка в исходное состояние.

Сразу перед началом редактирования у элемента управления `ListView` возникает событие

```
property OnEditing: TLVEditingEvent;
type TLVEditingEvent = procedure(Sender: TObject; Item: TListItem;
                                 var AllowEdit: Boolean) of object;
```

Изменяя параметр `AllowEdit`, программист может разрешить (`true`) или запретить (`false`) редактировать заголовок. После завершения редактирования вызывается обработчик события

```
property OnEdited: TLVEditedEvent;
type TLVEditedEvent = procedure(Sender: TObject; Item: TListItem;
                                var S: string) of object;
```

В параметре `S` находится новое значение заголовка.

Для самоликвидации элемент может вызвать процедуру

```
procedure Delete;
```

В момент удаления узла свойство

```
property Deleting: Boolean; //только для чтения
```

переводится в состояние `true`.

Выбор элементов списка

По умолчанию пользователю запрещен одновременный выбор нескольких элементов списка `TListView`. Для отмены этого ограничения установите в `true` свойство

```
property MultiSelect: Boolean;
```

Для одновременного выбора всех элементов воспользуйтесь методом

```
procedure SelectAll;
```

Общее число выбранных объектов контролирует свойство

```
property SelCount: Integer;
```

Доступ к первому выделенному элементу предоставляет свойство

```
property Selected: TListItem;
```

Для того чтобы снять выделение со всех элементов, воспользуйтесь свойством

```
procedure ClearSelection;
```

Метод

```
procedure DeleteSelected;
```

удалит все выделенные элементы.

Для копирования выбранных элементов списка вызовите метод

```
procedure CopySelection(Destination: TCustomListControl);
```

Обратите внимание, что в качестве получателя узлов должен выступать потомок класса `TCustomListControl` — `Destination`.

Близкая по духу процедура осуществляет перемещение выделенных элементов в компонент `Destination`

```
procedure MoveSelection(Destination: TCustomListControl);
```

По умолчанию при потере фокуса ввода расширенный список перестает "подсвечивать" выделенные элементы. Чтобы этого не происходило, установите в `false` свойство:

```
property HideSelection: Boolean;
```

Хотя расширенный список и позволяет одновременно выделить несколько своих элементов, но активным может быть только один элемент (например, редактируемый пользователем). Для того чтобы обратиться к активному элементу, надо работать со свойством

```
property ItemFocused: TListItem;
```

Если на данный момент нет элемента, находящегося в фокусе, свойство вернет неопределенный указатель `nil`.

Процесс выбора элемента списка сопровождается парой событий. В момент выделения элемента генерируется событие

```
property OnChanging: TLVChangingEvent;
type TLVChangingEvent = procedure(Sender: TObject; Item: TListItem;
          Change: TItemChange; var AllowChange: Boolean) of object;
```

Событие хранит ссылку на выделяемый элемент `Item`, параметр `Change` конкретизирует, какая часть элемента (`ctText`, `ctImage`, `ctState`) была выделена, параметр-переменная `AllowChange` позволяет разрешить (`true`) или запретить (`true`) выделение.

В случае успешного выбора элемента вызывается завершающий обработчик события

```
property OnChange: TLVChangeEvent;
type TLVChangeEvent = procedure(Sender: TObject; Item: TListItem;
                                Change: TItemChange) of object;
```

Аргументы процедуры вам уже знакомы, поэтому в особых комментариях не нуждаются.

Упорядочивание элементов

Расширенный список предлагает несколько способов упорядочивания своих элементов. Для этого в первую очередь нам понадобится свойство

```
property SortType: TSortType;
type TSortType = (stNone, //сортировка отключена
                  stData, //сортировка по данным Data
          stText, //сортировка по тексту в заголовках элементов
          stBoth);//сортировка и по тексту, и по данным
```

Для простой сортировки элементов в соответствии с алфавитным порядком воспользуйтесь методом

```
function AlphaSort: Boolean;
```

В случае успешного завершения процесса сортировки функция вернет `true`. Для сортировки элементов списка метод `AlphaSort()` производит попарное сравнение текстовых заголовков элементов. Более серьезные возможности по сортировке предоставляет событие

```
property OnCompare: TLVCompareEvent;
type TLVCompareEvent = procedure(Sender: TObject;
 Item1, Item2: TListItem; Data: Integer; var Compare: Integer) of object;
```

Здесь `Item1` и `Item2` — сравниваемые элементы списка, параметр `Compare` — определяемый программистом результат сравнения.

Поиск элементов

У компонента `TListView` имеется метод, позволяющий найти узел `TListItem` по его заголовку (свойство `Caption`):

```
function FindCaption(StartIndex: Integer; Value: string;
                     Partial, Inclusive, Wrap: Boolean): TListItem;
```

С помощью функции ищется элемент, текст заголовка которого идентичен значению аргумента `Value`. Если параметр `Inclusive` установлен в `true`, то поиск стартует с элемента с индексом `StartIndex`, в противном случае с элемента `StartIndex+1`. Если в `true` установлено свойство `Partial`, то поиск будет останавливаться даже на тех элементах, где `Value` является подстрокой в заголовке `caption` элемента списка `TListItem`. Если в `true` установлен параметр `Wrap` и поиск начинался не с первой позиции списка, то в случае, если в оставшейся части списка искомый элемент не найден, поиск автоматически возобновится с начала расширенного списка. В случае успеха функция возвращает найденный элемент `TListItem`, иначе — `nil`.

Похожий метод обеспечивает поиск по связанным с узлом данным:

```
function FindData(StartIndex: Integer; Value: Pointer;
                  Inclusive, Wrap: Boolean): TListItem;
```

Назначение параметров идентично функции `FindCaption()`, единственное исключение составляет то, что ищется узел с данными, на которые ссылается указатель `Value`.

Группировка элементов

В современных версиях Delphi у компонентов `TListView` появилась дополнительная возможность — группировка элементов. Для создания групп следует воспользоваться услугами свойства

```
property Groups: TListGroups;
```

Свойство `Groups` предоставляет доступ к коллекции групп `TListGroups`, каждый элемент коллекции описывается классом `TListGroup` (табл. 21.10).

> ***ВНИМАНИЕ!***
> Для того чтобы включить элемент списка `TListItem` в ту или иную группу, следует воспользоваться его свойством `GroupID`.

Для включения режима группировки следует установить в состояние `true` свойство

```
property GroupView: Boolean;
```

В противном случае список не станет отображать сгруппированные элементы.

Таблица 21.10. *Основные свойства группы* `TListGroup`

Свойство	Описание
`property Header: string;`	Верхний колонтитул группы
`property Footer: string;`	Нижний колонтитул группы
`property GroupID: Integer;`	Обязательный уникальный идентификатор
`property State: TListGroupStateSet;`	Состояние группы
`property HeaderAlign: TAlignment;`	Выравнивание верхнего колонтитула
`property FooterAlign: TAlignment;`	Выравнивание нижнего колонтитула
`property Subtitle: string;`	Подзаголовок
`property TitleImage: TImageIndex;`	Изображение заголовка
`property GroupView: Boolean;`	Включение/отключение показа группы
`property GroupHeaderImages: TCustomImageList;`	Коллекция изображений заголовков

> ***ЗАМЕЧАНИЕ***
> Режим группировки теряет работоспособность, если выбран режим представления данных в виде списка (`ViewStyle=vsList`).

Операции перерисовки

Класс `TListView` вооружен целым арсеналом событий, связанных с процессом перерисовки своей поверхности. Почти все методы расширенного списка решают одну и ту же задачу обычным и "продвинутым" (`OnAdvanced...`) способами. Выбор того или иного обработчика события определяется характером поставленной перед нами задачи.

Наибольшей популярностью пользуется обработчик события

```
property OnDrawItem: TLVDrawItemEvent;
type TLVDrawItemEvent = procedure(Sender: TCustomListView;
        Item: TListItem; Rect: TRect; State: TOwnerDrawState) of object;
```

Кстати, это единственное событие, не имеющее своей пары, — `Advanced...` Параметры обработчика события классические, и в своем большинстве они встретятся и в остальных событиях перерисовки расширенного списка. Здесь: параметр `Sender` указывает на расширенный список, обновляющий свою поверхность; `Item` — элемент списка, перерисовка которого сейчас начнется; `Rect` — прямоугольные координаты элемента списка; `State` — текущее состояние этого элемента. Например, значение `odSelected` свидетельствует, что он выбран пользователем, `odInactive` — элемент неактивен, `odFocused` — находится в фокусе ввода.

Генерация события `OnDrawItem()` тесно связана со значением свойства

```
property OwnerDraw : Boolean;
```

Это ключ, разрешающий (`True`) или запрещающий (`False`) вызов этого события. На поведение остальных шести событий перерисовки `OnDraw...` свойство влияния не оказывает.

Если необходимо решать простейшие задачи, связанные, например, с обычной заменой цвета фона `ListView`, то для этого достаточно применять `OnCustomDrawItem()` — самый "приземленный" обработчик события. Это событие возникает сразу после получения компонентом от операционной системы сообщения `WM_Paint` или вызова любого другого метода, инициирующего перерисовку.

```
property OnCustomDraw : TLVCustomDrawEvent;
type TLVCustomDrawEvent = procedure(Sender: TCustomListView;
                const ARect: TRect; var DefaultDraw: Boolean) of object;
```

> ***Замечание***
>
> Во всех шести событиях, генерируемых во время перерисовки элемента управления, встречается переменная `DefaultDraw`. Передаваемое в нее значение определяет, каким образом процесс прорисовки компонента продолжится после выполнения события. Если в параметр передается `true`, то по завершению события обязанности по дальнейшему рисованию объекта перехватит операционная система, значение `false` "заморозит" клиентскую область компонента в текущем состоянии.

Здесь параметр `Sender` указывает на расширенный список, выполняющий перерисовку; в структуре `ARect` содержатся прямоугольные координаты перерисовываемой клиентской поверхности.

Если дизайнерскому таланту не дают покоя отдельные элементы списка `TListItem` (заголовок, значок, место вывода), то рекомендую попробовать свои силы в описании события

```
property OnCustomDrawItem: TLVCustomDrawItemEvent;
type TLVCustomDrawItemEvent = procedure(Sender: TCustomListView;
                              Item: TListItem; State: TCustomDrawState;
                              var DefaultDraw: Boolean) of object;
```

Процедура обладает параметром `Item`, информирующим нас о том, какой именно элемент в данный момент подлежит выводу. А благодаря `State`, мы узнаем текущее состояние этого элемента, например, выбран ли он пользователем (значение `cdsSelected`).

Безусловно, для того чтобы внести какие-то усовершенствования в процесс перерисовки элемента `TListItem` списка просмотра, необходимо выяснить координаты прямоугольной области этого элемента. С этой задачей превосходно справляется метод

```
function DisplayRect(Code: TDisplayCode): TRect;
```

Учитывая, что визуально элемент `ListItem` интегрирует в себе значок и заголовок, в методе предусмотрен параметр, определяющий, какая именно часть элемента нас интересует.

```
type TDisplayCode = (
   drBounds,       {прямоугольник, объединяющий заголовок и значок узла}
   drIcon,         {прямоугольная область значка элемента}
   drLabel,        {прямоугольная область вокруг заголовка}
   drSelectBounds);{прямоугольная область значки и заголовка,
                    исключая колонки в табличном режиме просмотра}
```

Если элемент управления функционирует в режиме таблицы (`vsReport`), то особенности вывода на экран колонок могут быть определены в событии

```
property OnCustomDrawSubItem: TLVCustomDrawSubItemEvent;
```

```
type TLVCustomDrawSubItemEvent = procedure(Sender: TCustomListView;
             Item: TListItem; SubItem: Integer; State: TCustomDrawState;
             var DefaultDraw: Boolean) of object;
```

В нем появился еще один параметр — `SubItem`, идентифицирующий индекс колонки, к рисованию которой сейчас приступит Windows.

В завершении упомянем о тройке "продвинутых" событий, связанных с операциями перерисовки.

```
property OnAdvancedCustomDraw       : TLVAdvancedCustomDrawEvent;
property OnAdvancedCustomDrawItem   : TLVAdvancedCustomDrawItemEvent;
property OnAdvancedCustomDrawSubItem: TLVAdvancedCustomDrawSubItemEvent;
```

Все три события связывает то, что они могут применяться на любом из четырех этапов перерисовки расширенного списка, определяемых с помощью типа данных

```
type TCustomDrawStage = (cdPrePaint, //подготовка к графическому выводу
                         cdPostPaint,//по завершению вывода
            cdPreErase,   //предваряет процесс стирания изображения
            cdPostErase); //после процесса стирания
```

Пример работы с *TListView*

Реализуем полученные знания на практике — разработаем приложение, позволяющее собирать данные о дисках компьютера (рис. 21.6). Для этого нам понадобится следующий перечень элементов управления:

- список `ListView1:TListView`;
- коллекция изображений `ImageList1:TImageList`. Внесите в коллекцию значки, символизирующие гибкий, жесткий и сетевой диски, накопитель флэш-памяти и накопитель на оптических дисках.

Свою работу мы начнем с обработки события `OnShow()` главной формы проекта. В данном событии (листинг 21.9) нам предстоит подготовить список `ListView1` к работе.

Листинг 21.9. Инициализация проекта в момент показа главной формы

```
procedure TForm1.FormShow(Sender: TObject);
var LC: TListColumn; //экземпляр колонки
    LG:TListGroup;   //экземпляр группы
begin
  ListView1.ViewStyle:=vsReport; //режим представления данных — отчет
  ListView1.SmallImages:=ImageList1; //подключили коллекцию значков
```

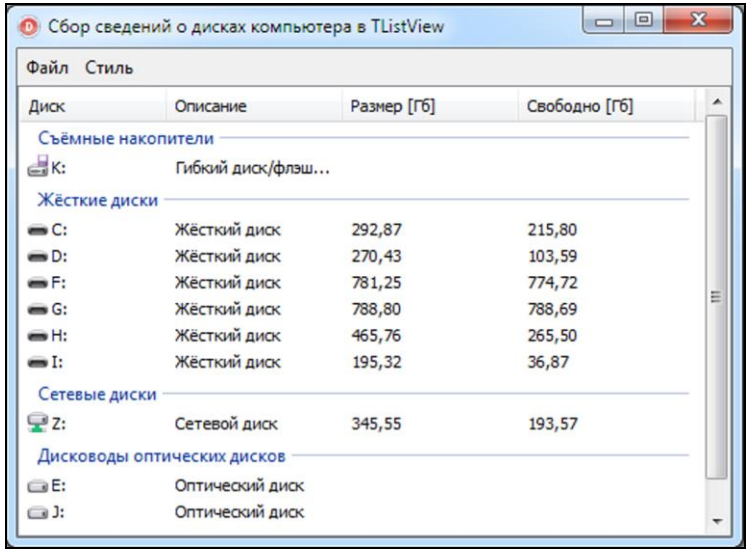


Рис. 21.6. Интерфейс приложения на основе `TListView`

```
with ListView1.Columns do //готовим колонки списка
begin
  Clear;   //удаляем все старые колонки
  LC:=Add; //добавляем новую колонку
    LC.Caption:='Диск';
    LC.Width:=100;
  LC:=Add; //добавляем новую колонку
    LC.Caption:='Описание';
    LC.Width:=120;
  LC:=Add; //добавляем новую колонку
    LC.Caption:='Размер [Гб]';
    LC.Width:=120;
  LC:=Add; //добавляем новую колонку
    LC.Caption:='Свободно [Гб])';
    LC.Width:=120;
end;
with ListView1.Groups do //готовим группы списка
begin
  Clear;
  LG:=Add; //новая группа
    LG.Header:='Съемные накопители'; //текст верхнего колонтитула
    LG.GroupID:=DRIVE_REMOVABLE;     //идентификатор группы
  LG:=Add;
    LG.Header:='Жесткие диски';
    LG.GroupID:=DRIVE_FIXED;
  LG:=Add;
    LG.Header:='Сетевые диски';
    LG.GroupID:=DRIVE_REMOTE;
  LG:=Add;
```

```
      LG.Header:='Дисководы оптических дисков';
      LG.GroupID:=DRIVE_CDROM;
    LG:=Add;
      LG.Header:='RAM-диск';
      LG.GroupID:=DRIVE_RAMDISK;
  end;

 // ShowDrives; //здесь будет осуществлен вызов процедуры сбора данных
end;
```

Вы наверняка обратили внимание на предпоследнюю строку листинга с комментарием о процедуре `ShowDrives()`, на которую будут возложены обязанности сбора данных о дисках компьютера. Это основная процедура нашего проекта, ее код представлен в листинге 21.10.

Листинг 21.10. Процедура сбора данных о дисках компьютера

```
procedure TForm1.ShowDrives;
const ByteInGb=1073741824; //2^30 — столько байтов в 1 гигабайте
var   Ch : ANSICHAR;       //буква имени диска
      DriveType : Integer; //тип диска
      AllBytes, FreeBytes : INT64; //размеры диска
      LI: TListItem;       //элемент списка
begin
  for Ch:='A' to 'Z' do //перебираем все символы (имена дисков)
  begin
    DriveType:=GetDriveType(PChar(Ch + ':\'));//тип диска
    if DriveType<=1
    then Continue; //если диск не обнаружен, то пропускаем шаг

    LI:=ListView1.Items.Add; //добавили элемент списка
    LI.Caption:=Ch+':';      //заголовок — буква диска

    case DriveType of //формируем пояснительные надписи
       DRIVE_REMOVABLE: begin
              LI.SubItems.Add('Гибкий диск/флэш-карта');
              LI.ImageIndex:=0; //зависит от индекса картинки!!!
              LI.GroupID:=DRIVE_REMOVABLE;//идентификатор группы
              end;
       DRIVE_FIXED :  begin LI.SubItems.Add('Жесткий диск');
              LI.ImageIndex:=1; LI.GroupID:=DRIVE_FIXED;
              end;
       DRIVE_REMOTE :  begin LI.SubItems.Add('Сетевой диск');
               LI.ImageIndex:=2; LI.GroupID:=DRIVE_REMOTE;
               end;
       DRIVE_CDROM  : begin LI.SubItems.Add('Оптический диск');
               LI.ImageIndex:=3; LI.GroupID:=DRIVE_CDROM;
               end;
        DRIVE_RAMDISK:  begin LI.SubItems.Add('RAM-диск');
               LI.ImageIndex:=4; LI.GroupID:=DRIVE_RAMDISK;
               end
```

```
      else     LI.SubItems.Add('???');
    end;
    {узнаем размер диска и свободное пространство}
  if GetDiskFreeSpaceEx(PChar(Ch + ':\'),FreeBytes,AllBytes,nil)=True Then
  begin
    LI.SubItems.AddObject(Format('%.2f',[AllBytes/ByteInGb]),
                          TObject(AllBytes div ByteInGb));
    LI.SubItems.AddObject(Format('%.2f',[FreeBytes/ByteInGb]),
                          TObject(FreeBytes div ByteInGb));
  end else
    begin
      LI.SubItems.AddObject('',TObject(0));
      LI.SubItems.AddObject('',TObject(0));
    end;
  end;
end;
```

В рамках процедуры `ShowDrives()` собирается информация о доступных дисках. Для этого с помощью функции `GetDriveType()` производится опрос типов дисков. Для отображения каждого найденного диска в списке `ListView1` создается новый элемент `TListItem`, в заголовок которого передается буква, идентифицирующая диск. Благодаря функции `GetDiskFreeSpaceEx()` получается дополнительная информация о размере диска и его свободном пространстве, указанные данные направляются в элемент `TListItem`. Обратите внимание на то, что данные о размере и свободном пространстве сохраняются не только в текстовом виде, но и в виде ссылки на данные. Такой подход позволит написать код, позволяющий упорядочивать элементы списка по размеру (листинг 21.11).

Листинг 21.11. Сортировка элементов списка

```
var SortMode:byte; //сохраняем индекс колонки
//...
uses math;
//...
procedure TForm1.ListView1ColumnClick(Sender: TObject;
                                      Column: TListColumn);
var i:byte;
begin
  SortMode := Column.Index; //передаем в переменную индекс колонки
  ListView1.AlphaSort;      //инициируем событие OnCompare()
end;

procedure TForm1.ListView1Compare(Sender: TObject;
     Item1, Item2: TListItem;  Data: Integer; var Compare: Integer);
begin
  case SortMode of
    0: Compare:=CompareText(Item1.Caption,Item2.Caption);
    1: Compare:=CompareText(Item1.SubItems[1],Item2.SubItems[1]);
```

```
    else Compare:=CompareValue(INT64(Item1.SubItems.Objects[SortMode-1]),
                               INT64(Item2.SubItems.Objects[SortMode-1]));
  end;
end;
```

Сортировка элементов списка осуществляется в два приема. По щелчку по колонке списка `OnColumnClick()` мы запоминаем в глобальной переменной `SortMode` индекс выбранной пользователем колонки. Далее (благодаря методу `AlphaSort`) генерируется второе событие — событие сортировки `OnCompare()`. Если щелчок был произведен по нулевой (имя диска) или по первой колонке (тип диска), то осуществляется сортировка по тексту. Если пользователь выбрал колонки 2 (размер диска) или 3 (свободное пространство на диске) — сортируем по числовым значениям.

ГЛАВА 22

Сетки

Очень многие современные программные продукты нуждаются в табличном представлении данных, среди них бухгалтерские приложения, статистические и аналитические программы, таблицы спортивных турниров, специализированные приложения для различных отраслей научных знаний (химия, физика, математика) и, конечно же, базы данных. Перечень примеров приложений, отображающих свои данные в двумерных таблицах, достаточно велик, но, пожалуй, наиболее показательным могут стать знаменитые электронные таблицы Excel корпорации Microsoft.

Для построения приложений на основе таблиц-сеток в составе VCL предложен целый ряд элементов управления, в этой главе мы с вами познакомимся с тремя наиболее показательными:

- сеткой для рисования `TDrawGrid`, специализирующейся на выводе графических данных. Сетка для рисования не умеет самостоятельно хранить данные, поэтому компонент `TDrawGrid` применяется в таких случаях, когда отображаемая в ячейках сетки информация содержится во внешнем хранилище, например в массиве;
- сеткой строк `TStringGrid`, предназначенной для табличного представления текстовых данных. Сетка строк обладает теми же возможностями, что и сетка для рисования, и вдобавок способна самостоятельно хранить данные;
- редактором списка значений `TValueListEditor`. Это специализированный компонент, предоставляющий пользователю хранить и редактировать некоторый набор значений.

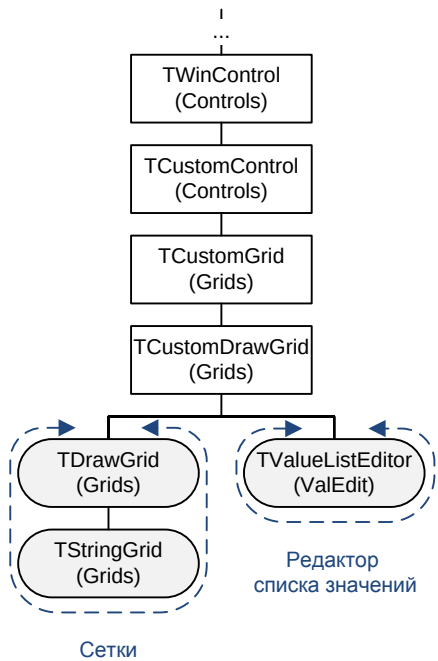


Рис. 22.1. Место сеток в иерархии классов VCL

Все перечисленные элементы берут начало от общих опорных классов `TCustomGrid` и `TCustomDrawGrid` (рис. 22.1).

Общие черты сеток, сетка *TDrawGrid*

Сетка для рисования `TDrawGrid` вобрала в себя качества, присущие всем базовым для сеток классам `TCustomGrid` и `TCustomDrawGrid`. Поэтому именно этот элемент управления будет рассмотрен первым.

Основу любой сетки составляют колонки и строки, как следствие, подавляющее большинство свойств и методов `TDrawGrid` прямо связаны с обслуживанием колонок, строк и отдельных ячеек сетки.

Число столбцов и строк сетки определяется свойствами

```
property ColCount: Longint; //по умолчанию 5 колонок
property RowCount: Longint; //по умолчанию 5 строк
```

Часть столбцов и строк сетки можно сделать неподвижными (фиксированными). Ячейки, принадлежащие фиксированной области сетки, недоступны для редактирования, не перемещаются с помощью полос прокрутки и обычно используются для размещения заголовков и комментариев.

```
property FixedCols: Integer; //по умолчанию 1 фиксированная колонка
property FixedRows: Integer; //по умолчанию 1 фиксированная строка
```

Отсчет фиксированных столбцов начинается слева направо, т. е. фиксированный столбец (столбцы) всегда располагается в левой части сетки. Отсчет фиксированных строк ведется сверху вниз, соответственно неподвижные ряды размещаются в верхней части сетки.

> ***Замечание***
>
> Для того чтобы отказаться от фиксированных колонок или строк, присвойте соответствующему свойству значение 0.

Фиксированные ряды и колонки на экране компьютера отображаются другим цветом, для изменения этого цвета используйте свойство

```
property FixedColor: TColor; //по умолчанию clBtnFace
```

Цвет фона рабочих ячеек традиционно определяется свойством

```
property Color: TColor; //по умолчанию clWindow
```

В простейшем случае оформление сетки зависит от состояния свойства

```
property DrawingStyle: TGridDrawingStyle;
type TGridDrawingStyle = (gdsClassic, //обычное оформление
                          gdsThemed,    //оформление в соответствии с темой
                          gdsGradient); //градиентная заливка
```

Назначаемые по умолчанию ширина столбца и высота строки сетки определяются в свойствах

```
property DefaultColWidth: Integer;  //по умолчанию 64
property DefaultRowHeight: Integer; //по умолчанию 24
```

Присвоение перечисленным свойствам каких-то значений "причесывает" все ячейки сетки под одну гребенку — они получат одинаковые размеры высоты и ширины. Вместе с тем, возможна и индивидуальная настройка ширины любой колонки и высоты строки. Для этого надо узнать их порядковый номер `Index` и передать его свойствам

```
property ColWidths[Index: Longint]: Integer; //ширина колонки
property RowHeights[Index: Longint]: Integer; //высота строки
```

Представленные свойства недоступны в Инспекторе объектов, поэтому управлять ими возможно только из кода программы.

Разделяющие ячейки линии также могут быть объектом настройки в части, касающейся их толщины:

```
property GridLineWidth: Integer; //по умолчанию 1 пиксел
```

Два доступных только для чтения свойства хранят сведения о высоте и ширине всей таблицы:

```
property GridHeight: Integer; //высота всей сетки
property GridWidth: Integer;  //ширина всей сетки
```

В случае если видимая область сетки не в состоянии вместить все ячейки, следует активировать полосы прокрутки:

```
property ScrollBars: TScrollStyle;
type TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth);
```

Количество полностью видимых в данный момент столбцов или строк можно получить из свойств

```
property VisibleColCount: Integer; //видимых колонок
property VisibleRowCount: Integer; //видимых рядов
```

При прокрутке сетки непомещающиеся в видимой области ячейки скрываются от взора пользователя. Для выяснения номера столбца и строки, соответствующих левой верхней видимой рабочей ячейке (не находящейся в фиксированной области), обращаемся к очередной паре свойств:

```
property LeftCol: Longint; //индекс левой ячейки
property TopRow: Longint;  //индекс верхней ячейки
```

Широкий спектр услуг по настройке поведения сетки обеспечивает множество `Options`, допустимые значения которого показаны в табл. 22.1.

```
property Options: TGridOptions;
```

Таблица 22.1. *Опции сетки* `TGridOption`

Значение	Результат
`goFixedVertLine`	Ячейки фиксированной области разделены вертикальными линиями
`goFixedHorzLine`	Ячейки фиксированной области разделены горизонтальными линиями
`goVertLine`	Колонки разделяются линиями
`goHorzLine`	Строки разделяются линиями
`goRangeSelect`	Разрешено одновременное выделение нескольких ячеек (опция несовместима с `goEditing`)
`goDrawFocusSelected`	Сфокусированная ячейка выделяется цветом
`goRowSizing`, `goColSizing`	Разрешение на установку индивидуальных размеров для строк и колонок
`goRowMoving`, `goColMoving`	Разрешение на перемещение строк и колонок мышью
`goEditing`	Разрешает пользователю редактировать текст ячеек

Таблица 22.1 *(окончание)*

Значение	Результат
goTabs	При включении опции переход между столбцами осуществляется при помощи клавиш <Tab> и <Shift>+<Tab>
goRowSelect	Запрещает выделение отдельной ячейки. Вместо этого выделяется весь ряд. При использовании этой опции отключается goAlwaysShowEditor
goAlwaysShowEditor	Имеет значение при включенной опции goEditing. Если опция включена, то, выбрав ячейку, пользователь сразу оказывается в режиме редактирования. В противном случае необходимо нажать клавиши <Enter> или <F2>
goThumbTracking	Определяет порядок прорисовки содержимого ячеек при перемещении бегунка полосы прокрутки сетки. Если опция отключена, то прорисовка начинается после отпускания кнопки мыши, в противном случае прорисовка осуществляется при малейшем перемещении бегунка
goFixedColClick	Сетка реагирует на щелчок по колонке из фиксированной области
goFixedRowClick	Сетка реагирует на щелчок строки из фиксированной области
goFixedHotTrack	Фиксированные колонки и ряды сетки подсвечиваются при перемещении над ними указателя мыши

Адресация ячейки

Каждая ячейка сетки однозначно адресуется номером столбца и строки. Традиционно отсчет начинается с нуля. Для выяснения координат выделенной ячейки обратитесь к свойствам

```
property Col: Longint;
property Row: Longint;
```

Эти же свойства допускают и обратную операцию — выделение ячейки программным образом.

Если режим работки сетки допускает одновременный выбор нескольких ячеек (см. опцию goRangeSelect в табл. 22.1), то диапазон ячеек станет доступен благодаря свойству

```
property Selection: TGridRect;
```

На этом способы выяснения координат ячейки не заканчиваются. Экранные координаты указателя мыши (X, Y) легко трансформируются в номера столбца и строки:

```
procedure MouseToCell(X, Y : Integer; var ACol, ARow : Longint);
```

Предусмотрен и обратный метод, возвращающий экранные координаты прямоугольной области соответствующей ячейки:

```
function CellRect(ACol, ARow : Longint): TRect;
```

Обработка событий

Наиболее важное событие сетки связано с выбором пользователем (например, с помощью указателя мыши или клавиш управления курсором) какой-либо ячейки в рабочей области сетки:

```
property OnSelectCell: TSelectCellEvent;
type TSelectCellEvent = procedure (Sender: TObject; ACol, ARow: Longint;
                                   var CanSelect: Boolean) of object;
```

Параметры ACol и ARow укажут колонку и строку, соответствующие выбираемой ячейке. Благодаря переменной CanSelect программист в состоянии запретить (false) или разрешить (true) пользователю выделить эту ячейку.

Во время просмотра сетки, содержащей большое число колонок и рядов, не помещающихся в видимой области сетки, зачастую бывает необходимо обеспечить реакцию на прокрутку пользователем ячеек. Для этого хорошо подходит событие

```
property OnTopLeftChanged: TNotifyEvent;
```

Событие генерируется в момент смены левой верхней видимой ячейки сетки.

Два события описывают реакцию сетки на перетаскивание колонок и строк

```
property OnColumnMoved: TMovedEvent;//перемещение колонки
property OnRowMoved: TMovedEvent;   //перемещение строки
type TMovedEvent = procedure (Sender: TObject;
                              FromIndex, ToIndex: Longint) of object;
```

Имейте в виду, что эти события генерируются только в том случае, если в опциях сетки (см. табл. 22.1) применяются флаги goColMoving и goRowMoving. В обработчике событий параметр FromIndex содержит старый индекс столбца/строки, ToIndex — новое значение индекса.

Сетка реагирует на щелчки по ячейкам, находящимся в фиксированной области сетки

```
property OnFixedCellClick: TFixedCellClickEvent;
type TFixedCellClickEvent = procedure(Sender: TObject;
                                      ACol, ARow: Longint) of object;
```

Обработчик события позволяет идентифицировать номера колонки и строки ячейки, по которой был произведен щелчок.

Несмотря на то, что сетка для рисования не в состоянии самостоятельно хранить текстовые данные, в ней объявлены три обработчика события, непосредственно связанные с вводом и редактированием текста. Эти методы целесообразнее использовать у ее потомка — компонента StringGrid, конечно при условии включения опции goEditing. При вводе текста первым этапом осуществляется проверка соответствия вводимых символов установленной маске.

```
property OnGetEditMask: TGetEditEvent;
type TGetEditEvent = procedure (Sender: TObject; ACol, ARow: Longint;
                                var Value: string) of object;
```

Маска описывается в параметре Value, а правила ее описания полностью соответствует маске класса TMaskEdit. Как только текст, ранее находящийся в ячейке, заносится в параметр Value, генерируется событие

```
property OnGetEditText: TGetEditEvent;
```

По завершению редактирования содержимого ячейки вызывается событие

```
property OnSetEditText: TSetEditEvent;
type TSetEditEvent = procedure (Sender: TObject; ACol, ARow: Longint;
                                const Value: string) of object;
```

На этом этапе вполне реально наложить дополнительные ограничения на вводимые данные или, например, преобразовать их к верхнему регистру.

Расширенные возможности по оформлению сетки

Важнейшим событием сетки для рисования является событие, возникающее в момент перерисовки ячейки сетки.

```
property OnDrawCell: TDrawCellEvent;
type TDrawCellEvent = procedure (Sender: TObject; ACol, ARow: Longint;
                          Rect: TRect; State: TGridDrawState) of object;
```

Рисование осуществляется на холсте (свойство `Canvas`) сетки изображений. В перечень параметров входят: `ACol`, `ARow` — координаты ячейки; `Rect` — прямоугольная область ячейки; `State` — аргумент, отражающий состояние ячейки и представляющий собой множество

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed,
                        gdRowSelected, gdHotTrack, gdPressed);
```

Если вы намерены полностью забрать бразды правления по прорисовке сетки в свои руки, то установите в `false` свойство

```
property DefaultDrawing: Boolean;//по умолчанию true
```

Если же свойство останется в состоянии `true`, то сетка постарается самостоятельно управлять графическим выводом.

Листинг 22.1 демонстрирует возможности сетки по графическому выводу данных. В листинге описаны два события. Событие `OnShow()` главной формы проекта позволяет настроить основные свойства сетки, обработчик события `OnDrawCell()` раскрашивает ячейки во все цвета радуги.

Листинг 22.1. Демонстрация графического вывода в сетке

```
procedure TForm1.FormShow(Sender: TObject);
begin
  with DrawGrid1 do{инициализация сетки DrawGrid1}
  begin
    Options:=Options+[goThumbTracking];
    DoubleBuffered:=true; //предварительный вывод в память
    DefaultDrawing:=false;//ручной режим прорисовки
    ColCount:=256;        //число колонок
    RowCount:=256;        //число строк
    DefaultColWidth:=2;   //ширина ячейки
    DefaultRowHeight:=2;  //высота ячейки
   end;
end;

procedure TForm1.DrawGrid1DrawCell(Sender: TObject; ACol, ARow: Integer;
  Rect: TRect; State: TGridDrawState);
begin
  with DrawGrid1.Canvas do
```

```
  begin
    if (aCol>DrawGrid1.FixedCols-1) and (aRow>DrawGrid1.FixedRows-1) then
      begin{прорисовка рабочих ячеек сетки}
        Brush.color:=RGB(255,Byte(aCol),Byte(aRow));//подбор цвета
        FillRect(Rect); //заливка ячейки
        if (gdSelected in State) then
          begin {если ячейка выбрана пользователем}
            Pen.Color:=NOT Brush.Color;
            Rectangle(Rect);
          end;
      end else
      begin {прорисовка фиксированных ячеек}
        Brush.color:=DrawGrid1.FixedColor;
        FillRect(Rect);
        if (aCol=0) then TextRect(Rect,Rect.Left,Rect.Top,IntToStr(aRow));
        if (aRow=0) then TextRect(Rect,Rect.Left,Rect.Top,IntToStr(aCol));
      end;
  end;
end;
```

Обратите внимание, каким образом назначается цвет заливки для ячеек сетки. Для этого используется макрос `RGB()`, конвертирующий номера колонок и рядов в значения зеленой и синей составляющих цвета, а красный цвет установлен в свой максимум.

Сетка строк *TStringGrid*

Компонент `TStringGrid` еще более совершенен, чем его прямой предок — сетка для рисования `TDrawGrid`. Основное отличие сетки строк от своего пращура (см. рис. 22.1) заключается в способности `TStringGrid` не просто отображать, но и хранить данные. Внешнее проявление этого качества отразилось в появлении нового свойства

```
property Cells[ACol, ARow: Integer]: string;
```

Теперь программисту позволено адресовать любую из ячеек сетки и передать в нее текстовое значение. Более того, сетка практически сразу готова предоставить пользователю сервис по редактированию текстовых данных в ячейках, правда, перед этим стоит активировать опцию `[goEditing]` (см. табл. 22.1).

> ***ЗАМЕЧАНИЕ***
>
> На первый взгляд может показаться, что способ хранения текста в сетке строк `TStringGrid` основан на идее двумерного массива, однако это впечатление ошибочно. На самом деле, данные хранятся в `TStringSparseList` (это прямой наследник хорошо знакомого нам класса `TStrings`).

Еще один, более скоростной способ доступа к столбцу (строке) сетки реализован при посредничестве свойств

```
property Cols[Index: Integer]: TStrings; //элементы колонки Index
property Rows[Index: Integer]: TStrings; //элементы ряда Index
```

Например, таким образом можно быстро очистить всю сетку (листинг 22.2).

Листинг 22.2. Инициализация сетки строк

```
procedure TForm1.FormShow(Sender: TObject);
var aCol,aRow:Integer;
begin
  StringGrid1.FixedCols:=0;
  StringGrid1.ColCount:=7;
  {заполняем нулевую строку ("шапку" таблицы) названиями дней недели}
  StringGrid1.Cells[0,0]:='Пнд';
  StringGrid1.Cells[1,0]:='Вт';
  ...
  {очищаем строки сетки}
  for aRow:=1 to StringGrid1.RowCount-1 do StringGrid1.Rows[aRow].Clear;
end;
```

ЗАМЕЧАНИЕ

В Delphi имеется системный массив `FormatSettings.ShortDayNames`, в котором хранятся названия дней недели. Будьте внимательны — в этом массиве неделя начинается не с понедельника, а с воскресенья.

Элемент управления `TStringGrid` способен на очень многое. Проявив немного изобретательности, мы с легкостью превратим сетку строк в календарь (листинг 22.3).

Листинг 22.3. Сетка в роли календаря

```
uses DateUtils;
...
procedure TForm1.ShowMonthCalendar(aYear, aMonth: word);
var _DM,_DW,aDay:word;
    aCol,aRow:Integer;
begin
  //уточним, сколько дней в этом месяце
  _DM:=DaysInAMonth( AYear, AMonth);
  //узнаем день недели, с которого начинается месяц
  _DW:=DayOfTheWeek(EncodeDate(aYear,aMonth,1));
  aRow:=1; aCol:=_DW-1;
  for aDay := 1 to _DM do
    begin //заполняем ячейки сетки днями
      StringGrid1.Cells[aCol,aRow]:=IntToStr(aDay);
      Inc(aCol);
      if aCol>StringGrid1.ColCount-1 then
        begin
          aCol:=0;
          Inc(aRow);
        end;
    end;
end;
```

На вход процедуре `ShowMonthCalendar()` достаточно передать год и номер месяца, обо всем остальном позаботится цикл, в котором даты распределяются по заданным ячейкам сетки

строк (рис. 22.2). Обратите внимание на то, что для работы процедуры необходима помощь модуля `DateUtils`, в котором описаны вспомогательные функции по работе с датой и временем.

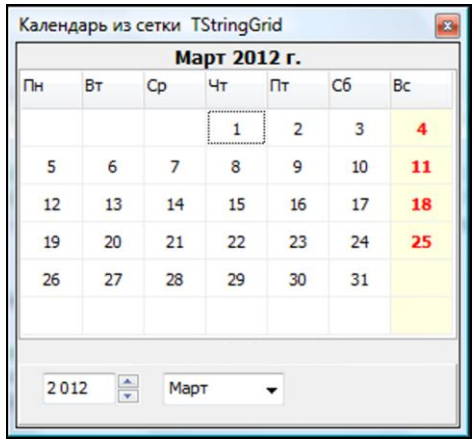


Рис. 22.2. Применение сетки строк в качестве календаря

Еще одно неоспоримое преимущество сетки `TStringGrid` над сеткой рисования заключается в умении сетки строк связывать с любой своей ячейкой ссылку на объект:

```
property Objects[ACol, ARow: Integer]: TObject;
```

Благодаря способности хранить объекты, сетка может решать весьма неординарные задачи.

Попробуем собрать в сетку картинки из поставляемого еще со времен первых версий Delphi набора изображений. В Embarcadero RAD Studio XE2 по умолчанию они расположены в каталоге C:\Program Files (x86)\Embarcadero\RAD Studio\9.0\Images\Buttons.

Сердцем нашей программы станет процедура `FindPictures()`, разыскивающая файлы с расширением имени bmp в папке `Directory`. Код процедуры представлен в листинге 22.4.

Листинг 22.4. Сбор растровых изображений в сетку `TStringGrid`

```
procedure TForm1.FindPictures(const Directory: string);
var SR: TSearchRec;
    I,aCol,aRow: integer;
begin
  aCol:=0;aRow:=0;
  I:=FindFirst(Directory+'*.bmp',faAnyFile,SR); //старт сбора данных
  while I=0 do
  begin
    StringGrid1.Cells[aCol,aRow]:=SR.Name;
    StringGrid1.Objects[aCol,aRow]:=TBitmap.Create;
    with StringGrid1.Objects[aCol,aRow] as TBitmap do
      LoadFromFile(Directory+SR.Name);
      INC(aCol);
      if aCol>StringGrid1.ColCount-1 then //закончились колонки
        begin
          INC(aRow); //переход на новую строку
```

```
          aCol:=0;
          if StringGrid1.RowCount-1<aRow then //если строк не хватает
            StringGrid1.RowCount:=StringGrid1.RowCount+1; //добавим строку
        end;
    I:=FindNext(SR); //переход к следующему файлу
  end;
  FindClose(SR); //конец сбора данных
end;
```

При обнаружении файла с растровой картинкой процедура `FindPictures()` передает имя файла в ячейку сетки с координатами [0, 0]. Затем процедура создает контейнер класса `TBitmap`, загружает в контейнер битовый образ и связывает экземпляр `TBitmap` с этой же ячейкой [0, 0]. Заполнив ячейку сетки, процедура переходит к следующей и повторяет операцию до тех пор, пока в каталоге не заканчиваются подходящие файлы.

Для того чтобы пользователь смог увидеть в сетке найденные растровые образы, достаточно напомнить сетке строк, что в ее жилах течет "кровь" сетки для рисования `TDrawGrid`. Это мы сделаем в рамках события `OnDrawCell()`, генерирующегося в момент прорисовки ячеек сетки (листинг 22.5).

Листинг 22.5. Отображение картинок в ячейках сетки строк

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
              ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
  if StringGrid1.Objects[ACol,ARow] is TBitmap then
    with StringGrid1.Canvas do
      Draw(Rect.Left+15,Rect.Top+20,
           TBitmap(StringGrid1.Objects[ACol,ARow]));
end;
```

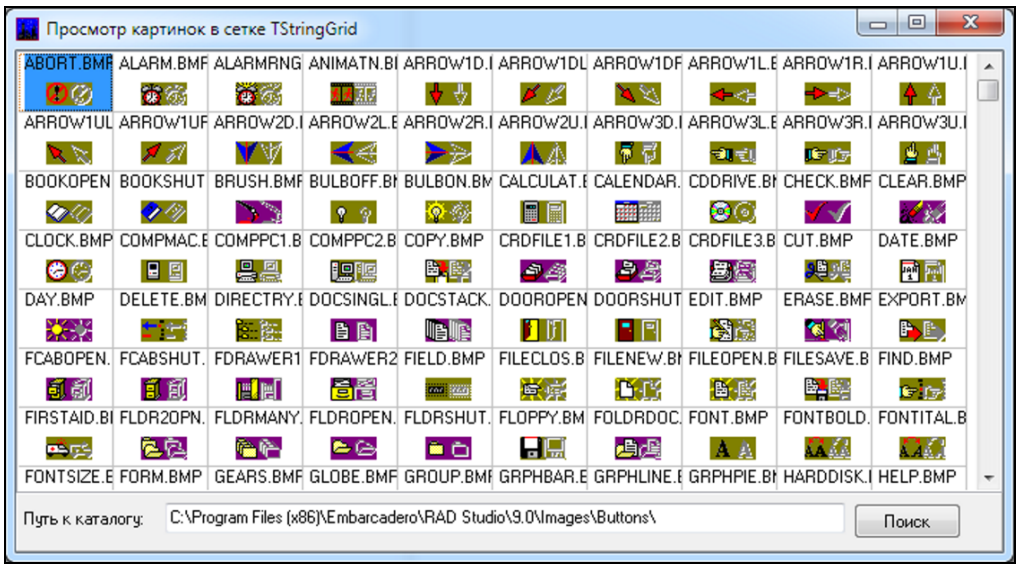


Рис. 22.3. Сетка строк в роли хранилища растровых образов

Мы неплохо потрудились! Нажав клавишу <F9>, вы увидите, как ячейки сетки заполнятся рисунками из коллекции Delphi (рис. 22.3).

Редактор списка значений *TValueListEditor*

Идея, заложенная в базовые классы сеток, оказалась настолько удачной, что на основе них кроме сетки для рисования и сетки строк были построены и другие полезные элементы управления. Сейчас мы познакомимся с одним из них — редактором списка значений `TValueListEditor`.

Как и следует из названия компонента, редактор списка значений предоставляет пользователю интерфейс для просмотра и внесения правок в определенный набор значений. Для обслуживания списка в состав объекта интегрирован список строк, доступ к которому осуществляется при посредничестве свойства

```
property Strings: TStrings;
```

Благодаря специальному форматированию каждая строка списка делится на две части: в первой половине строки хранится наименование параметра, во второй — соответствующее этому параметру текстовое значение.

> ***ЗАМЕЧАНИЕ***
>
> При работе со свойством `Strings` надо помнить, что наименование параметра и значение должны разделяться символом равенства (=).

Несмотря на наличие прямого доступа к списку строк через свойство `Strings`, в компоненте предусмотрен более удобный способ управления парами "наименование параметра = значение". Вставка нового ряда осуществляется методом

```
function InsertRow(const KeyName, Value: string; Append: Boolean): Integer;
```

Здесь программисту следует указать наименование (`KeyName`), значение (`Value`). Параметр `Append` определяет место добавляемой строки относительно текущей строки редактора (в состоянии `true` — после текущей, а в состоянии `false` — перед текущей). Пример работы метода предложен в листинге 22.6.

Листинг 22.6. Заполнение редактора во время показа формы

```
procedure TForm1.FormShow(Sender: TObject);
begin
  with ValueListEditor1 do //подготовка списка к работе
    begin
      TitleCaptions.Clear; //заголовки колонок
      TitleCaptions.Add('Свойство');
      TitleCaptions.Add('Описание');
      Strings.BeginUpdate;
      Strings.Clear;  //строки "параметр=значение"
      InsertRow('Фамилия','',true);
      InsertRow('Имя','',true);
      InsertRow('Пол','',true);
      InsertRow('Дата рождения','',true);
      InsertRow('Номер телефона','',true);
```

```
      Strings.EndUpdate;
    end;
end;
```

В представленном в листинге 22.6 примере мы не только заполнили редактор пятью парами "параметр = значение", но и назначили заголовки колонкам. Для этого мы воспользовались свойством

```
property TitleCaptions: TStrings;
```

Свойство не ограничивает число вводимых элементов, однако заголовками станут только первые две строки.

На рис. 22.4 представлен экранный снимок элемента управления `TValueListEditor` с заполненными значениями. Для понимания способа хранения данных в редакторе в правой части формы расположен многострочный редактор `TMemo`, он выводит неформатированное содержание пар "параметр = значение".

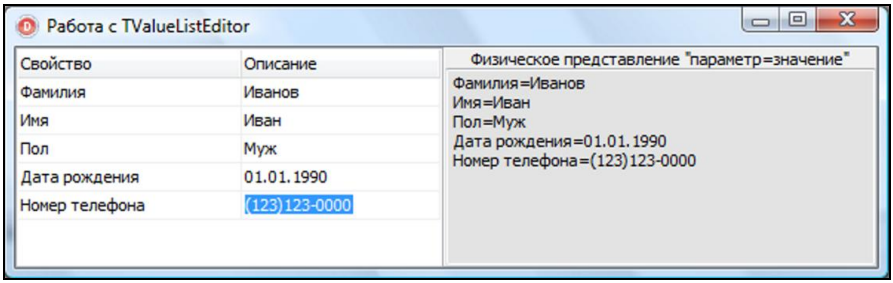


Рис. 22.4. Внешний вид редактора списка значений `TValueListEditor`

Удаление строки производится методом

```
procedure DeleteRow(ARow: Integer);
```

В данном случае достаточно назначить номер ненужной строки.

Для поиска строки по наименованию параметра следует воспользоваться услугами функции

```
function FindRow(const KeyName: string; var Row: Integer): Boolean;
```

В случае успеха метод возвратит значение `true` и передаст индекс обнаруженной строки в параметр-переменную `Row`.

Перебирая индексы редактора значений с помощью свойства

```
property Keys[Index: Integer]: string;
```

можно просмотреть все наименования параметров.

Для быстрого перехода к определенному значению можно воспользоваться свойством

```
property Values[const Key: string]: string;
```

Направив в аргумент `Key` наименование параметра, мы получим возможность чтения/записи выбранного значения.

В редакторе списка значений предусмотрен традиционный для сетки строк доступ к ячейкам с помощью свойства

```
property Cells[ACol, ARow: Integer]: string;
```

ВНИМАНИЕ!

Информирующие о числе колонок и строк свойства (`ColCount` и `RowCount`) компонента `TValueListEditor` доступны только для чтения и поэтому не могут определять размерность сетки.

Права пользователя по редактированию строк зависят от набора опций

```
property Options: TGridOptions;//по умолчанию []
```

В перечень входят четыре опции, представленные в табл. 22.2.

Таблица 22.2. *Описание опций* `TKeyOption`

Опция	Описание
`keyEdit`	Пользователю разрешается выбирать и редактировать колонку с наименованием параметра
`keyAdd`	Пользователь получит возможность добавлять новый ряд "параметр = значение" (необходимо включить опцию `keyEdit`)
`keyDelete`	Пользователю разрешается удалять строки из редактора (следует нажать клавишу <Delete>)
`keyUnique`	Запрет на создание одинаковых параметров (необходимо включить опцию `keyEdit`)

В табл. 22.3 представлены наиболее важные обработчики событий редактора списка значений.

Таблица 22.3. *Основные события* `TValueListEditor`

Событие	Описание
`property OnStringsChanging: TNotifyEvent;`	Событие генерируется перед началом внесения изменений в список строк "параметр = значение"
`property OnStringsChange: TNotifyEvent;`	Вызывается после завершения изменений в строках редактора
`property OnEditButtonClick: TNotifyEvent;`	Генерируется в момент нажатия кнопки в ячейке редактора (или при нажатии комбинации клавиш <Ctrl>+<Enter>)
`property OnValidate: TOnValidateEvent;` `type TOnValidateEvent = procedure(Sender: TObject; ACol, ARow: Longint; const KeyName, KeyValue: string) of object;`	Генерируется перед тем, как фокус ввода покидает ячейку редактора. Событие предназначено для проверки корректности данных, введенных в ячейку `[aCol, aRow]`. Параметры `KeyName` и `KeyValue` отражают содержимое параметра и значения
`property OnGetPickList: TGetPickListEvent;` `type TGetPickListEvent = procedure(Sender: TObject; const KeyName: string; Values: TStrings) of object;`	Событие предназначено для создания списка выбора, выпадающего из редактируемой ячейки. Константа `KeyName` идентифицирует название параметра, в список `Values` следует занести значения списка
`property OnGetEditMask: TGetEditEvent;` `type TGetEditEvent = procedure (Sender: TObject; ACol, ARow: Longint; var Value: string) of object;`	Позволяет создать маску ввода для ячейки с адресом `[aCol, aRow]`. Шаблон маски ввода следует передать в параметр `Value` (см. табл. 17.2)

В листингах 22.7—22.9 представлен исходный код для основных обработчиков событий компонента, он позволит усовершенствовать начатый в листинге 22.6 пример редактора

списка значений. Так листинг 22.7 демонстрирует порядок работы с событием создания маски ввода. Благодаря ему мы описываем шаблоны ввода для даты рождения и номера телефона.

Листинг 22.7. Создание выпадающего списка в ячейке ввода значения

```
procedure TForm1.ValueListEditor1GetEditMask(Sender: TObject;
                          ACol, ARow: Integer; var Value: string);
begin
  if aCol=1 then
  begin
    if ValueListEditor1.Cells[0,aRow]='Дата рождения'
                    then Value:='!99/99/0000;1;_'
    else
      if ValueListEditor1.Cells[0,aRow]='Номер телефона'
                    then  Value:='!\(999\)000-0000;1;#';
  end;
end;
```

Благодаря событию `OnGetPickList()` отвечающая за ввод пола человека ячейка сетки дополняется выпадающим списком (листинг 22.8). Список содержит два допустимых варианта ответа: "Муж" или "Жен".

Листинг 22.8. Создание маски ввода в ячейке ввода значения

```
procedure TForm1.ValueListEditor1GetPickList(Sender: TObject;
  const KeyName: string; Values: TStrings);
begin
  if KeyName='Пол' then begin
                        Values.Add('Муж');
                        Values.Add('Жен');
                      end;
end;
```

Предложенный в листинг 22.9 код контролирует корректность данных, введенных пользователем в ячейку. Если введенное значение отличается от "Муж" или "Жен", выводится сообщение об ошибке и ошибочное значение стирается.

Листинг 22.9. Проверка корректности значения в ячейке редактора

```
procedure TForm1.ValueListEditor1Validate(Sender: TObject;
                  ACol, ARow: Integer; const KeyName, KeyValue: string);
begin
  if (aCol=1) and (KeyName='Пол') and (KeyValue<>'') then
    if (UpperCase(KeyValue,loUserLocale)<>'МУЖ') AND
       (UpperCase(KeyValue,loUserLocale)<>'ЖЕН') then
    begin
       MessageBox(Application.Handle,
         'Недопустимое значение!', 'Ошибка!', MB_ICONERROR OR MB_OK);
       ValueListEditor1.Cells[aCol,aRow]:='';
    end;
end;
```

ГЛАВА 23

Иерархические данные и компонент *TTreeView*

Мы с вами уже являемся неплохими специалистами по самым разнообразным способам представления данных. Нам уже хорошо знаком относительно простой способ отображения текста в виде списка, мы подробно изучили очень популярный табличный вывод информации. Эта глава посвящена еще одному востребованному способу представления данных — в виде иерархической (древообразной) структуры.

Иерархические структуры распространены весьма широко: это и штатная структура предприятия, и дерево каталогов на жестком диске компьютера, и иерархия наследования классов в VCL. Можно продолжать приводить многочисленные примеры иерархически организованных данных, но лучше сразу заострить свое внимание на их объединяющей черте — наличии между элементами данных отношения "главный — подчиненный" (родительский — дочерний). Еще один важный отличительный признак иерархических структур заключается в том, что родительский узел дерева способен обладать неограниченным числом дочерних узлов. В свою очередь любой дочерний узел имеет право выступать владельцем целой ветви подчиненных узлов. Тот факт, что структура дерева заранее неизвестна, значительно затрудняет хранение иерархии в памяти и осуществление операций с узлами дерева. К счастью, в составе VCL имеется элемент управления, с легкостью решающий задачу обслуживания иерархических данных: герой этой главы — компонент `TTreeView`.

Как всегда, работу с новым классом начнем с анализа его иерархии наследования (рис. 23.1). Обратите внимание на то, что дерево нуждается в помощи двух вспомогательных классов:

- класса `TTreeNode`, на который возложено описание узла дерева, каждому вновь создаваемому узлу будет соответствовать отдельный экземпляр этого класса;
- класс `TTreeNodes`, который играет роль хранилища всех узлов, входящих в иерархическую структуру.

Доступ к хранилищу узлов обеспечивает ключевое свойство компонента

```
property Items: TTreeNodes;
```

Это не что иное, как посредник между нами и инкапсулированным в элемент управления объектом-контейнером `TTreeNodes`. Все манипуляции, производимые со свойством `Items`, обеспечиваются свойствами и методами, включенными в состав этого класса.

Щелчок по свойству `Items` в Инспекторе объектов или обращение к пункту контекстного меню **Items Editor** активизирует встроенный редактор узлов компонента (рис. 23.2). Редактор обладает интуитивно понятным интерфейсом. Кнопка **New Item** добавляет в дерево но-

вый узел на том же уровне, где находится выбранный элемент иерархии. Название узла следует ввести в строку ввода **Text**. Кнопка **New SubItem** добавляет к выбранному узлу дочерний узел. Кнопка **Delete** удаляет активный узел и все его потомство. Если дерево было подготовлено заранее и сохранено в файл, то для его загрузки потребуется помощь кнопки **Load**.

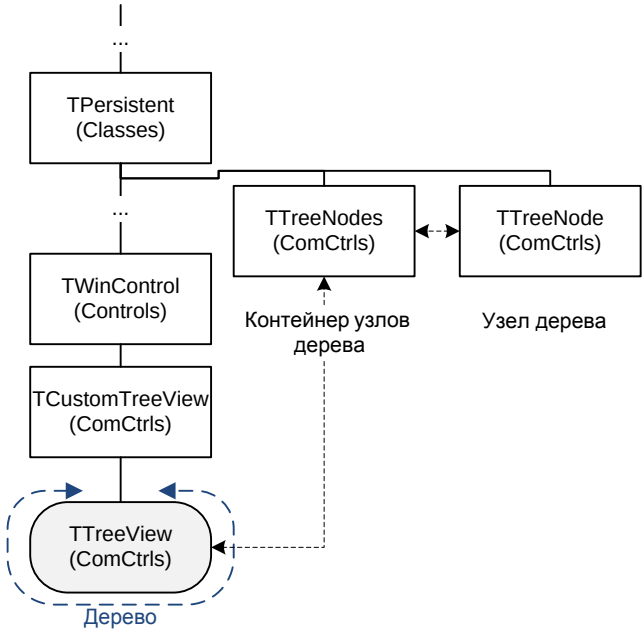


Рис. 23.1. Компонент `TTreeView` и вспомогательные классы в иерархии VCL

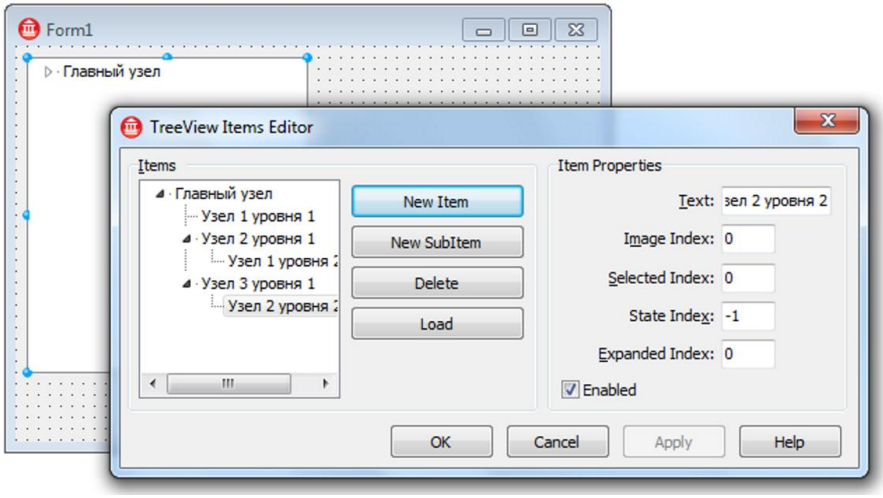


Рис. 23.2. Встроенный редактор узлов компонента `TTreeView`

Сохранение и загрузка дерева

Находящиеся в памяти компонента TTreeView узлы могут быть сохранены в файл либо в поток с помощью группы перегружаемых методов:

```
procedure SaveToFile(const FileName: string); overload;
procedure SaveToFile(const FileName: string; Encoding: TEncoding); overload;
procedure SaveToStream(Stream: TStream); overload;
procedure SaveToStream(Stream: TStream; Encoding: TEncoding); overload;
```

Обратите внимание на то, что расширенные версии методов сохранения данных позволяют конкретизировать кодовую страницу таблицы символов в параметре Encoding.

Вполне естественно, что предусмотрен набор "зеркальных" методов, решающих обратную задачу — загрузки набора узлов из файла или потока:

```
procedure LoadFromFile(const FileName: string); overload;
procedure LoadFromFile(const FileName: string; Encoding: TEncoding); overload;
procedure LoadFromStream(Stream: TStream); overload;
procedure LoadFromStream(Stream: TStream; Encoding: TEncoding); overload;
```

Формат файла с данными о составе узлов дерева особым изыском не отличается, это обычный текстовый файл, в каждой строке которого сохраняется отдельный заголовок. Заголовки узлов более низкого уровня сдвигаются с помощью символов табуляции: чем ниже уровень, тем больше символов табуляции.

> ***ВНИМАНИЕ!***
> При сохранении иерархической структуры в файл (в поток) направляются только сведения о названиях узлов дерева. Если с узлом связаны какие-то дополнительные данные (см. свойство Data), то они сохранены не будут.

Выбор узла в дереве

Элемент управления TTreeView знает, какой из узлов в данный момент выделен пользователем, ссылка на узел находится в свойстве

```
property Selected : TTreeNode;
```

Если ни один узел не выбран, то в Selected окажется неопределенный указатель nil.

По умолчанию выбор узла пользователем осуществляется с помощью клавиш управления курсором и по щелчку левой кнопкой мыши. К перечисленным возможностям можно добавить способ выбора узла щелчком правой кнопкой мыши. Для этого установите в true свойство

```
property RightClickSelect : Boolean;//по умолчанию false
```

Для снятия выделения с узла предназначена процедура

```
procedure Deselect(Node: TTreeNode);
```

В ходе выбора узла дерева последовательно вызывается пара событий. О начале процесса выбора сигнализирует событие

```
property OnChanging : TTVChangingEvent;
type TTVChangingEvent = procedure(Sender: TObject; Node: TTreeNode;
                                  var AllowChange: Boolean) of object;
```

В рамках данного события программист способен запретить выбор узла `Node`, для этого достаточно присвоить значение `false` параметру `AllowChange`.

Второе событие генерируется только в том случае, если пользователь действительно смог выбрать узел.

```
property OnChange: TTVChangedEvent;
type TTVChangedEvent = procedure(Sender: TObject;
                                 Node: TTreeNode) of object;
```

Ссылка на выделенный узел окажется в параметре `Node`.

Иногда требуется несколько отсрочить генерацию события `OnChage()`. Например, такая необходимость может возникнуть при отражении в дереве файловой структуры логического диска компьютера. Если ваша программа позволяет управлять файлами и папками, то следует дать операционной системе время на физическое создание (удаление и т. п.) папки или файла. В таком случае следует установить время задержки в свойство

```
property ChangeDelay: Integer; //по умолчанию 0 миллисекунд
```

> ***ЗАМЕЧАНИЕ***
>
> Все изменения в составе файлов и папок Проводник Windows отображает с задержкой в 50 миллисекунд.

При потере фокуса ввода компонент `TTreeView` перестает подсвечивать выделенные в нем узлы. Однако такое поведение можно изменить. Чтобы в этом случае эти элементы визуально продолжали показывать, что они выделены, установите в `false` свойство

```
property HideSelection: Boolean; //по умолчанию true
```

Полезный метод, проверяющий наличие узла в точке с координатами (`X`, `Y`):

```
function GetNodeAt(X, Y: Integer): TTreeNode;
```

Если узла нет, то метод возвратит пустой (неопределенный) указатель `nil`.

Еще большей степенью детализации обладает метод, преобразующий экранные координаты в формат `THitTests`:

```
function GetHitTestInfoAt(X, Y: Integer): THitTests;
```

Одновременный выбор нескольких узлов

Если логика работы программы предполагает возможность одновременного выделения нескольких узлов, то установите в `true` свойство компонента `TTreeView`

```
property MultiSelect : Boolean; //по умолчанию false
```

Весь перечень выделенных узлов окажется в свойстве

```
property Selections[Index: Integer]: TTreeNode;
```

О количестве выбранных пользователем узлов проинформирует свойство

```
property SelectionCount: Cardinal; //только для чтения
```

Для того чтобы конкретизировать, каким образом будет производиться подключение очередного узла к группе уже выбранных элементов, необходимо воспользоваться свойством

```
property MultiSelectStyle: TMultiSelectStyle; //[msControlSelect]
```

Описание TMultiSelectStyle предложено в табл. 23.1.

Таблица 23.1. *Особенности выбора нескольких узлов* `TMultiSelectStyle`

Значение	Описание
msControlSelect	Очередной узел включается в группу выделенных узлов, если в момент щелчка мышью по узлу пользователь удерживал в нажатом состоянии клавишу <Ctrl>
msShiftSelect	Если при щелчке по узлу удерживалась клавиша <Shift>, то в группу выбранных включаются все узлы, находящиеся между данным и предыдущим узлом
msVisibleOnly	То же самое, что и в режиме msShiftSelect, но в группу выбранных узлов не попадут дочерние элементы свернутых узлов
msSiblingOnly	Разрешается одновременно выбирать только узлы одного и того же уровня

Если возникает необходимость последовательно обратиться к каждому из выделенных узлов дерева, вызываем метод

```
function FindNextToSelect: TTreeNode;
```

Отправной точкой перебора станет выбранный пользователем узел (узел, на который ссылается свойство Selected).

Узел дерева *TTreeNode*

Узел дерева реализован на основе класса TTreeNode и предназначен для хранения данных. Для доступа к данным в классе объявлены два свойства:

```
property Text: string;
property Data: Pointer;
```

Свойство Text специализируется на обслуживании текстовой информации, отображаемой в качестве заголовка узла. Указатель Data — универсальное свойство, способное связать с узлом любой объект.

Узел может быть выбран пользователем, об этом просигнализирует свойство

```
property Selected: Boolean;
```

Это же свойство позволяет выделить узел программным образом.

Узел может находиться в фокусе:

```
property Focused: Boolean;
```

Если узел должен участвовать в операциях с буфером обмена, то полезно знать о существовании свойства

```
property Cut: Boolean;
```

Переведя свойство в состояние true, вы добьетесь визуального эффекта полупрозрачности узла, и это сигнализирует о том, что узел вырезан в буфер обмена.

Каждый узел содержит ссылку на коллекцию узлов, в состав которой он входит:

```
property Owner: TTreeNodes;
```

В узле хранится ссылка на владельца этого элемента — компонент-дерево:

```
property TreeView: TCustomTreeView;
```

Положение узла в дереве

Все узлы дерева пронумерованы, и их индивидуальный номер находится в свойстве

```
property AbsoluteIndex : Integer; //только для чтения
```

Самый верхний узел обладает нулевым абсолютным индексом, следующий за ним — первым и т. д. Абсолютный индекс не вечен, любая операция добавления, удаления или перемещения узлов внутри коллекции автоматически переиндексирует все узлы дерева.

Каждый узел знает, на каком уровне дерева он расположен, для этого следует обратиться к свойству

```
property Level: Integer;
```

Корневой узел иерархии размещен на нулевом уровне, первый его потомок — на первом, потомок потомка — на втором и т. д.

Родительские узлы

Каждый элемент дерева обязательно владеет всей информацией о своем родительском узле

```
property Parent: TTreeNode;
```

Если речь идет об узле самого верхнего уровня (а он не имеет родителя), то свойство `Parent` вернет неопределенный указатель `nil`.

В представленном в листинге 23.1 примере предложен способ выяснения пути от выбранного пользователем узла `Node` до самого верхнего узла в дереве.

Листинг 23.1. Построение пути от узла к вершине дерева

```
function GetPath(Node: TTreeNode): string;
var n : TTreeNode;
begin
  Result:='';
  if node=nil then exit;
  n:=Node;
  while n<>nil do //цикл выполняется, пока существует узел n
    begin
      Result:=n.Text+'/'+Result; //склеиваем строку пути
      n:=n.Parent;     //обращаемся к родительскому узлу
    end;
end;
```

Порядковый номер дочернего узла в родительском списке хранится в свойстве

```
property Index: Longint; //только для чтения
```

При необходимости можно проконтролировать всю цепочку предков, помощь в этом может оказать функция

```
function HasAsParent(Value: TTreeNode): Boolean;
```

Метод проверяет, не выступает ли узел `Value` предком (причем в любом поколении) по отношению к нашему узлу.

Дочерние узлы

О наличии дочерних узлов можно судить по свойству

```
property HasChildren: Boolean;
```

Если существует хотя бы один подчиненный узел, то свойство вернет значение `true`.

Доступ к списку дочерних узлов в соответствии с их индексом осуществляется при помощи свойства

```
property Item[Index: Integer]: TTreeNode;
```

О количестве дочерних узлов можно уточнить у свойства

```
property Count: Integer;
```

Родительский узел обладает возможностью упорядочить по алфавиту свои дочерние элементы

```
function AlphaSort(ARecurse: Boolean = False): Boolean;
```

Если единственный параметр метода установлен в состояние `false`, то сортировке подвергнутся только непосредственные потомки узла. Значение `true` укажет, что надо упорядочить все подчиненное дерево полностью.

Методы перехода между узлами дерева

В табл. 23.2 собраны ключевые методы узла `TTreeNode`, позволяющие ему обращаться к своим ближайшим соседям.

Таблица 23.2. *Методы перебора узлов дерева*

Метод	Описание
`function GetFirstChild: TTreeNode;`	Возвращает первый дочерний узел
`function GetNextChild(Value: TTreeNode): TTreeNode;`	Обращение к следующему после `Value` дочернему узлу
`function GetPrevChild(Value: TTreeNode): TTreeNode;`	Обращение к дочернему узлу, стоящему перед узлом `Value`
`function GetLastChild: TTreeNode;`	Возвращает последний дочерний узел
`function GetNext: TTreeNode;`	Возвращает следующий узел в дереве
`function GetPrev: TTreeNode;`	Предыдущий узел в дереве
`function GetNextSibling: TTreeNode;`	Возвращает следующий узел на том же уровне
`function GetPrevSibling: TTreeNode;`	Предыдущий узел на том же уровне

Один из примеров применения методов перебора узлов предложен в листинге 23.2. Здесь благодаря методу `GetNext()` мы последовательно просмотрим все узлы дерева.

Листинг 23.2. Перебор всех узлов дерева

```
var Node : TTreeNode;
begin
  Node:=TreeView1.Items.GetFirstNode;//самый первый узел в дереве
  while Node<>nil do
  begin
    //операции с узлом Node
    Node:=Node.GetNext;
  end;
end;
```

Познакомимся еще с одним примером, на этот раз позволяющим собрать сведения только о прямых потомках узла `ParentNode` (листинг 23.3).

Листинг 23.3. Просмотр прямых потомков узла

```
procedure TForm1.GetChildsList(ParentNode: TTreeNode);
var Node : TTreeNode;
begin
  Node:=ParentNode.GetFirstChild;
  while Node<>nil do
  begin
    //операции с узлом Node
    Node:=Node.GetNextChild(Node);
  end;
end;
```

При переборе узлов в дереве стоит учитывать вероятность того, что узел дерева может оказаться невидимым (например, когда родительский узел свернут). Узнать о видимости узла можно благодаря свойству

```
property IsVisible: Boolean;
```

Если логика вашей программы требует перебрать только видимые узлы, то следует воспользоваться услугами методов

```
function GetNextVisible: TTreeNode;  //следующий видимый узел
function GetPrevVisible : TTreeNode; //предыдущий видимый узел
```

Перемещение узла

Иерархическая структура элементов динамична. Каждый узел имеет право выбирать свое местоположение. Например, для смены своего владельца узел `TTreeNode` использует метод

```
procedure MoveTo(Destination: TTreeNode; Mode: TNodeAttachMode);
type TNodeAttachMode =
     (naAdd, naAddFirst, naAddChild, naAddChildFirst, naInsert);
```

Здесь `Destination` — новый узел-получатель. Хотя, точнее его назвать не получателем, а ориентиром, относительно которого осуществляется перемещение узла. Дело в том, что элемент `Destination` может и не стать владельцем перемещаемого узла. Все зависит от значения, передаваемого программистом в параметр `Mode` (табл. 23.3).

Таблица 23.3. *Режим перемещения узла* `TNodeAttachMode`

Значение	Описание
`naAdd`	Перемещаемый узел окажется последним на одном и том же уровне, что и узел `Destination`
`naAddFirst`	Перемещаемый узел окажется первым на уровне узла `Destination`
`naInsert`	Перемещаемый узел установится на месте узла `Destination`, сместив его вниз на том же уровне
`naAddChild`	Перемещаемый узел вставится как последний дочерний
`naAddChildFirst`	Перемещаемый узел окажется первым дочерним

Удаление узла

В составе методов узла предусмотрены две процедуры, отвечающие за уничтожение экземпляра класса `TTreeNode`. Физическое удаление узла и всех подчиненных ему узлов произойдет при обращении к методу

```
procedure Delete;
```

Еще один метод нацелен на удаление только дочерних узлов:

```
procedure DeleteChildren;
```

ЗАМЕЧАНИЕ

Ответственность за создание узла и вставку его в дерево возложена на хранилище узлов `TTreeNodes`.

Значок узла

Каждый узел умеет отображать определенный набор картинок. Для этого необходимо подключить к дереву `TTreeView` контейнер изображений `TImageList` (свойства `Images`), а если требуется отображать и значки состояния узла, то еще один контейнер присоединяется к свойству `StateImages`. После связывания дерева и контейнера изображений каждый узел `Node` приобретает право обладать личной пиктограммой, для этого достаточно только установить индекс (табл. 23.4).

Таблица 23.4. *Индексы значков узла* `TTreeNode`

Свойства	Описание
`property ImageIndex: Integer;`	Индекс значка невыбранного узла в нормальном состоянии
`property SelectedIndex: Integer;`	Индекс значка выбранного узла
`property OverlayIndex: Integer;`	Индекс картинки, накладываемой поверх основной
`property StateIndex: Integer;`	Дополнительная картинка состояния из контейнера, подключенного к свойству `StateImages`

ЗАМЕЧАНИЕ

Дополнительные рычаги по управлению значками можно получить, обратившись к событиям `OnGetImageIndex()` и `OnGetSelectedIndex()`.

Свертывание и развертывание узла

Имеющий дочерние элементы узел в состоянии скрывать (сворачивать) и показывать (разворачивать) подчиненные ему элементы. Эти операции инициируются пользователем двойным щелчком кнопкой мыши по узлу или одинарным щелчком по значку слева от узла. Для программного управления процессом сворачивания/разворачивания необходима помощь свойства

```
property Expanded: Boolean;
```

Еще большего результата можно добиться, обратившись к методам:

```
procedure Collapse(Recurse: Boolean);
procedure Expand(Recurse: Boolean);
```

Передав в параметр `Recurse` значение `true`, вы укажете узлу дерева, что следует развернуть (свернуть) всю плеяду подчиненных узлов.

Непосредственно в компоненте `TTreeView` имеются методы, позволяющие развернуть и свернуть все узлы дерева:

```
procedure FullExpand;   //развернуть все
procedure FullCollapse; //свернуть все
```

При остром желании можно заставить дерево самостоятельно разворачивать свои узлы при простом выделении узла указателем мыши или с помощью клавиатуры. Для этого следует передать значение `true` в свойство

```
property AutoExpand: Boolean; //по умолчанию False
```

Процесс свертывания и развертывания узлов в компоненте `TTreeView` сопровождают две пары событий. Перед началом сворачивания узла `Node` генерируется событие

```
property OnCollapsing : TTVCollapsingEvent;
type TTVCollapsingEvent = procedure(Sender: TObject; Node: TTreeNode;
                                    var AllowCollapse: Boolean) of object;
```

Благодаря параметру `AllowCollapse` программист имеет право остановить процесс свертывания (`false`). Если же в параметр передано значение `true`, то узел сворачивается, и настает черед события

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

Вторая пара событий обслуживает процесс разворачивания узла. Перед началом разворачивания вызывается событие

```
property OnExpanding: TTVExpandingEvent;
type TTVExpandingEvent = procedure(Sender: TObject; Node: TTreeNode;
                                   var AllowExpansion: Boolean) of object;
```

Здесь параметр `AllowExpansion` также предназначен для запрета (`false`) или разрешения (`true`) разворачивания. Очередное событие

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

вызывается сразу после разворачивания узла. Во всех четырех событиях параметр `Node` ссылается на узел, инициировавший этот процесс.

В листинге 23.4 представлен пример изменения картинки узла "на лету" в момент его разворачивания (событие `OnExpanded`) и сворачивания (событие `OnCollapsed`). Допустим, что с помощью свойства `Images` к дереву `TreeView1` подключена коллекция картинок `ImageList1`, в этой коллекции имеются рисунки с изображением закрытой папки (нулевой индекс) и распахнутой папки (первый индекс).

Листинг 23.4. Смена значка узла

```
procedure TForm1.TreeView1Expanded(Sender: TObject; Node: TTreeNode);
begin
  Node.ImageIndex:=1;
  Node.SelectedIndex:=1;
end;

procedure TForm1.TreeView1Collapsed(Sender: TObject; Node: TTreeNode);
begin
  Node.ImageIndex:=0;
  Node.SelectedIndex:=0;
end;
```

Во время разворачивания и сворачивания узла мы меняем индексы значков узла `Node`, предоставляя пользователю дополнительную информацию о состоянии узла (рис. 23.3).

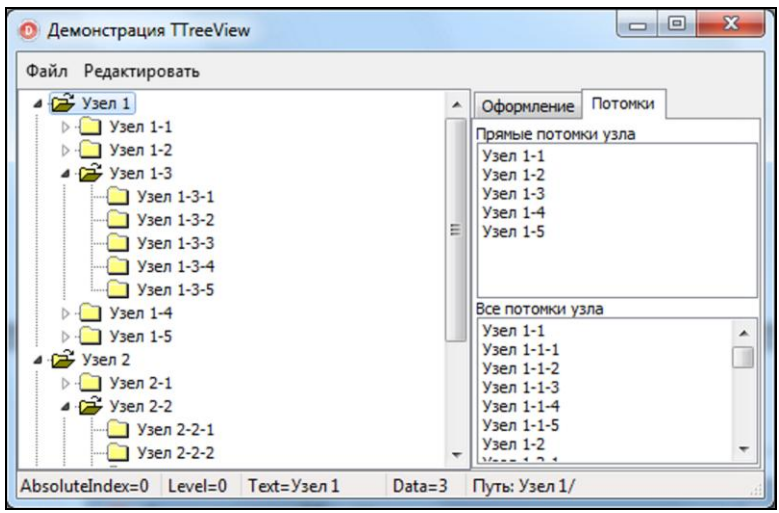


Рис. 23.3. Применение событий `OnExpanded()` и `OnCollapsed()` для смены значка узла

Хранилище узлов класс *TTreeNodes*

Инкапсулированный в состав элемента управления `TTreeView` класс `TTreeNodes` не афиширует принятый в нем способ хранения узлов в памяти компьютера. Внешней стороной хранилища узлов выступает удобный набор свойств и методов, позволяющих управлять узлами

дерева. Класс TTreeNodes несет прямую ответственность за вставку, редактирование и удаление узлов дерева, упорядочивание узлов по алфавиту (или другому критерию), предоставление доступа к отдельному узлу.

Воспользовавшийся услугами компонента TTreeView программист может рассматривать узлы дерева как элементы массива, доступ к отдельному элементу массива осуществляется традиционно — по его индексу с помощью свойства

```
property Item[Index: Integer]: TTreeNode;
```

Максимальный индекс в массиве зависит от количества узлов, занесенных в дерево

```
property Count: Integer;
```

Предусмотрен быстрый способ доступа к самому первому узлу коллекции (визуально это самый верхний узел дерева), его обеспечивает функция

```
function GetFirstNode: TTreeNode;
```

По своей сути это аналог обращения к свойству Item с нулевым индексом.

Ссылка на владельца всего набора узлов (элемент управления TTreeView) доступна благодаря свойству

```
property Owner: TCustomTreeView; //только для чтения
```

Еще одна ссылка на родительский компонент находится в свойстве

```
property Handle: HWND;
```

Это оконный дескриптор, позволяющий управлять элементом управления при посредничестве функций Windows API.

Добавление узлов

Предусмотрен большой выбор методов, позволяющих пополнять хранилище новыми узлами. Эти методы можно разделить на две категории: методы, осуществляющие простое добавление нового узла, и методы, добавляющие вместе с узлом ссылку на внешний объект с произвольными данными (табл. 23.5). Во всех методах имеется повторяющийся перечень аргументов: параметр Node определяет опорный узел дерева, относительно которого осуществляются операции добавления (вставки) узла; параметр S содержит название узла; параметр Ptr представляет собой указатель на необязательные внешние данные. Все методы возвращают ссылку на созданный узел TTreeNode.

Таблица 23.5. *Методы класса TTreeNodes, предназначенные для создания нового узла и добавления его в дерево*

Метод	Описание
Простые методы создания нового узла и добавления его в дерево	
`function Add(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется самым последним на уровне узла Node
`function AddFirst(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется первым на уровне узла Node
`function AddChild(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется в качестве последнего дочернего узла относительно Node

***Таблица 23.5** (окончание)*

Метод	Описание
`function AddChildFirst(Node: TTreeNode; const S: string): TTreeNode;`	Новый узел добавляется в качестве первого дочернего узла относительно `Node`. Остальные дочерние узлы сдвигаются вниз
`function Insert(Node: TTreeNode; const S: string): TTreeNode;`	Новый узел вставляется перед `Node`
Создание и узла и связанного с ним объекта	
`function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Новый узел добавляется самым последним на уровне узла `Node`
`function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer) : TTreeNode;`	Узел добавляется первым на уровне узла `Node`
`function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer) : TTreeNode;`	Узел добавляется к `Node` в качестве последнего дочернего узла
`function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Узел добавляется в качестве первого дочернего узла относительно `Node`, остальные дочерние узлы сдвигаются вниз
Добавление к дереву уже существующего узла	
`function InsertNode(Node, Sibling: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Вставляет узел `Sibling` перед узлом `Node` на том же уровне
`function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Вставка нового узла перед `Node`

Порядок построения произвольного дерева демонстрирует листинг 23.5. В цикле мы создаем три узла верхнего уровня. Появившиеся на свет узлы никому не подчинены — об этом свидетельствует неопределенный указатель `nil` в первом параметре метода `Add()`. После создания каждого узла верхнего уровня вызывается вложенный цикл, в рамках которого появляются еще 5 дочерних узлов.

Листинг 23.5. Создание группы узлов

```
var x,y:integer;
    ParentNode:TTreeNode;
begin
  with TreeView1.Items do
  begin
    BeginUpdate;
    for x:=1 to 3 do
    begin
      ParentNode:=Add(nil,'Узел '+IntToStr(x));
      for y:=1 to 5 do
        AddChild(ParentNode,'Узел '+IntToStr(x)+'-'+IntToStr(y));
    end;
    EndUpdate;
  end;
end;
```

ЗАМЕЧАНИЕ

Если требуется добавить узел в самый верх иерархии, то в качестве параметра `Node` передавайте неопределенный указатель `nil`.

Обратите внимание на задействованные в листинге 23.1 методы

```
procedure BeginUpdate;
procedure EndUpdate;
```

Они призваны ускорить процесс вывода информации на экран в случаях, когда добавляется, перемещается или удаляется значительное количество узлов. Метод `BeginUpdate()` уведомляет систему о том, что сейчас произойдет изменение состава узлов, и прекращает процесс перерисовки дерева до тех пор, пока не будет вызван метод `EndUpdate()`.

Листинг 23.6 представляет фрагмент более сложного примера, он демонстрирует процесс добавления к дереву узла с дополнительными данными.

Листинг 23.6. Создание узла с дополнительными данными

```
type pAddInfo=^TAddInfo;
     TAddInfo=Record   //структура данных
      Text:string[20]; //текстовое поле
      Value:integer;   //числовое поле
end;
...
var AI:pAddInfo;
begin
  New(AI); //создаем в памяти структуру и заполняем поля
  AI^.Text:='Дополнительный текст';
  AI^.Value:=100;
  //одновременно с созданием узла передаем указатель на структуру
  TreeView1.Items.AddObject(nil,'Узел '+IntToStr(x),AI);
...
```

Ассоциируемые с узлом внешние данные могут характеризоваться любой степенью сложности (от простых чисел до структурированных объектов), в нашем примере в качестве внешних данных использовалась запись `TAddInfo`.

ЗАМЕЧАНИЕ

Доступ к связанным с узлом `TTreeNode` внешним данным предоставляет его свойство `Data`.

Процесс создания нового узла сопровождается событием

```
property OnCreateNodeClass: TTVCreateNodeClassEvent;
type TTVCreateNodeClassEvent = procedure(Sender: TCustomTreeView;
                                 var NodeClass: TTreeNodeClass) of object;
```

Событие может понадобиться в том случае, когда программист намерен создать собственную версию узла, для этого достаточно подменить параметр `NodeClass` своей версией узла.

При вставке нового узла вызывается обработчик события

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

Здесь `Node` — ссылка на только что добавленный узел.

Для того чтобы научиться решать обратную задачу получения данных по указателю, достаточно изучить листинг 23.7.

Листинг 23.7. Чтение данных из свойства Data узла TTreeNode

```
var AI: TAddInfo;
...
if TreeView1.Selected.Data<>nil then AI:= TAddInfo(Node.Data^);
```

Убедившись, что указатель выделенного узла дерева не пуст, мы передаем данные из памяти в переменную AI.

Сортировка узлов

Рассматривая задачу упорядочивания узлов, в первую очередь заметим, что компонент TTreeView способен автоматически сортировать принадлежащие ему узлы. Для этого следует отказаться от значения по умолчанию в свойстве

```
property SortType: TSortType; //по умолчанию stNone
```

и установить подходящий режим упорядочивания (stData, stText или stBoth).

> ***ЗАМЕЧАНИЕ***
>
> Сам факт изменения состояния свойства SortType на любое значение, отличное от stNone, генерирует событие сравнения узлов дерева OnCompare().

Программист вправе усовершенствовать процесс сортировки. Для этого предназначено событие

```
property OnCompare: TTVCompareEvent;
type TTVCompareEvent = procedure(Sender: TObject;
  Node1, Node2: TTreeNode; Data: Integer; var Compare: Integer) of object;
```

Процедура вызывается при каждом сравнении двух узлов дерева: Node1 и Node2. Параметр Data может содержать дополнительные данные, передаваемые из внешнего метода компонента. Результат сравнения узлов окажется в последнем параметре события — значение меньше 0 указывает на то, что узел lParam1 должен следовать перед узлом lParam2, нулевое значение говорит, что узлы равны, положительный результат указывает на то, что узел lParam1 должен находиться после узла lParam2.

Допустим, что в свойствах Data узлов дерева находятся указатели на какие-то целые числа. Тогда для упорядочивания узлов по этим данным следует перевести свойство SortType в состояние stData и описать реакцию на событие OnCompare() (листинг 23.8).

Листинг 23.8. Сортировка узлов дерева по значениям в Data

```
procedure TForm1.TreeView1Compare(Sender: TObject;
        Node1, Node2: TTreeNode; Data: Integer; var Compare: Integer);
begin
  if Integer(Node1.Data^)>Integer(Node2.Data^) then Compare:=1
  if Integer(Node1.Data^)=Integer(Node2.Data^) then Compare:=0
  else Compare:=-1;
end;
```

Процесс сортировки узлов дерева способно инициировать и хранилище узлов TTreeNodes. В простейшем случае для сортировки узлов дерева по их текстовой надписи следует вызвать метод

```
function AlphaSort(ARecurse: Boolean = False): Boolean;
```

Параметр ARecurse предназначен для уточнения направления сортировки: false — по возрастанию, true — по убыванию значений. Отметим, что метод AlphaSort() является надстройкой над более сложной функцией упорядочивания узлов в дереве:

```
function CustomSort(SortProc: TTVCompare; Data: Longint;
                    ARecurse: Boolean = False): Boolean;
```

На этот раз сортировка производится не по отражаемому в узле тексту, а по связанным с узлом данным. Параметр ARecurse определяет направление сортировки, параметр Data при необходимости позволяет передать данные во внешнюю функцию (функцию обратного вызова), на которую ссылается самый первый параметр метода, и в обработчик события OnCompare(), который мы упомянули несколькими строками ранее.

Удаление узлов из коллекции

Для удаления отдельного узла Node предназначен метод

```
procedure Delete(Node: TTreeNode);
```

Для полного стирания всех элементов иерархической структуры вызывается процедура

```
procedure Clear;
```

Не забывайте, что если мы сопоставили с каждым узлом дополнительные данные, то нашей прямой обязанностью будет и освобождение связанных объектов. Например, перед вызовом метода, очищающего дерево, необходимо освободить память, зарезервированную для хранения дополнительных объектов (листинг 23.9).

Листинг 23.9. Удаление данных из узлов перед полной очисткой дерева

```
var i : integer;
begin
  for i:=0 To TreeView1.Items.Count-1 do
    if TreeView1.Items.Item[i].Data<>Nil then
       Dispose(TreeView1.Items.Item[i].Data);
  TreeView1.Items.Clear;
end;
```

> ***ЗАМЕЧАНИЕ***
>
> Узел TTreeNode также обладает способностью самоуничтожения (метод Delete) и возможностью удалить все дочерние по отношению к нему элементы (метод DeleteChildren).

Процесс удаления узла сопровождается вызовом события

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

Здесь Node — ссылка на доживающий последние мгновения элемент иерархии.

Редактирование текста узла

Если компонент TTreeView доступен для редактирования (ReadOnly=false), то пользователь получает возможность изменить текст узла. Для этого достаточно дважды щелкнуть левой кнопкой мыши по области текста.

Узел, находящийся в режиме редактирования, проинформирует об этом программиста значением true в свойстве

```
function IsEditing: Boolean;
```

Дерево TTreeView умеет отслеживать все попытки пользователя изменить текст узла Node. В начале редактирования возникает событие

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

На данном этапе можно запретить процесс редактирования, а для этого достаточно установить в false значение AllowEdit. После того как пользователь закончил вводить текст, генерируется заключительное событие

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

Так как только что введенный пользователем текст пока находится в параметре S, мы можем использовать событие OnEdited() для возврата текста в исходное состояние (листинг 23.10).

Листинг 23.10. Редактируем текст узла

```
procedure TForm1.TreeView1Edited(Sender: TObject; Node: TTreeNode;
                                 var S: string);
var msg:string;
begin
  msg:=Format('Изменить %s в %s?', [Node.Text,S]);
  if MessageBox(TreeView1.Handle, pWideChar(msg),
       'Редактируем текст', MB_YESNO)=idNo then  S:= Node.Text;
end;
```

Оформление дерева

По умолчанию компонент TTreeView имеет достаточно скромный вид. Если активизировано свойство

```
property ShowButtons: Boolean; //по умолчанию true
```

то слева от названия узла отображается кнопка со значком в виде треугольника, наклон треугольника изменяется в зависимости от того, развернут или свернут узел.

Дочерние и родительские узлы соединены линиями. За показ соединительных линий между родительскими и дочерними узлами отвечает свойство

```
property ShowLines: Boolean; //по умолчанию true
```

Внешний вид корневых узлов определяется свойством

```
property ShowRoot: Boolean; //по умолчанию true
```

Каждый узел умеет отображать определенный набор картинок. Для этого необходимо подключить к дереву один (или два) контейнер изображений `TImageList`. Контейнер присоединяется благодаря свойствам

```
property Images: TCustomImageList;
property StateImages: TCustomImageList;
```

Свойство `Images` — стыковочный шлюз для основного контейнера `TImageList`, хранящего базовый набор значков. Свойство `StateImages` используется реже, оно предназначено для присоединения контейнера, содержащего значки состояния — дополнительные рисунки, выводимые слева от названия узла.

Со сменой картинок, отображаемых в узлах дерева, связаны два события:

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

Первое из них предназначено для смены основной картинки узла, второе используется для смены дополнительного значка у узла перед его прорисовкой на экране.

Все приложения Windows и элементы управления этих приложений периодически нуждаются в перерисовке своей поверхности. Обновление поверхности начинается с момента получения приложением соответствующего сообщения от операционной системы. Кроме того, перерисовка может быть инициирована из кода самого приложения. Процесс графического вывода элемента управления `TTreeView` разделяется на четыре этапа, описываемые типом данных `TCustomDrawStage`.

```
type TCustomDrawStage = (cdPrePaint,  //подготовка к графическому выводу
                         cdPostPaint, //после графического вывода
                         cdPreErase,  //перед началом стирания
                         cdPostErase);//после стирания
```

Для реакции на основные этапы цикла перерисовки в компоненте `TTreeView` предусмотрены два усовершенствованных (advance) обработчика события. За вывод фоновой части дерева отвечает событие

```
property OnAdvancedCustomDraw: TTVAdvancedCustomDrawEvent;
type TTVAdvancedCustomDrawEvent = procedure(Sender: TCustomTreeView;
                         const ARect: TRect; Stage: TCustomDrawStage;
                         var DefaultDraw: Boolean) of object;
```

Здесь: `Sender` — ссылка на компонент `TTreeView`; `ARect` — прямоугольные координаты перерисовываемой области; `Stage` информирует о текущем этапе графического вывода; параметр-переменная `DefaultDraw` на этапе `cdPrePaint` позволяет включать/отключать режим вывода по умолчанию.

Прорисовка узлов дерева проводится в рамках события

```
property OnAdvancedCustomDrawItem: TTVCustomDrawItemEvent;
type   TTVAdvancedCustomDrawItemEvent =
        procedure(Sender: TCustomTreeView; Node: TTreeNode;
                  State: TCustomDrawState; Stage: TCustomDrawStage;
                  var PaintImages, DefaultDraw: Boolean) of object;
```

В дополнение к уже известным нам по предыдущему событию параметрам здесь появились аргументы: `Node` — ссылка на выводимый на экран узел дерева; `State` — состояние узла (`cdsSelected` — выделен, `cdsFocused` — в фокусе и т. д.); `PaintImages` — на стадии прорисовки `Stage=cdPrePaint` разрешает/запрещает вывод связанного с узлом рисунка.

Листинг 23.11 демонстрирует вариант применения события `OnAdvancedCustomDrawItem()` с целью вывода рядом с заголовком узла связанных с узлом дополнительных данных. В данном примере предполагается, что с отдельным элементом дерева может быть связано какое-то целочисленное значение типа `integer`.

Листинг 23.11. Усовершенствованный вывод узла с дополнительными данными

```
procedure TForm1.TreeView1AdvancedCustomDrawItem(Sender: TCustomTreeView;
  Node: TTreeNode; State: TCustomDrawState; Stage: TCustomDrawStage;
  var PaintImages, DefaultDraw: Boolean);
var i:integer;
    Rect:TRect;
    s:string;
begin
  if  (Stage=cdPostPaint) and (Node.Data<>nil) then
    with TreeView1.Canvas do
    begin
      Font.Name:='Arial';
      i:=INTEGER(Node.Data^);
      Rect:=Node.DisplayRect(true);
      Font.Color:=clGrayText;
      s:='(доп.значение: '+IntToStr(i)+')';
      TextOut(Rect.Right+5,Rect.Top+1,s);
      Font.Color:=clWindowText;
    end;
end;
```

Услуги события нам потребовались на заключительном этапе графического вывода `cdPostPaint`. Мы считываем целочисленное значение (на которое ссылается указатель `Data` узла `Node`) и выводим его правее основного текста узла. Обратите внимание на то, что для вычисления прямоугольной области, занимаемой узлом, мы воспользовались методом

```
function DisplayRect(TextOnly: Boolean): TRect;
```

Единственный параметр метода уточняет, включать в область вывода только текст узла или все пространство элемента дерева. Результаты выполнения листинга 23.11 представлены на рис. 23.4.

> ***Замечание***
>
> В ранних версиях Delphi управление графическим выводом дерева обычно возлагалось на события `OnCustomDraw()` и `OnCustomDrawItem()`. Первое событие вызывается непосредственно перед началом операций прорисовки узлов дерева, как правило, оно использовалось для фона элемента управления. Второе событие генерируется перед началом прорисовки отдельного узла. Сегодня эти события остались для поддержки обратной совместимости с программами старого парка, и вместо них следует использовать более совершенные события `OnAdvancedCustomDraw()` и `OnAdvancedCustomDrawItem()`.

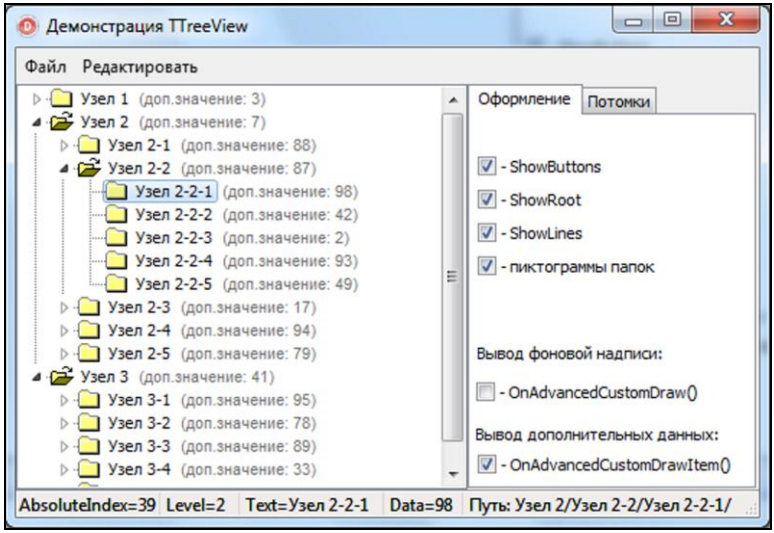


Рис. 23.4. Вывод дополнительных данных с помощью события `OnAdvancedCustomDrawItem()`

ГЛАВА 24

Панели-контейнеры

Визуальная библиотека компонентов Delphi предоставляет в распоряжение программиста широкий перечень компонентов, исполняющих роль контейнеров для других визуальных элементов управления. В первую очередь это многочисленные панели и области с возможностью прокрутки (скроллинга).

Общая черта изучаемых в этой главе элементов управления в том, что они становятся родительскими по отношению к расположенным на их поверхности компонентам. Другими словами, перенос на панель (область скроллинга, планку) любого другого визуального элемента управления приводит к тому, что в свойстве `Parent` дочернего компонента появляется ссылка на панель. В свою очередь панель приобретает ряд прав по управлению дочерними компонентами, прежде всего связанными с вопросами размещения и отображения.

Простые панели

Все панели являются полноценными оконными элементами управления. Благодаря этому они обладают дескриптором окна, способны реагировать на фокус ввода и, самое главное, могут выступать в роли родительского контейнера для других элементов управления VCL. Ко всему прочему, при проектировании пользовательского интерфейса панели позволят нам получить ряд полезных преимуществ, в их числе:

- *обеспечение логической группировки компонентов.* Размещенные на поверхности панелей элементы управления компонуются в согласованно работающую логическую группу. В данном случае наиболее показательны примеры с кнопкой выбора `TRadioButton` и быстрой кнопкой `TSpeedButton`. Перенеся несколько кнопок `TRadioButton` на панель, вы автоматически создадите логическое объединение кнопок, по своему поведению сильно напоминающее компонент `TRadioGroup`. Очень похожий результат вы получите, расположив на панели несколько быстрых кнопок с одинаковым значением индекса `GroupIndex`;
- *общий стиль визуального оформления компонентов.* Панели существенно повышают наглядность пользовательского интерфейса приложения за счет формирования зрительного образа единой группы элементов управления, эффект можно существенно усилить благодаря применению разнообразных дизайнерских приемов (управлением объемом, начертанием границ, текстовыми подписями);
- *упрощение процесса размещения и выравнивания элементов управления* при изменении размеров рабочих форм приложения. Благодаря наличию свойств `Align` и `Anchors` и об-

работчиков событий `ConstrainedResize()` и `OnResize()` мы сможем одновременно управлять поведением всех компонентов, принадлежащих панелям.

На рис. 24.1 отражен фрагмент иерархии VCL с компонентами-панелями. Среди представленных на схеме компонентов наиболее просты в эксплуатации обычная панель `TPanel` и область группировки `TGroupBox`. Эти элементы управления просто играют роль контейнеров для дочерних компонентов. Панели `TFlowPanel` и `TGridPanel` представляют собой развитие традиционной панели и предоставляют разработчику дополнительный сервис по размещению на панели дочерних элементов управления. В частности, панель-сетка `TGridPanel` и панель-поток `TFlowPanel` очень удобны в приложениях, в которых пользовательский интерфейс формируется динамически (во время выполнения программы).

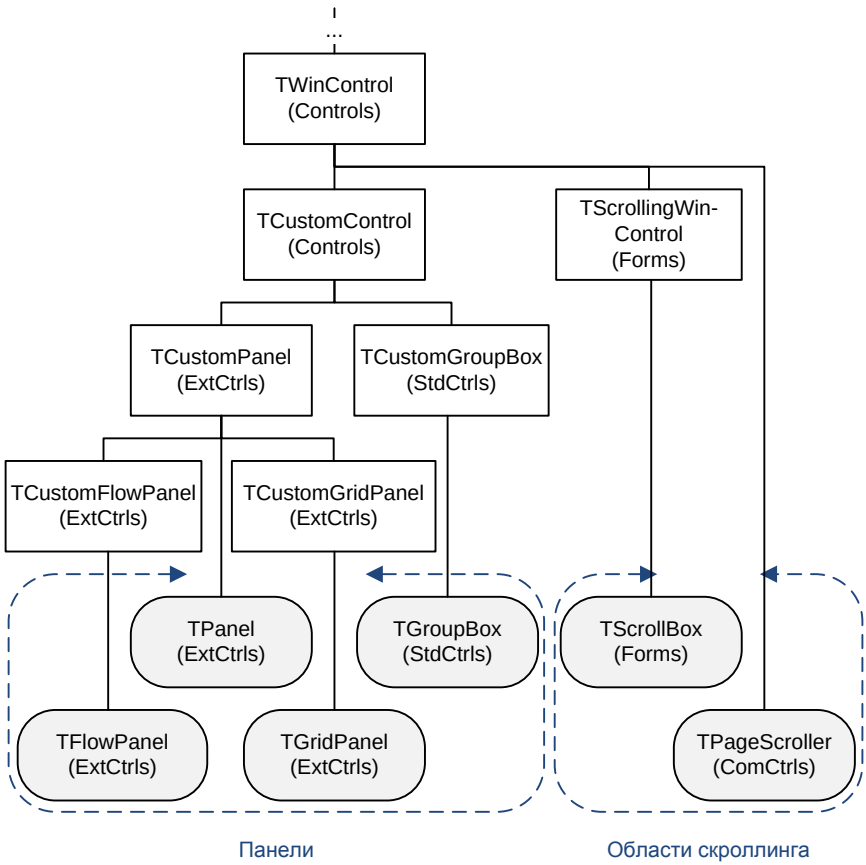


Рис. 24.1. Фрагмент иерархии VCL с панелями

Простая панель *TPanel*

Вряд ли существует особая необходимость в отдельном представлении панели `TPanel`. Это один из наиболее простых и одновременно один из самых популярных элементов управления, активно используемых программистами еще с первых версий Delphi.

Для того чтобы повысить эстетические качества панели, создатели компонента вооружили панель рядом свойств, влияющих на ее внешний вид. В первую очередь это свойства, по-

зволяющие настроить эффект объемного вида при графическом выводе фасок. Стиль начертания фасок панели определяет свойство

```
property BevelKind: TBevelKind; //по умолчанию bkNone
type TBevelKind = (bkNone, bkTile, bkSoft, bkFlat);
```

Эффект выпуклой, вдавленной или плоской панели обеспечат свойства

```
property BevelInner: TPanelBevel; //по умолчанию bvNone
property BevelOuter: TPanelBevel; //по умолчанию bvRaised
```

Толщину фаски устанавливает свойство

```
property BevelWidth: TBevelWidth; //по умолчанию 1
```

Кроме того, панель позволяет управлять начертанием ее границ

```
property BorderStyle: TBorderStyle;//по умолчанию bsNone
```

По умолчанию прорисовка границы панели отключена, для ее включения следует направить в свойство значение `bsSingle`. Ширина границы определяется свойством

```
property BorderWidth: TBorderWidth;
```

Комбинируя рассмотренные свойства, можно создавать весьма симпатичные образцы панелей, наиболее показательные из которых представлены на рис. 24.2.

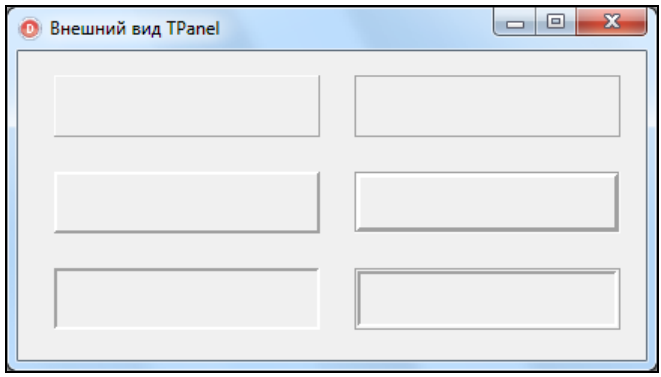


Рис. 24.2. Варианты отображения `TPanel`

Процесс прорисовки фасок панели связан с унаследованным от класса `TWinControl` свойством

```
property BevelEdges: TBevelEdges;// [beLeft, beTop, beRight, beBottom]
```

По умолчанию включены все элементы множества `TBevelEdges`, что указывает на то, что графическому выводу подлежат все четыре грани панели. Ситуацию можно изменить, отключив ненужные фаски.

Панель способна выводить текстовую надпись. Для этого предназначено свойство `Caption`. Однако оно практически никогда не применяется, т. к. строку текста обычно закрывают расположенные на панели элементы управления. Для отключения отображения текста проще всего воспользоваться свойством

```
property ShowCaption: Boolean;
```

переведя его в состояние `false`.

Если без заголовка панели обойтись нельзя, то можно попытаться "вытащить" текст из-под скрывающих его элементов управления, сместив надпись по горизонтали и вертикали с помощью свойств

```
property Alignment: TAlignment;
property VerticalAlignment: TVerticalAlignment;
```

Панель *TFlowPanel*

Панель `TFlowPanel` призвана упростить размещение и упорядочивание на своей поверхности однотипных элементов управления. Приставка *flow* (поток, струя) подсказывает о наличии у элемента управления одной уникальной черты. При переносе на панель `TFlowPanel` других элементов управления они располагаются не там, где их оставит программист (именно так ведут себя компоненты на традиционной панели `TPanel`), а упорядоченно заполняют панель, размещаясь один за другим так, словно они объединены в поток.

> ***Замечание***
>
> При изменении своих размеров панель `TFlowPanel` перестраивает принадлежащие ей элементы, не нарушая их последовательности.

Порядок размещения элементов управления определяется свойством

```
property FlowStyle: TFlowStyle; //по умолчанию fsLeftRightTopBottom
```

По умолчанию принят классический стиль потока компонентов слева направо, сверху вниз. Однако свойство дает простор для экспериментов, предоставляя еще семь вариантов упорядочивания дочерних элементов управления.

```
type TFlowStyle = (fsLeftRightTopBottom, fsRightLeftTopBottom,
       fsLeftRightBottomTop, fsRightLeftBottomTop, fsTopBottomLeftRight,
       fsBottomTopLeftRight, fsTopBottomRightLeft, fsBottomTopRightLeft);
```

Достойно упоминания еще одно свойство панели:

```
property AutoWrap: Boolean; //по умолчанию True
```

Переведя его в состояние `false`, вы нарушите работу панели, запретив многострочное размещение дочерних элементов управления.

Панель-сетка *TGridPanel*

Самое главное отличие между существующей в составе библиотеки компонентов Delphi еще со времен первой версии панели `TPanel` и сравнительно новой панелью-сеткой `TGridPanel` заключается в их подходе к размещению на своей поверхности других элементов управления. Если традиционной панели по большому счету было безразлично, в каком именно месте ее клиентской области окажется очередной элемент управления, то у панели-сетки на этот счет имеются свои соображения. Панель `TGridPanel` самостоятельно примет решение о расположении элемента управления или (если пожелания программиста и возможности панели совпадают) выровняет компонент по фактическому месту.

Наличием в своем названии слова *grid* (сетка) компонент обязан двум свойствам:

```
property ColumnCollection: TColumnCollection;
property RowCollection: TRowCollection;
```

Свойства предоставляют доступ к коллекциям колонок и рядов, разделяющих пространство клиентской области панели на секторы. Создавая очередную колонку (ряд), программист должен определить, какую часть пространства панели предстоит занимать колонке (ряду). При этом допускается указывать абсолютные величины (допустим, 100 пикселов), процентное соотношение (например, 50% от общего пространства) или установить автоматический режим подбора размеров. По умолчанию панель-сетка разделена на 4 сектора, каждый из которых занимает ровно четверть пространства ее поверхности. При переносе визуального компонента (например, кнопки) с палитры Delphi на эту панель он окажется ровно в центре "северо-западного" сектора (рис. 24.3).

Замечание

И коллекция колонок, и коллекция рядов панели `TGridPanel` построены на основе одного и того же предка — коллекции ячеек класса `TCellCollection`.

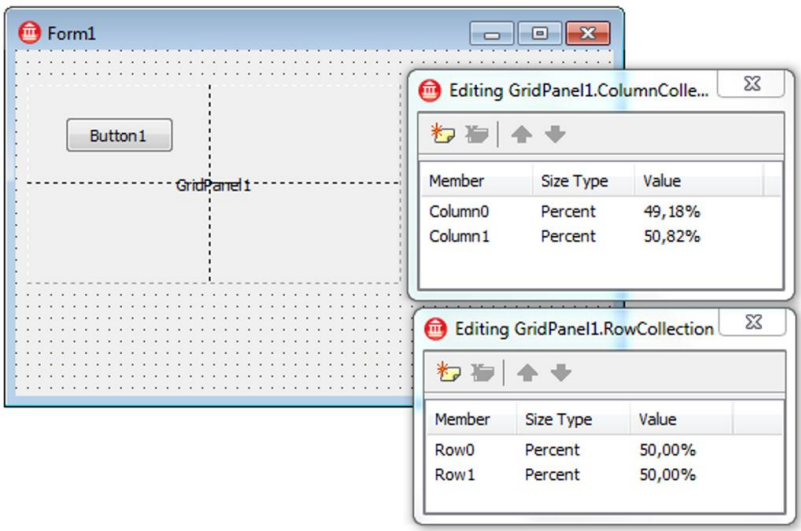


Рис. 24.3. Панель-сетка с активными редакторами колонок и рядов и компонентом `Button1`

Замечание

Для удобства программиста во время визуального проектирования колонки и ряды панели разделяются пунктирной линией, после старта приложения пунктирная линия исчезает.

Наличие четырех видимых секторов позволяет нам разместить на поверхности панели 4 элемента управления. Что будет происходить, если нам потребуется перенести на сетку пятый по счету компонент? Дальнейшее развитие событий в первую очередь определяется состоянием свойства

```
property ExpandStyle: TExpandStyle; //по умолчанию emAddRows
```

Свойство определяет поведение сетки в ситуации, когда количество компонентов, претендующих на попадание на поверхность панели-сетки, превышает число свободных секторов. По умолчанию включен режим автоматического добавления нового ряда `emAddRows`, но предусмотрены еще два альтернативных варианта поведения.

```
type TExpandStyle = (emAddRows,    //автоматическое добавление ряда
                     emAddColumns, //автоматическое добавление колонки
                     emFixedSize); //неизменяемый размер
```

Панель способна управлять принадлежащими ей элементами (и претендентами на принадлежность к панели) с помощью свойства

```
property ControlCollection: TControlCollection;
```

Коллекция обладает всем набором возможностей по предоставлению доступа к своим элементам (свойство `ControlItems`) и добавлением/удалением элемента (методы `AddControl` и `RemoveControl`). При добавлении элемента в коллекцию допустимо указать, в какой именно сектор панели-сетки следует поместить компонент (листинг 24.1).

Листинг 24.1. Метод `AddControl()` определяет сектор размещения элемента

```
var aRow, aCol:Integer;
    btn : TSpeedButton;
begin
  for aRow:=0 to GridPanel1.RowCollection.Count-1 do
    for aCol:=0 to GridPanel1.ColumnCollection.Count-1 do
    begin
      btn:=TSpeedButton.Create(GridPanel1);
      btn.GroupIndex:=aRow+1; //индекс группировки
      btn.Caption:=IntToStr(aRow)+'-'+IntToStr(aCol);
      GridPanel1.ControlCollection.AddControl(btn,aRow,aCol);
      btn.Parent:=GridPanel1;
    end;
end;
```

В представленном в листинге 24.1 примере мы заполняем все секторы панели-сетки экземплярами кнопок `TSpeedButton`, одновременно группируя кнопки по колонкам.

При необходимости можно узнать размеры любого сектора и прямоугольные координаты сектора, для этого достаточно передать индексы колонки и ячейки в соответствующие свойства:

```
property CellSize[AColumn, ARow: Integer]: TPoint;//размер сектора
property CellRect[AColumn, ARow: Integer]: TRect; //координаты ячейки
```

Если вам тяжело перемножить число колонок на число рядов, чтобы выяснить количество ячеек на панели, за вас это сделает свойство

```
property CellCount: Integer;//только для чтения
```

Во всем остальном панель-сетка практически ничем не отличается от традиционной панели `TPanel`.

Область группировки *TGroupBox*

Элемент управления `TGroupBox` создавался специально для обеспечения логической группировки однотипных элементов управления. В отличие от панелей область группировки не обладает возможностью настраивать особенности графического вывода своих границ и фасок, но этот недостаток отчасти компенсируется наличием встроенной рамки с надписью в верхней ее части. Во всем остальном порядок работы с `TGroupBox` ничем не отличается от работы с классической панелью `TPanel`.

Контейнеры с возможностью скроллинга

При проектировании пользовательского интерфейса разработчик программного обеспечения часто сталкивается с проблемой нехватки свободного пространства на поверхности формы. Зачастую бывает просто невозможно разместить в клиентской области окна чертеж, рисунок или фотографию, масла в огонь может подлить необходимость работы с документом с увеличенным масштабом. В подобной ситуации снизить остроту проблемы поможет область с полосами прокрутки `TScrollBox`.

Несколько другие задачи решает страница с кнопками прокрутки `TPageScroller`. Данный элемент управления в первую очередь предназначен для устранения проблемы нехватки рабочего пространства для управляющих элементов, например кнопок для инструментальной планки `TToolBar`. В этом случае `TPageScroller` может быть использован в качестве контейнера для непомещающейся в отведенное пространство группы кнопок, страница обеспечит просмотр принадлежащих ей компонентов за счет кнопок скроллинга.

Оба изучаемых контейнера являются потомками класса `TWinControl` и, как следствие, владеют всеми преимуществами оконных элементов управления (см. рис. 24.1).

Область с полосами прокрутки *TScrollBox*

Элемент управления `TScrollBox` представляет собой контейнер, обладающий собственными полосами прокрутки. Вовнутрь контейнера программист помещает необходимые компоненты (например, изображение `TImage`), в которые будет загружен обрабатываемый документ. Если размеры документа превысят клиентскую область контейнера, автоматически включаются полосы прокрутки, благодаря которым пользователь сможет просмотреть любую область окна, не уменьшая масштаба загруженного в окно документа.

Чтобы понять принцип работы элемента управления, разместите на главной форме нового проекта элемент управления `ScrollBox1:TScrollBox`, а затем положите в него компонент для просмотра изображений `Image1:TImage`.

Воспользовавшись событием `OnShow()` главной формы проекта, напишите несколько строк кода, загружающих в компонент `Image1` изображение большого размера (листинг 24.2).

Листинг 24.2. Загрузка графического образа

```
procedure TForm1.FormShow(Sender: TObject);
begin
  with Image1 do
  begin
    AutoSize:=True; //компонент автоматически устанавливает свои размеры
    Stretch:=False; //отключаем режим растягивания рисунка
    Proportional:=false; //отключаем поддержку пропорциональности
    Left:=0; Top:=0;
    Picture.LoadFromFile(путь и имя файла с рисунком);
  end;
end;
```

Рисунок 24.4 отражает результат работы нашего кода. После загрузки графического образа в компонент `Image1` последний автоматически установит свою ширину и высоту так, чтобы

вывести образ на экран. Как только размер `Image1` превысит габариты клиентской области `ScrollBox1`, включаются полосы прокрутки.

Нетрудно догадаться, что основными свойствами класса `TScrollBox` окажутся свойства, предоставляющие доступ к горизонтальной и вертикальной полосам прокрутки:

```
property HorzScrollBar: TControlScrollBar;
property VertScrollBar: TControlScrollBar;
```

Для того чтобы полосы прокрутки включались автоматически, убедитесь, чтобы свойство

```
property AutoScroll: Boolean; //по умолчанию true
```

было установлено в состояние `true`.

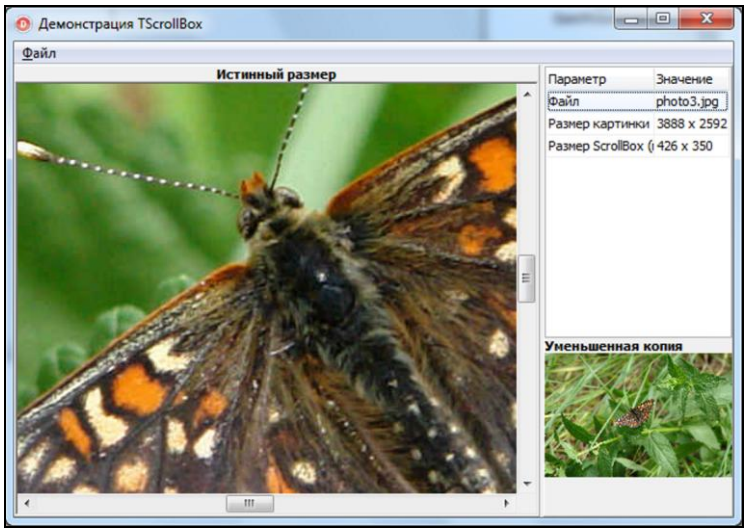


Рис. 24.4. Применение элемента управления `TScrollBox` для вывода большого изображения

Страница с кнопками прокрутки *TPageScroller*

Элемент управления `TPageScroller` (точно так же, как и его "коллега" — область прокрутки `TScrollBox`) специализируется на скроллинге области, частично не помещающейся в окно просмотра. Однако в отличие от своего более универсального напарника, которому абсолютно безразлично количество размещенных на его поверхности компонентов, страница с кнопками прокрутки `TPageScroller` предпочитает работать только с одним дочерним объектом.

Ссылку на принадлежащий `TPageScroller` объект можно обнаружить в свойстве

```
property Control: TWinControl;
```

У свойства `Control` есть еще одно применение — оно позволяет передавать в распоряжение компонента `TPageScroller` любой объект во время выполнения программы.

> ***Замечание***
>
> Обратите внимание, что страница с кнопками прокрутки рассчитывает на то, что в качестве дочернего элемента управления будет задействован только потомок класса `TWinControl`.

Еще одна особенность компонента TPageScroller заключается в том, что он не способен одновременно осуществлять скроллинг по вертикали и горизонтали. Поэтому в самом начале работы нам с вами требуется определиться с предпочтительным направлением прокрутки, для этого обращаемся к свойству

```
property Orientation: TPageScrollerOrientation;//soHorizontal
type TPageScrollerOrientation = (soHorizontal, soVertical);
```

Если принадлежащий компоненту TPageScroller объект полностью не помещается в заданные габариты, то на краях страницы появляются кнопки прокрутки. По умолчанию они крайне малы и занимают всего 12 пикселов, положение вещей можно исправить, обратившись к свойству

```
property ButtonSize: Integer; //по умолчанию 12
```

По умолчанию прокрутка видимой области осуществляется только по щелчку по кнопкам прокрутки, однако, установив свойство

```
property AutoScroll: Boolean; //по умолчанию false
```

в состояние true, мы добьемся того, что прокрутка начнется автоматически в момент появления указателя мыши над кнопкой скроллинга.

Предусмотрен и программный способ прокрутки окна компонента. Для осуществления скроллинга надо изменить значение свойства

```
property Position: Integer; //по умолчанию 0
```

Это же свойство можно использовать и в информационных целях — для оценки текущего местоположения.

В момент старта прокрутки вызывается событие

```
property OnScroll: TPageScrollEvent;
type TPageScrollEvent = procedure (Sender: TObject; Shift: TShiftState;
                  X, Y: Integer; Orientation: TPageScrollerOrientation;
                  var Delta: Integer) of object;
```

Здесь: Sender хранит ссылку на компонент TpageScroller; Shift контролирует состояние служебных клавиш; X и Y описывают горизонтальное и вертикальное положения страницы прокрутки; направление прокрутки хранится в параметре Orientation; последний параметр Delta устанавливает шаг скроллинга.

Разделитель панелей, компонент *TSplitter*

Последние страницы главы посвящены элементу управления, с точки зрения иерархии наследования классов не имеющему к панелям никакого отношения (класс TSplitter является прямым потомком графического класса TGraphicControl и, как следствие, не может быть владельцем других элементов управления). Вместе с тем TSplitter способен оказать панелям (и не только им) неоценимую помощь. Дело в том, что экземпляр класса TSplitter специализируется на разделении двух объектов (например, панелей TPanel) с одновременным предоставлением услуги по управлению их размерами. И для этого нашему брату-программисту не надо писать ни одной строки кода!

Для того чтобы компонент TSplitter смог проявить свои нетривиальные способности, разделяемые компоненты должны быть выровнены с помощью свойства Align. После этого

разделитель располагается между обслуживаемыми объектами, и по большому счету на этом программирование заканчивается: пользователь перемещает разделительную линию с помощью мыши, а все остальное TSplitter сделает сам...

Впрочем, небольшая инспекция возможностей компонента ему не повредит. По сравнению со своим предком разделитель богаче на четыре свойства. В первую очередь это свойство, определяющее минимальный размер разделяемого объекта, по достижению которого TSplitter перестанет уменьшать своего визави

```
property MinSize: NaturalNumber; //по умолчанию 30 пикселов
```

Достигнув минимального размера, уменьшаемый объект может полностью исчезнуть с экрана. Это произойдет, если свойство

```
property AutoSnap: Boolean;//по умолчанию true
```

установлено в true.

Эффект от перемещения разделителя тоже может быть изменен. Это позволит сделать свойство

```
property ResizeStyle: TResizeStyle;//по умолчанию rsPattern
type TResizeStyle = (rsNone, rsLine, rsUpdate, rsPattern);
```

По умолчанию компонент предлагает нам режим rsPattern. Особенность этого режима в том, что во время перемещения разделителя указателем мыши система реально не изменяет размеры обслуживаемых объектов до тех пор, пока пользователь не отпустит кнопку мыши в новом месте. Точно такой же результат достигается в состоянии rsLine, небольшое различие между двумя режимами заключается в особенности визуализации процесса. В первом случае разделительная полоса рисуется полупрозрачной, а во втором — обычной линией. Режим rsUpdate более нетерпелив, он заставит разделяемые компоненты моментально отреагировать на перемещение мыши и изменять свои размеры в динамике (не дожидаясь, пока пользователь соизволит отпустить левую кнопку). Режим rsNone в комментариях не нуждается — в этом состоянии TSplitter теряет работоспособность.

Процесс перемещения разделителя сопровождается парой событий. Первое из них просто информирует программиста, что пользователь передвигает разделительную полосу:

```
property OnMoved: TNotifyEvent;
```

Второе событие обладает большими возможностями:

```
property OnCanResize: TSplitterCanResizeEvent;
type TSplitterCanResizeEvent = procedure(Sender: TObject;
                  var NewSize: Integer; var Accept: Boolean) of object;
```

Здесь, контролируя предлагаемый системой новый размер NewSize, мы можем запретить (Accept:=false) или разрешить (Accept:=true) изменение габаритов объекта.

ГЛАВА 25

Инструментальные планки

В этой главе нам предстоит познакомиться с особой категорией элементов управления — планками (полосами, bar), предназначенными для быстрой разработки пользовательского интерфейса приложений Delphi. В сфере нашего внимания на этот раз окажутся компоненты (рис. 25.1):

- инструментальная планка `TToolBar`;
- планки-контейнеры `TCoolBar` и `TControlBar`;
- панель состояния `TStatusBar`.

Основное предназначение полос `TToolBar`, `TCoolBar` и `TControlBar` заключается в предоставлении услуг контейнера для визуальных компонентов, специализирующихся на управлении приложением. Инструментальная планка `TToolBar` служит пристанищем для разнообразных кнопок, комбинированных списков и других, небольших по размеру элементов управления, с помощью которых пользователь приобретает возможность получать быстрый доступ к тем или иным функциям приложения. Хорошим примером задействования инструментальной планки может стать интегрированная среда разработки Delphi. Сразу под главным меню IDE расположилось несколько инструментальных полос с кнопками, предоставляющими возможность оперативного обращения к наиболее часто востребованным операциям сохранения проекта, переключения форм, старта компиляции и т. п. Все кнопки снабжены значками и всплывающими подсказками, что позволяет создавать интуитивно понятный интерфейс приложения.

В отличие от инструментальной планки `TToolBar`, планки-контейнеры `TCoolBar` и `TControlBar` предпочитают работать не с отдельными кнопками, а с более значимыми объектами. Их основными клиентами обычно выступают такие объекты, как: `TToolBar`, `TActionMainMenuBar` и `TActionToolBar` *(см. главу 20)*. Таким образом, это своеобразные контейнеры контейнеров. Основная функциональная нагрузка полос `TCoolBar` и `TControlBar` заключается в предоставлении пользователю возможности переупорядочивать расположенные на их поверхности командные и инструментальные планки — настраивать интерфейс приложения под собственные пожелания.

Функциональная нагрузка, возложенная на планку состояния `TStatusBar`, принципиально отличается от задач, стоящих перед ее "коллегами". Хотя планка состояния также является наследником класса `TWinControl` и поэтому потенциально готова играть роль контейнера, но на этот раз даже не ведется речи о размещении на поверхности `TStatusBar` других компонентов с палитры VCL. Вместо этого статусная планка способна самостоятельно созда-

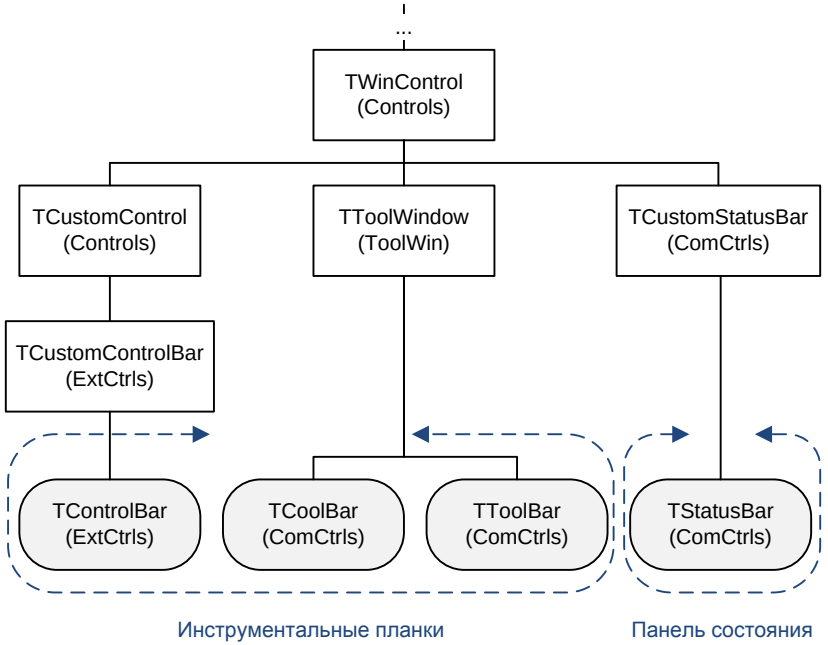


Рис. 25.1. Инструментальные планки в иерархии VCL

вать небольшие панельки, поверхность которых используется программистом для вывода текстовых подсказок о текущем статусе той или иной операции, состоянии служебных клавиш и т. п.

Инструментальная планка *TToolBar*

Инструментальная планка `TToolBar` представляет собой удобный контейнер для кнопок и других элементов управления. Однако в отличие о традиционной панели `TPanel`, инструментальная планка обладает рядом уникальных возможностей, значительно упрощающих процесс проектирования. Основная изюминка компонента `TToolBar` в том, что он способен создавать собственный тип кнопок. Поэтому, для того чтобы поместить на планку инструментов новую кнопку, не стоит перебирать элементы управления на палитре компонентов Delphi, а достаточно просто щелкнуть правой кнопкой мыши по инструментальной планке `TToolBar` и во всплывающем меню выбрать пункт **New Button**. В результате на полосе окажется кнопка собственного "сочинения" — экземпляр класса `TToolButton`.

Кнопка *TToolButton*

Инструментальная кнопка `TToolButton` построена на основе класса `TGraphicControl` и поэтому обладает всеми характерными чертами графических элементов управления, в частности такая кнопка не обладает дескриптором и не способна получать фокус ввода. С одной стороны это недостаток, однако благодаря отсутствию в цепочке наследования класса `TWinControl` графические элементы управления не потребляют столько ресурсов, как их "коллеги" — оконные элементы управления.

По умолчанию кнопка почти неотличима от классической "быстрой" кнопки `TSpeedButton`, однако с помощью свойства

```
property Style: TToolButtonStyle; //по умолчанию tbsButton
```

можно значительно расширить ее возможности (табл. 25.1).

Таблица 25.1. *Возможные стили инструментальной кнопки* `TToolButtonStyle`

Значение	Описание
`tbsButton`	Обычная кнопка с поведением, аналогичным `TSpeedButton`
`tbsCheck`	Кнопка с двумя фиксированными состояниями: кнопка утоплена (свойство `Down=true`) и кнопка приподнята (`Down=false`)
`tbsDropDown`	Симбиоз кнопки и контекстного меню, кроме реакции на обычный щелчок кнопка позволяет инициировать выпадение пунктов своего меню (см. свойство `DropdownMenu`)
`tbsSeparator`	Заменяет кнопку разделителем, он предназначен для визуального отделения друг от друга групп кнопок
`tbsDivider`	Вторая разновидность разделителя, единственное отличие от стиля `tbsSeparator` (пустого пространства) — наличие вертикальной разделительной линии
`tbsTextButton`	Текстовая кнопка (см. свойство `AllowTextButtons`)

Если стиль `tbsCheck` назначен более чем для одной кнопки, то эти кнопки могут быть сгруппированы. Для этого установите в `true` свойство

```
property Grouped: Boolean;
```

На поведение объединенных в группу кнопок оказывает влияние свойство

```
property AllowAllUp: Boolean;
```

Если оно установлено в `false`, то это означает, что в группе всегда одна из кнопок должна быть нажата. После этого поведение сгруппированных кнопок начинает напоминать повадки переключателей (радиокнопок) — при нажатии на одну из кнопок предыдущая нажатая кнопка выводится из отмеченного состояния.

Для программного нажатия кнопки переведите свойство

```
property Down: Boolean;
```

в состояние `true`.

Установив в `true` свойство

```
property Indeterminate: Boolean;
```

вы сможете перевести кнопку в третье, неопределенное состояние.

Для того чтобы пометить кнопку, воспользуйтесь свойством

```
property Marked: Boolean;
```

Визуально отмеченная кнопка отличается дополнительным затенением.

Значительный практический интерес представляет стиль tbsDropDown, позволяющий интегрировать в кнопку контекстное меню. Для этого программист ссылается на компонент TPopupMenu или TPopupActionBar в свойстве

```
property DropdownMenu: TPopupMenu;
```

Для превращения кнопки в текстовую кнопку (напоминающую TButton) недостаточно перевести ее стиль в режим tbsTextButton, кроме этого следует установить в состояние true свойство

```
property AllowTextButtons: Boolean;
```

Допустим, что мы разрабатываем свою версию менеджера файлов, готовую (по своим функциональным возможностям) составить конкуренцию Проводнику из стандартной поставки Windows. Как и положено качественному менеджеру файлов, наше приложение должно предоставлять пользователю удобный интерфейс, позволяющий выводить подробную информацию о хранящихся на дисках компьютера файлах и папках.

Если вы внимательно изучили *главу 21*, то уже догадались, что для этой цели как нельзя лучше подойдет элемент управления TListView — у компонента имеется несколько режимов представления данных, определяемых свойством ViewStyle. Управляя названным свойством, мы приобретем возможность переключать режим просмотра данных (табличный, список, большие и маленькие значки) точно так, как это делает Проводник. Для управления режимом представления данных в компоненте TListView как нельзя лучше подойдут инструментальная полоса TToolBar и кнопка TToolButton, работающая с контекстным меню.

Допустим, что в нашем распоряжении имеются следующие элементы управления (рис. 25.2):

- список ListView1:TListView, в который мы научили собирать сведения о папках компьютера *(см. главы 6 и 21)*;
- менеджер команд ActionManager:TActionManager, в составе которого имеются четыре команды, управляющие режимом просмотра ViewStyle списка:
  - Name=acIcon, Caption='Большие значки', Tag=0;
  - Name=acSmallIcon, Caption='Маленькие значки', Tag=1;
  - Name=acList, Caption='Список', Tag=2;
  - Name=acReport, Caption='Таблица', Tag=3;
- инструментальная планка ToolBar1:TTollBar с кнопкой btnViewStyle:TToolButton, стиль кнопки Style установлен в состояние tbsDropDown;
- к свойству DropDownMenu кнопки btnViewStyle присоединено контекстное меню PopupActionBar1:TPopupActionBar1, которое обладает четверкой пунктов:
  - N1:TMenuItem, Action=acIcon;
  - N2:TMenuItem, Action=acSmallIcon;
  - N3:TMenuItem, Action=acList;
  - N4:TMenuItem, Action=acReport.

Проницательный читатель уже наверняка догадался, что обращение к команде acIcon должно переключить список ListView1 в режим представления данных "Большие значки" (свойству ViewStyle компонента присваивается значение vsIcon), команда acSmallIcon переключит просмотр в vsSmallIcon и т. д. Мы не станем создавать четыре однотипных обработчика события для каждой из команд, а поступим хитрее — разработаем универсальный код и разделим его между всеми командами (листинг 25.1).

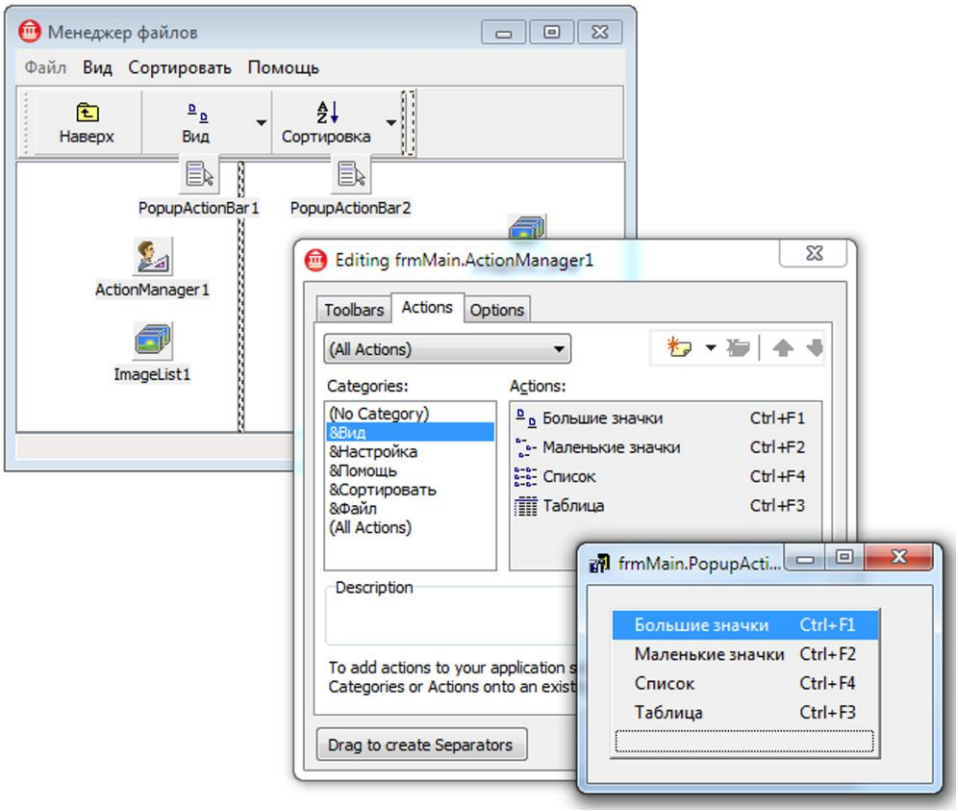


Рис. 25.2. Состав приложения Менеджер файлов

Листинг 25.1. Общий обработчик события для команд `TAction`

```
procedure TfrmMain.acIconExecute(Sender: TObject);
var ViewStyle:TViewStyle;
begin//управляем стилем отображения LV:TListView
  ViewStyle:=TViewStyle((Sender as TAction).Tag); //узнаем стиль
  LV.ViewStyle:=ViewStyle; //изменим стиль
  btnViewStyle.Tag:=(Sender as TAction).Tag; //кнопка запомнит стиль
end;
```

Идея, заложенная в код события, такова: каждая команда идентифицируется уникальным значением, находящимся в свойстве `tag`. У команды `acIcon` свойство `tag` равно `0`, что соответствует стилю `vsIcon`; у команды `acSmallIcon` свойство `tag` равно `1` (стиль `vsSmallIcon`) и т. д. Таким образом, получая значение `tag` команд, мы изменим стиль отображения данных в элементе управления `ListView1`.

Внимание!

Листинг 25.1 описывает событие `OnExecute()` только для команды `acIcon`, обязательно с помощью страницы **Events** Инспектора объектов свяжите его со всеми событиями `OnExecute()` остальных команд нашего примера.

Наш код уже вполне работоспособен, щелчок по соответствующему пункту контекстного меню (выпадающего из кнопки `btnViewStyle`) переключит режим отображения списка. Но предлагаю пойти еще дальше — обработать обычный щелчок по кнопке `btnViewStyle` (листинг 25.2).

Листинг 25.2. Обычный щелчок по кнопке инструментальной планки

```
procedure TfrmMain.btnViewStyleClick(Sender: TObject);
var NewViewStyle:TViewStyle;
begin
  NewViewStyle:=TViewStyle(btnViewStyle.Tag+1); //даем приращение tag
  case NewViewStyle of //теперь вызываем нужное действие
    vsSmallIcon: acSmallIcon.Execute;
    vsList     : acList.Execute;
    vsReport   : acReport.Execute;
    else acIcon.Execute;
  end;
end;
```

Вы заметили, что в предпоследней строке листинга 25.1, в свойстве `tag` кнопки `btnViewStyle` мы сохранили числовой код текущего стиля отображения. Это мы сделали с далеко идущими намерениями. Теперь каждый щелчок по кнопке будет изменять режим отображения `ViewStyle`, "пролистывая" стиль представления данных о папках и файлах по кругу: `vsIcon`, `vsSmallIcon`, `vsList`, `vsReport` и вновь `vsIcon`. Плоды нашей работы отражает рис. 25.3.

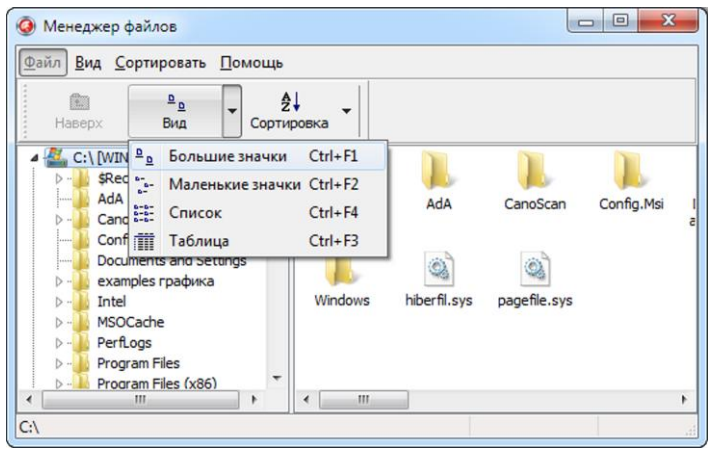


Рис. 25.3. Совместная работа кнопки `TToolButton` и контекстного меню

Управление кнопками

Как уже отмечалась ранее, хотя инструментальная планка способна служить контейнером практически для всех стандартных элементов управления, но основное предназначение рассчитано на совместную работу со специальными кнопками `TToolButton`.

Кнопки хранятся в свойстве-массиве

```
property Buttons[Index: Integer]: TToolButton;
```

Общее число экземпляров кнопок мы узнаем из свойства

```
property ButtonCount : Integer; //только для чтения
```

Геометрические размеры кнопок устанавливаются в свойствах

```
property ButtonHeight: Integer;//высота кнопки
property ButtonWidth: Integer; //ширина кнопки
```

Однако это не относится к кнопкам, играющим роль разделителей (свойство `Style` установлено в `tbsSeparator` или `tbsDivider`).

По умолчанию панель инструментов не хочет, чтобы принадлежащие ей кнопки отображали свои заголовки. Для изменения состояния дел передайте `true` в свойство

```
property ShowCaptions: Boolean;
```

Единственный недостаток этого решения в том, что кнопки с заголовками потребуют значительно больше пространства для своего размещения.

Пользовательские настройки

Одной из полезных черт инструментальной панели является возможность настраивать содержимое панели во время выполнения программы. Для этого достаточно в программном коде установить в `true` свойство

```
property Customizable : Boolean;
```

После этого пользователь получит превосходную возможность двигать и удалять кнопки (при помощи мыши и удержания в нажатом состоянии клавиши <Shift>). Еще один способ редактирования панели управления пользователь получит после двойного щелчка по ее поверхности.

Для того чтобы изменения, вносимые пользователем, не пропадали даром и загружались при следующем запуске программы, надо описать два свойства:

```
property CustomizeKeyName : string;   //имя ключа реестра
property CustomizeValueName : string; //название значения
```

В первом свойстве назначается имя раздела (в секции `HKEY_CURRENT_USER` реестра Windows), в котором будут сохраняться изменения. Второе свойство определяет имя подраздела, с которым и работает панель инструментов.

С процессом редактирования содержимого панели связан целый ряд событий (все они начинаются со словосочетания `OnCustom...`).

Оформление

Для того чтобы снабдить каждую кнопку соответствующим значком, компонент `TToolBar` требует подключения коллекции картинок (компонента `TImageList`) к свойству

```
property Images : TCustomImageList;
```

Затем, используя свойства `ImageIndex` каждой из кнопок, выбираем подходящее изображение. Если кнопка переходит в пассивное состояние (`Enabled=false`), то на ее поверхности

вместо цветного изображения выводится его монохромная версия. Если такое поведение вам не подходит, то подключите еще одну коллекцию "пассивных" картинок к свойству

```
property DisabledImages : TCustomImageList;
```

В дополнение ко всему программист обладает возможностью задать для каждой кнопки еще одну пиктограмму, отображаемую в момент прохождения над поверхностью кнопки указателя мыши. Для этой цели потребуется очередной (третий по счету) компонент `TImageList` и свойство панели инструментов

```
property HotImages : TCustomImageList;
```

Для придания кнопкам плоского вида установите в `true` свойство

```
property Flat: Boolean;
```

За прозрачность панели инструментов отвечает свойство

```
property Transparent: Boolean;
```

Отступ от левого края панели инструментов до первой кнопки (или другого элемента управления) задается свойством

```
property Indent: Integer;
```

По умолчанию значение отступа установлено в 0.

Заголовок кнопки можно отображать под значком или справа от него. Для этого измените значение свойства

```
property List: Boolean;
```

Если панель инструментов будет поддерживать интерфейс буксировки — drag and dock, то имеет смысл назначить заголовок и для самой инструментальной полосы:

```
property Caption: TCaption;
```

Назначенный в свойстве текст окажется в заголовке находящегося вне дока окна планки.

Если весь ряд элементов не в состоянии уместиться на панели инструментов из-за ограничений ее горизонтального размера, то для организации автоматического переноса кнопок в новый ряд установите в `true` свойство

```
property Wrapable: Boolean;
```

В качестве индикатора количества рядов следует использовать свойство

```
property RowCount: Integer;
```

Интересного поведения панели инструментов можно добиться, воспользовавшись его свойством

```
property Menu: TMainMenu;
```

Тогда на поверхности компонента появится точная копия всех пунктов главного меню приложения.

Планка *TCoolBar*

Элемент управления на основе класса `TCoolBar` (если перевести дословно — стильная планка) предназначен для выполнения роли контейнера для других элементов управления. При размещении на поверхности стильной планки какого-нибудь компонента (обычно на эту

роль назначаются экземпляры классов TToolBar, TActionMainMenuBar и TActionToolBar) для него автоматически создается отдельная дочерняя полоса — TCoolBand, которую впоследствии можно перемещать, изменять ее размеры, скрывать и вновь показывать и т. п.

> ***Замечание***
>
> Планка TCoolBand создается только для обслуживания компонентов-потомков класса TWinControl. При размещении на TCoolBar компонентов, не состоящих в родственных связях с названным классом, стильная планка особо не возмутится, но и отдельной полосы для непрошеного гостя не создаст.

Основу компонента TCoolBar составляет хранилище полос

```
property Bands: TCoolBands;
```

предоставляющее доступ к дочерним полосам TCoolBand.

Число принадлежащих коллекции полос доступно благодаря свойству

```
property Count: Integer; //только для чтения
```

Для обращения к отдельной полосе передайте ее индекс в свойство

```
property Items[Index: Integer]: TCoolBand;
```

Дочерние полосы могут быть снабжены индивидуальными значками, но для этого следует подключить коллекцию с картинками к свойству

```
property Images: TCustomImageList;
```

Если необходимо к списку полос программным способом добавить новую, то обратитесь к функции

```
function Add: TCoolBand;
```

Иногда необходимо уточнить, на какой из полос TCoolBand размещен тот или иной элемент управления. Для решения такой задачи приспособлен метод

```
function FindBand(AControl: TControl): TCoolBand;
```

Единственный параметр функции требует передачи ссылки на элемент управления.

С выравниванием полос на поверхности планки связано знакомое нам свойство

```
property Align: TAlign;
```

Кроме того, Align в некотором роде продублировано свойством

```
property Vertical: Boolean;
```

По умолчанию оно установлено в false. Это означает, что дочерние полосы расположены по горизонтали, перевод свойства в состояние true переориентирует полосы на вертикальное размещение.

Наиболее важное событие TCoolBar возникает при изменении местоположения или размеров ее дочерних полос.

```
property OnChange: TNotifyEvent;
```

Дочерняя полоса *TCoolBar*

Практически все свойства класса TCoolBar нацелены на обслуживание внешнего вида полосы и ее поведения при изменении размера (табл. 25.2).

Таблица 25.2. *Ключевые свойства дочерней полосы* `TCoolBar`

Свойство	Описание
`property Control: TWinControl;`	Ссылка на принадлежащий полосе объект
`property ImageIndex: TImageIndex;`	Индекс индивидуальной картинки полосы
`property ParentBitmap: Boolean;`	В состоянии `true` позволит дочерней полосе в качестве фоновой картинки использовать изображение с родительского компонента `TCoolBar`
`property Bitmap: TBitmap;`	Назначает панели личную фоновую картинку, в этом случае свойство `ParentBitmap` автоматически установится в `false`
`property MinWidth: Integer;`	Определяет минимальную ширину полосы
`property MinHeight: Integer;`	Минимальная высота полосы
`property FixedSize : Boolean;`	В состоянии `true` запрещает пользователю изменять размеры полосы
`property Text : String;`	Подпись полосы
`property Break: Boolean;` `{по умолчанию True}`	В состоянии `true` требует, чтобы для новой полосы создавался новый ряд

Планка управления *TControlBar*

Планка управления `TControlBar` — это своего рода аналог изученного только что компонента `TCoolBar`. Разница только в том, что `TCoolBar` рожден в Microsoft и спрятан в библиотеке COMCTL32.DLL, а наш новый знакомый создан разработчиками Delphi и не зависит от библиотек операционной системы.

Основная задача компонента `TControlBar` — служить контейнером для других элементов управления. Для каждого располагаемого на его поверхности компонента формируется отдельная обрамляющая панель. Как правило, в роли клиентов планки управления применяют объекты `TToolBar` и `TActionMenuBar`. Очень часто `TControlBar` используется программистами в качестве "посадочной площадки" в операциях буксировки компонентов.

Подавляющее число свойств и методов класса унаследовано от `TWinControl` и достаточно полно отражено в предыдущих главах книги, поэтому мы сосредоточимся только на ключевых особенностях полосы управления, а это в первую очередь поддержка операций буксировки drag and dock.

Для превращения полосы управления в простейший док нет необходимости сочинять ни единой строки кода. Для проверки этого утверждения создадим новый проект и разместим на главной форме элемент управления `TControlBar`. Выровняем планку по верхнему краю формы (`Align:=alTop`). Теперь перенесите на поверхность `ControlBar1` любой оконный или графический элемент управления, например панель управления `TToolBar`. Установим порядок перетягивания данного элемента (`DragKind:=dkDock`) и включим автоматический режим (`DragMode:=dmAutomatic`). Запустите проект на выполнение, ведь "программирование" уже закончилось.

Если вдруг планка управления `TControlBar` отказывается поддерживать операции буксировки, то обязательно проверьте свойство

```
property AutoDock : Boolean; //по умолчанию true
```

Свойство должно быть установлено в `true`. Это признак того, что компонент автоматически предоставляет "посадочную площадку" для буксируемого компонента.

К особенностям полосы управления стоит отнести связанные с поведением буксируемых элементов управления обработчики событий. Во всех этих событиях параметр `Control` хранит ссылку на буксируемый элемент (табл. 25.3).

Таблица 25.3. *Реакция на процесс буксировки* `TControlBar`

Свойство	Описание
property OnBandDrag: TBandDragEvent; **type** TBandDragEvent = **procedure** (Sender: TObject; Control: TControl; **var** Drag: Boolean) of object;	Вызывается в начале операции перетягивания, параметр `Drag` разрешает (`true`) или запрещает (`false`) перетягивание буксируемого элемента `Control`
property OnBandMove: TBandMoveEvent; **type** TBandMoveEvent = **procedure** (Sender: TObject; Control: TControl; **var** ARect: TRect) of object;	Генерируется во время буксировки полосы
property OnBandInfo: TBandInfoEvent; **type** TBandInfoEvent = **procedure** (Sender: TObject; Control: TControl; **var** Insets: TRect; **var** PreferredSize, RowCount : Integer) of object;	Возникает в момент размещения буксируемого компонента в доке. Здесь `Insets` — пространство между левым верхним углом полосы и левым верхним углом буксируемого элемента. Корректируя это значение, программист может изменять местоположение элемента управления. `PreferredSize` предпочтительная ширина панели обрамления буксируемого элемента. Если размер неудовлетворителен, то он должен быть уменьшен. Параметр `RowCount` содержит количество рядов

Для того чтобы полоса управления смогла самостоятельно подстроить свои геометрические размеры под находящийся на ней элемент управления, установите в `true` свойство

```
Property AutoSize : Boolean;
```

Вполне естественно, что полоса управления предъявляет определенные требования к размерам потенциальных клиентов. Поэтому предельный вертикальный размер компонента ограничивается высотой строки:

```
property RowSize: TRowSize; //по умолчанию 26 пикселов
```

Для творческих натур может пригодиться свойство

```
property Picture: TPicture;
```

Это выводимая на полосе фоновая картинка.

В последовательности операций перерисовки полосы первоначально выводится фоновое изображение (свойство `Picture`), затем вызывается обработчик `OnPaint()` и в самом конце — `OnBandPaint()`. Именно в этом событии следует осуществить более тонкую настройку особенностей графического вывода полосы.

```
property OnBandPaint: TBandPaintEvent;
type TBandPaintEvent = procedure (Sender: TObject; Control: TControl;
                Canvas: TCanvas; var ARect: TRect;
                var Options: TBandPaintOptions) of object;
```

Здесь: `Sender` — полоса управления; `Control` — перерисовываемый элемент управления; `Canvas` — канва панелей обрамления элемента управления; `aRect` — координаты прямоугольной области панели обрамления.

Панель состояния *TStatusBar*

Панель состояния `TStatusBar` отвечает за информирование пользователя о каких-либо параметрах приложения во время его выполнения, например: уведомление о нажатых служебных клавишах, строку подсказки, текущее время (рис. 25.4).

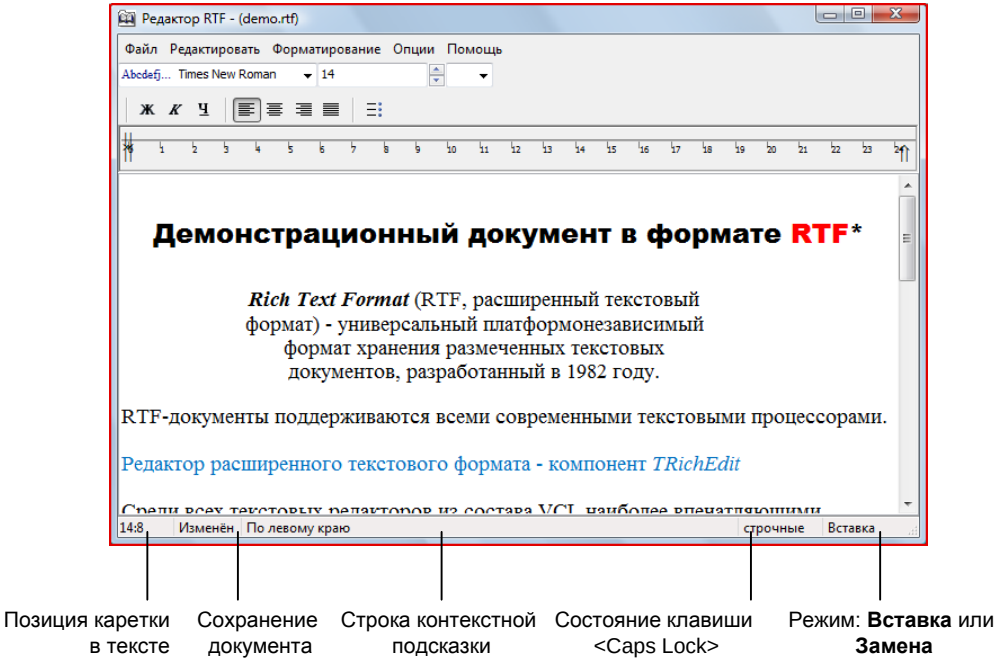


Рис. 25.4. Роль статусной планки `TStatusBar` в приложении

Простейший прием работы с `TStatusBar` заключается в переводе статусной планки в режим простой панели, для этого следует передать значение `true` в свойство

```
property SimplePanel: Boolean; //по умолчанию false
```

В таком случае программист получает весьма скромную возможность вывода на поверхности статусной панели единственной строки текста

```
property SimpleText: string;
```

Подобное использование планки, мягко говоря, не рационально, тем более что с ролью отображения строки текста без хлопот справится элементарная метка `TLabel` или обычная панель `TPanel`. Так что про режим `SimplePanel=true` забудем сразу, тем более что он не поддерживает ключевую особенность статусной планки — способность содержать несколько элементарных панелек (экземпляров класса `TStatusPanel`), которые могут отображать независимую информацию.

Для создания статусных панелек достаточно вызвать контекстное меню компонента и выбрать в нем пункт **Panels Editor**. Редактор позволяет добавлять, перемещать и убирать ненужные панельки. Тот же результат можно получить, обратившись к свойству

```
property Panels: TStatusPanels;
```

предоставляющему доступ к коллекции панелек во время визуального проектирования.

Для управления составом панелей из работающего приложения достаточно вспомнить навык работы со свойствами `Items` в компонентах-списках. Каждая отдельная панелька обладает индексом, воспользовавшись которым мы сможем обращаться к требуемой панели:

```
property Items[Index: Integer]: TStatusPanel;
```

Самая важная характеристика элементарной панели `TStatusPanel` — выводимый ею текст

```
property Text: string;
```

Разместите на форме статусную полосу и создайте на ней 4 панельки. Найдите свойство `KeyPreview` формы и переведите его в режим `true`, это действие заставит форму контролировать все нажатия клавиш. Далее обращаемся к событию `OnKeyDown()` формы и повторяем предложенный в листинге 25.3 код.

Листинг 25.3. Отображение состояния служебных клавиш на статусной полосе

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
                             Shift: TShiftState);
const MASK=$FFFE;  //значение маски 1111 1111 1111 1110
var State:short;
begin
  State:=GetKeyState((VK_CAPITAL)) OR MASK; {Состояние клавиши CapsLock}
  if state=-1 then StatusBar1.Panels[0].Text:='ПРОПИСНЫЕ'
              else StatusBar1.Panels[0].Text:='строчные';

  State:=GetKeyState((VK_INSERT)) OR MASK; {Состояние клавиши Insert}
  if state=-1 then StatusBar1.Panels[1].Text:='Замена'
              else StatusBar1.Panels[1].Text:='Вставка';

  State:=GetKeyState((VK_NUMLOCK)) OR MASK; {Состояние клавиши NUMLOCK}
  if state=-1 then StatusBar1.Panels[2].Text:='Num Lock'
              else StatusBar1.Panels[2].Text:='';

  State:=GetKeyState((VK_SCROLL)) OR MASK; {Состояние клавиши SCROLL}
  if state=-1 then StatusBar1.Panels[3].Text:='Scroll Lock'
              else StatusBar1.Panels[3].Text:='';
end;
```

Состояние служебных клавиш <Caps Lock>, <Insert>, <Num Lock> и <Scroll Lock> проверяет функция `GetKeySate()`. Если старший бит возвращаемого функцией значения установлен в 1, то это означает, что клавиша в работе. В этом случае на поверхности соответствующей панельки отобразится нужная надпись.

Для того чтобы пространство панельки позволяло полностью вывести назначенный ей текст, следует установить наиболее подходящую ширину:

```
property Width: Integer;
```

Допустим, что в нашем проекте используется статусная полоса, содержащая 5 панелек так, как было представлено на рис. 25.4. Понятно, что наибольшее пространство потребуется для панельки с индексом 2, ведь на ней выводится достаточно длинный текст контекстной подсказки. Неплохо было бы научить наше приложение при изменении размеров главного окна отводить под эту панельку максимально возможное пространство, при этом не ущемляя права остальных панелек. Один из способов решения этой задачи предложен в листинге 25.4.

Листинг 25.4. Настройка размеров статусных панелек

```
procedure TfrmMain.StatusBar1Resize(Sender: TObject);
begin
  with StatusBar1 do
    Panels[2].Width:=ClientWidth-Panels[0].Width-
            Panels[1].Width-Panels[3].Width-Panels[4].Width-15;
end;
```

В момент вызова события `OnResize()` полосы состояния мы вычисляем свободное пространство и отдаем его в распоряжение наиболее нуждающейся панели.

Внешний вид панельки определяется состоянием свойства

```
property Bevel: TStatusPanelBevel; //по умолчанию pbLowered
```

По умолчанию панелька "утапливается" (`Bevel=pbLowered`). Кроме того, можно создать выпуклую `pbRaised` и плоскую `pbNone` панельку.

Если свойство

```
property AutoHint : Boolean;
```

установлено в состояние `true`, то статусная панель получит возможность перехвата всплывающих подсказок. Для этого нам понадобится событие

```
property OnHint : TNotifyEvent;
```

и знание, что текст всплывающей подсказки окажется в свойстве `Hint` приложения (листинг 25.5).

Листинг 25.5. Отображение подсказок приложения

```
procedure TForm1.StatusBar1Hint(Sender: TObject);
begin
  StatusBar1.Panels[2].Text:=Application.Hint;
end;
```

Для желающих внести свой личный вклад в процесс прорисовки элементарных панелек пригодится событие

```
property OnDrawPanel: TDrawPanelEvent;
type TDrawPanelEvent = procedure(StatusBar: TStatusBar;
                     Panel: TStatusPanel; const Rect: TRect) of object;
```

Здесь: `StatusBar` — ссылка на компонент; `Panel` — выводимая в данный момент на экран панелька; `Rect` — прямоугольные координаты панельки.

Имейте в виду, что указанное событие возникнет, только если свойство

```
property Style: TStatusPanelStyle; //по умолчанию psText
```

установлено в состояние `psOwnerDraw`.

Панельки `TStatusPanel` могут быть созданы и удалены во время выполнения приложения. Добавление панели обеспечивается вызовом метода

```
function Add: TStatusPanel;
```

В этом случае новая элементарная панелька присоединяется к концу списка. Вдобавок экземпляр `TStatusPanel` можно вставить и в запланированное место, но в этом случае потребуется обратиться к другому методу:

```
function Insert(Index: Integer): TStatusPanel;
```

Удаление панельки осуществляет метод

```
procedure Delete(Index: Integer);
```

В момент рождения новой панельки у ее владельца (полосы состояния) генерируется событие

```
property OnCreatePanelClass: TSBCreatePanelClassEvent;
type TSBCreatePanelClassEvent = procedure(Sender: TCustomStatusBar;
                          var PanelClass: TStatusPanelClass) of object;
```

Это событие должно применяться только в тех случаях, когда нас по какой-то причине не устраивает создаваемая по умолчанию панелька. В рамках события программист вправе подменить панельку `TStatusPanel` панелькой собственной редакции, для этого предназначен параметр `PanelClass`. При этом нельзя забывать о том, что наш вариант панельки должен быть наследником класса `TStatusPanel`.

ГЛАВА 26

Наборы закладок и блокноты

Одна из распространенных проблем, с которой неминуемо сталкивается любой разработчик пользовательского интерфейса приложения, связана с нехваткой пространства на формах проекта для размещения на них всех необходимых элементов управления. Зачастую после долгих попыток оптимизации размеров и расположения компонентов программный интерфейс становится настолько неудобным, что начинает отпугивать пользователей программы.

Что делать, если логика обрабатываемых приложением данных требует расположения на форме большого числа элементов управления, но это пожелание идет вразрез с физическими возможностями экрана компьютера? Один из способов устранения проблемы нехватки рабочего пространства основан на использовании в нашем проекте компонентов-закладок и блокнота (рис. 26.1).

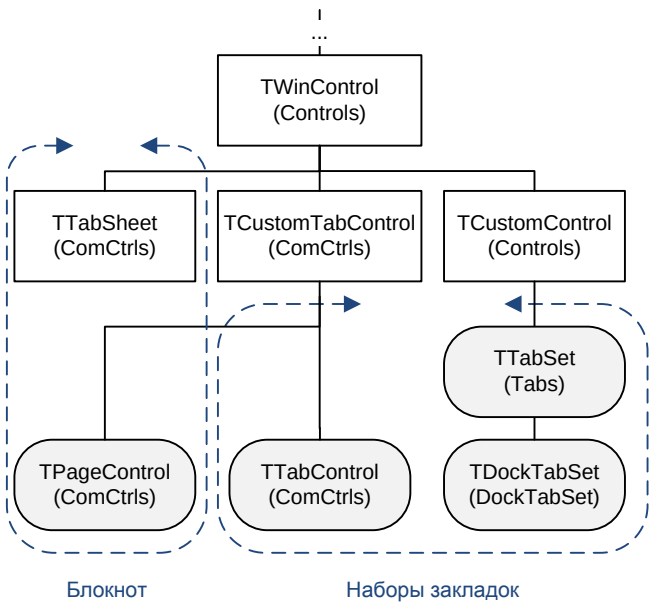


Рис. 26.1. Компоненты-закладки и компоненты-блокноты в иерархии классов VCL

Набор закладок, *TTabControl*

Своим внешним видом элемент управления `TTabControl` очень напоминает несколько закладок, вложенных в книгу. Чтобы убедиться в таком внешнем сходстве, достаточно взглянуть на рис. 26.2, на котором представлено несколько вариантов оформления элемента управления `TTabControl`.

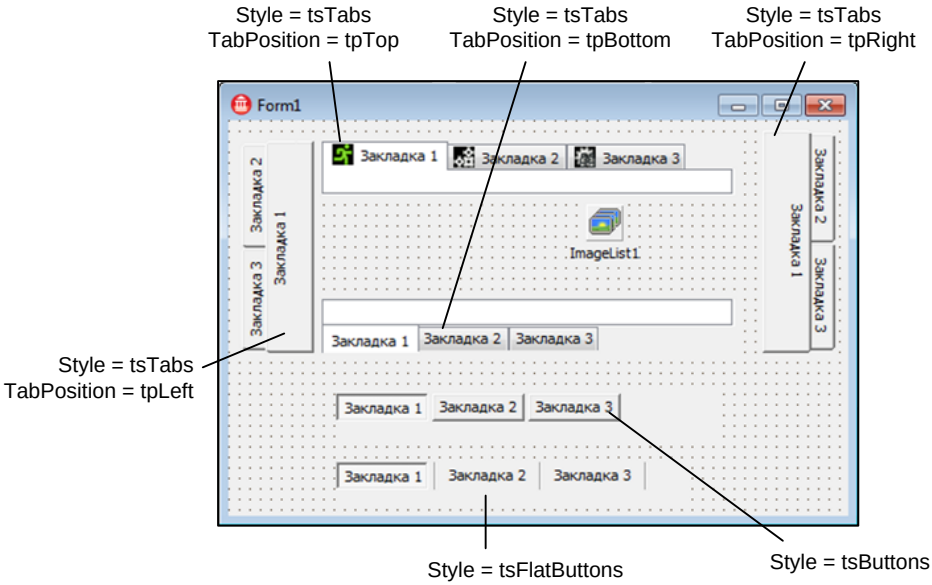


Рис. 26.2. Вариации внешнего вида набора закладок `TTabControl`

Так как по своей сути отдельная закладка является не более чем символьной строкой, создатели компонента остановились на традиционном способе хранения строк: текстовые названия закладок заносятся в инкапсулированный в элемент управления — список строк `TStrings`. Доступ к перечню закладок осуществляется благодаря свойству

```
property Tabs : TStrings;
```

Внешний вид закладок в первую очередь определяется их стилем:

```
property Style: TTabStyle; //по умолчанию tsTabs
type TTabStyle = (tsTabs,        //закладки
                  tsButtons,     //кнопки
                  tsFlatButtons);//плоские кнопки
```

Если при работе с набором закладок оставлен стиль по умолчанию (`tsTabs`), то программист получит дополнительную сервисную возможность — определить позицию закладок

```
property TabPosition: TTabPosition; //по умолчанию tpTop
type TTabPosition = (tpTop, tpBottom, tpLeft, tpRight);
```

По умолчанию `TabPosition` установлено в состояние `tpTop`, что соответствует выравниванию по верхнему краю компонента.

Выбор пользователем закладки визуально приводит к переводу этой закладки на передний план или (если `Style=tsButtons` или `tsFlatButtons`) нажатию соответствующей кнопки.

Для идентификации активной закладки обращаются к свойству

```
property TabIndex : Integer;
```

С его помощью выясняется порядковый номер выбранной пользователем закладки. Кроме того, свойство позволяет активировать необходимую закладку программным образом. Индекс самой первой закладки соответствует 0, последней — `Tabs.Count-1`. Если ни одна закладка не выбрана, то свойство возвратит значение –1.

Вполне допустим выбор нескольких закладок одновременно, для этого установите в `true` свойство

```
property MultiSelect: Boolean; //по умолчанию false
```

Но при этом нельзя забывать, что в этом случае стиль закладок должен быть установлен в состояние `tsButtons` или `tsFlatButtons`.

По умолчанию за назначение размера закладок отвечает сам компонент, однако эти права можно у него отнять и самостоятельно определить пространство, отводимое под закладку с помощью свойств

```
property TabHeight: Smallint; //по умолчанию 0
property TabWidth: Smallint;  //по умолчанию 0
```

Если этим свойствам возвратить значения 0, то вновь включится автоматическое управление размерами.

По умолчанию закладки размещаются слева направо вдоль одной линии. Для того чтобы разрешить многострочное расположение закладок, установите в `true` свойство

```
property MultiLine: Boolean; //по умолчанию false
```

Компонент постарается самостоятельно перестроить ряды закладок так, чтобы закладки заняли все доступное пространство. Для уточнения числа рядов следует вызвать метод

```
function RowCount: Integer;
```

Порядок расположения закладок (кнопок) на поверхности компонента также зависит от свойства

```
property RaggedRight : boolean;//по умолчанию false
```

Свойство контролирует, чтобы закладки занимали все доступное пространство по горизонтали.

При многострочном расположении закладок поведение рядов, на которых выделены закладки, зависит от состояния свойства

```
property ScrollOpposite : Boolean; //по умолчанию false
```

Установив свойство в `true`, вы добьетесь визуального эффекта перемещения ряда с выделенной закладкой "навстречу" пользователю.

Если вы не воспользовались многострочной системой размещения закладок, то вполне вероятно, что все закладки физически не поместятся на поверхности компонента `TTabControl`. В таком случае в правом углу списка закладок появятся две кнопки скроллинга, обеспечивающие перемещение к невидимым закладкам. Для программного скроллинга у компонента реализован метод

```
procedure ScrollTabs(Delta: Integer);
```

Направление и шаг перемещения зависит от знака и величины значения, переданного в параметр `Delta` процедуры.

На преобразовании экранных координат (`X`, `Y`) в индекс закладки специализируется метод

```
function IndexOfTabAt(X, Y: Integer): Integer;
```

Если в точке с указанными координатами закладка отсутствует, то функция вернет значение –1.

Координаты прямоугольной области закладки с порядковым номером `Index` возвращает функция

```
function TabRect(Index: Integer): TRect;
```

В классе `TTabControl` реализовано несколько специфичных обработчиков событий. Первые два связаны со сменой активной закладки. Перед моментом смены закладки возникает событие

```
property OnChanging : TTabChangingEvent;
type TTabChangingEvent = procedure (Sender: TObject;
                                    var AllowChange: Boolean) of object;
```

В рамках события программист способен запретить смену закладки, для этого надо установить в `false` переменную `AllowChange`.

После смены закладки вызывается событие

```
property OnChange: TNotifyEvent;
```

Если на динамически создаваемой закладке нужно отобразить определенный значок, то для этого следует подключить к набору закладок коллекцию изображений

```
property Images: TCustomImageList;
```

Недостаток такого решения в том, что отдельная закладка (не являясь самостоятельным объектом) не может явным образом определить подходящий для нее индекс пиктограммы. Поэтому первая пиктограмма из коллекции изображений выводится на первой закладке, вторая — на второй и т. д. Однако ситуацию можно исправить, воспользовавшись услугами события

```
property OnGetImageIndex: TTabGetImageEvent;
type TTabGetImageEvent = procedure(Sender: TObject; TabIndex: Integer;
                                   var ImageIndex: Integer) of object;
```

Рассматриваемое событие генерируется единственный раз сразу после добавления к набору новой закладки. Здесь `TabIndex` — индекс закладки, в переменной `ImageIndex` вы устанавливаете порядковый номер картинки из подключенного к набору закладок компонента `TImageList`.

Программисту предоставлена возможность усовершенствовать процесс графического вывода закладки, для этого следует отказаться от режима прорисовки по умолчанию, установив свойство

```
property OwnerDraw: Boolean;//по умолчанию false
```

в состояние `true` и описать обработчик события

```
property OnDrawTab : TDrawTabEvent;
type TDrawTabEvent = procedure(Control: TCustomTabControl;
       TabIndex: Integer; const Rect: TRect; Active: Boolean) of object;
```

Здесь: `Control` — непосредственно компонент — список закладок; `TabIndex` — порядковый номер перерисовываемой в данный момент закладки; `Rect` — координаты прямоугольной области этой закладки; `Active` — флаг, информирующий об активности закладки.

Закладки *TTabSet* и *TDockTabSet*

Элементы управления `TTabSet` и `TDockTabSet` представляют собой упрощенную версию наборов закладок и сохранены на палитре компонентов Delphi в первую очередь для обеспечения обратной совместимости с прежними версиями среды проектирования. Основные характеристики компонентов представлены в табл. 26.1.

К особенностям элемента управления `TDockTabSet` следует отнести поддержку технологии перетаскивания элементов закладок между однотипными компонентами.

***Таблица 26.1.** Основные характеристики `TTabSet` и `TDockTabSet`*

Свойство/метод	Описание	
property `Tabs: TStrings;`	Список закладок	
property `TabIndex: Integer;`	Индекс активной закладки	
property `Style: TTabStyle;`	Стиль элемента управления	
property `TabPosition: TTabPosition;`	Расположение закладок	
property `TabHeight: Integer;`	Высота закладок	
function `ItemWidth(Index: Integer): Integer;`	Ширина элемента	
property `VisibleTabs: Integer;`	Число видимых закладок	
property `FirstIndex: Integer;`	Индекс первой видимой закладки	
property `OnChange: TTabChangeEvent;`	Событие смены активной закладки	
property `OnGetImageIndex: TTabGetImageEvent;`	Событие присвоения закладке индекса изображения	
property `OnDrawTab: TDrawTabEvent;`	Событие прорисовки закладки	
property `OnMeasureTab: TMeasureTabEvent;`	Применяется для расчета размера элемента закладки	
property `DestinationDockSite: TWinControl;`	Ссылка на компонент — получатель закладок	Только для `TDockTabSet`
property `OnTabAdded: TNotifyEvent;`	Событие добавления закладок	
property `OnTabRemoved: TNotifyEvent;`	Событие удаления закладок	

Блокнот *TPageControl*

Некоторое визуальное сходство между блокнотом `TPageControl` и набором закладок `TTabControl` объясняется общим предком — классом `TCustomTabControl` (рис. 26.1). Вместе с тем блокнот с закладками обладает существенным отличием — в нем каждой закладке соответствует отдельная страница — экземпляр класса `TTabSheet`. Страница может служить контейнером для других элементов управления (рис. 26.3).

Для того чтобы во время визуального проектирования добавить к блокноту новую страницу, щелкните по компоненту правой кнопкой мыши и в контекстном меню выберите пункт **New Page**. Для удаления ненужной страницы просто выделите ее мышью и нажмите клавишу <Del>.

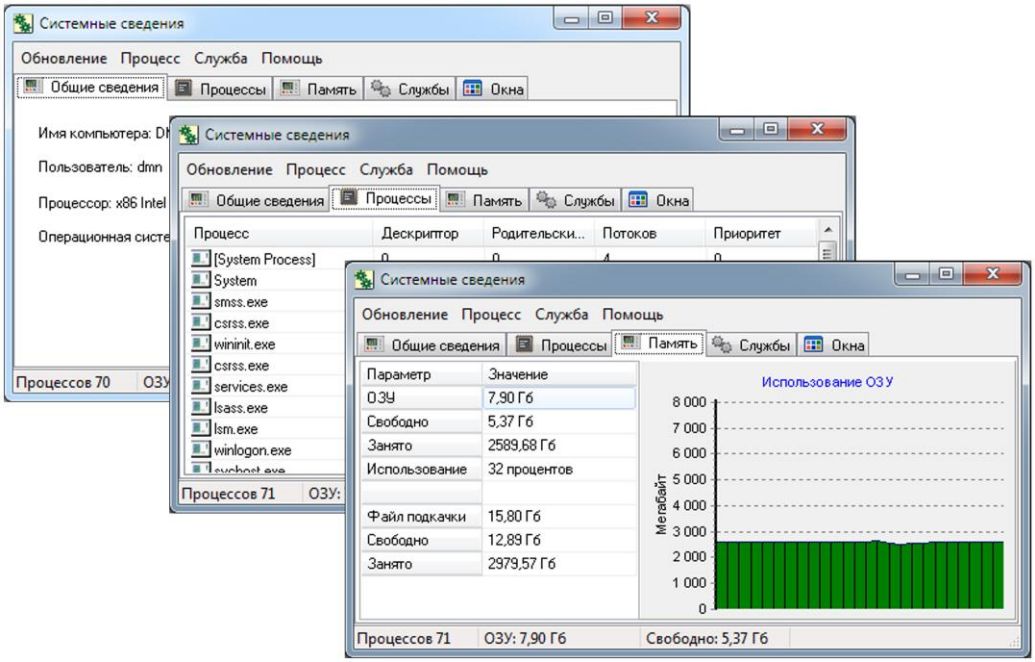


Рис. 26.3. Внешний вид приложения на основе блокнота `TPageControl`

> ***Внимание!***
>
> Основное отличие блокнота `TPageControl` от элементов управления, инкапсулирующих наборы закладок (например, `TTabControl`) в том, что каждая страница блокнота способна выступать в роли самостоятельного контейнера для других элементов управления.

Доступ к коллекции страниц блокнота предоставляет свойство

```
property Pages[Index: Integer] : TTabSheet; //только для чтения
```

Свойство не отражается в Инспекторе объектов и поэтому может быть обслужено только из кода программы.

Число страниц, входящих в состав блокнота, доступно благодаря свойству

```
property PageCount : Integer;
```

Ссылка на экземпляр активной (находящейся на переднем плане) страницы блокнота находится в свойстве

```
property ActivePage : TTabSheet;
```

Активной может быть только одна страница. Для изменения активной страницы пользователь должен щелкнуть по закладке кнопкой мыши. Индекс активной страницы мы обнаружим в свойстве

```
property ActivePageIndex: Integer;
```

Для пролистывания страниц блокнота можно воспользоваться методом

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

Направление перехода определяется параметром `GoForward` (`true` — вперед, `false` — назад).

К странице блокнота способна обратиться функция

```
function FindNextPage(CurPage: TTabSheet;
                      GoForward, CheckTabVisible: Boolean): TTabSheet;
```

Метод осуществляет перемещение по страницам относительно страницы `CurPage`. Назначение параметра `GoForward` — определять направление перехода. Параметр `CheckTabVisible` определяет, следует ли производить поиск среди невидимых страниц.

> ***ЗАМЕЧАНИЕ***
>
> Методы обработки событий в `TPageControl` аналогичны обработчикам `TTabControl`, т. к. унаследованы от одного и того же класса `TCustomTabControl`.

Страница блокнота *TTabSheet*

Страница блокнота `TTabControl` создается из специализированного класса `TTabSheet`. Хотя и экземпляр страницы вполне может существовать в памяти самостоятельно, но страница не может быть выведена на экран вне пределов своего блокнота — компонента `TPageControl`. Поэтому после вызова конструктора страницы необходимо явным образом обозначить принадлежность страницы к определенному блокноту `TPageControl`. Для этого предназначено свойство

```
property PageControl: TPageControl;
```

Благодаря тому, что страница блокнота является потомком класса `TWinControl`, она способна служить контейнером для других элементов управления. Листинг 26.1 демонстрирует способ динамического создания 5 страниц с размещением на каждой из страниц метки.

Листинг 26.1. Динамическое создание страниц блокнота

```
var i:Integer;
    TabSheet:TTabSheet;
begin
  for i:=0 to 4 do
  begin
    TabSheet:=TTabSheet.Create(PageControl1);//создаем страницу
    with TabSheet do
    begin
      Parent:=PageControl1;
      Caption:='Страница '+IntToStr(i);
      PageControl:=PageControl1;
    end;
    with TLabel.Create(TabSheet) do //разместим на странице метку
    begin
      Parent:=TabSheet;
      Caption:='Метка '+IntToStr(i);
    end;
  end;
end;
```

> ***ЗАМЕЧАНИЕ***
>
> Опираясь на свойство `PageControl`, мы получаем возможность обмениваться страницами между разными компонентами-блокнотами.

В списке страниц блокнота каждая страница характеризуется своим уникальным индексом

```
property PageIndex: Integer;
```

Индекс назначается в момент вставки страницы в блокнот, первая страница получает индекс 0, вторая — 1 и т. д. Не забывайте, что при перемещении или удалении страницы индексы переупорядочиваются.

Страницу блокнота можно сделать невидимой, для этого предназначено свойство

```
property TabVisible : Boolean;
```

При установке этого свойства в `false` данная страница скрывается с поверхности родительского блокнота. Внутри блокнота видимые страницы сортируются в соответствии с порядковым номером, описанным в свойстве

```
property TabIndex: Integer;
```

Первая видимая страница получает индекс 0, вторая — 1 и т. д. Индекс невидимой страницы устанавливается в –1.

При желании в области закладки страницы можно отобразить значок. Для этого подключите к владеющему страницей блокноту коллекцию картинок `TImageList` и в свойстве

```
property ImageIndex: TImageIndex;
```

укажите индекс соответствующей картинки.

Для того чтобы визуально выделить страницу блокнота, установите в `true` свойство

```
property Highlighted: Boolean;
```

С этого момента вывод страницы на экран будет осуществляться более ярким цветом.

ГЛАВА 27

Работа с датой и временем

В этой главе нам предстоит изучить компоненты VCL, связанные с обработкой таких специфичных данных, как дата и время. Но прежде чем мы приступим к поиску героев этой главы на палитре компонентов, нам предстоит получить ответ на самый важный вопрос: каким образом Delphi хранит значения даты и времени?

При выборе способа хранения значений даты и времени создатели Delphi воспользовались идеей, реализованной еще в старом языке Pascal. В основе идеи лежат два тезиса:

- во-первых, по своей сути дата и время являются двумя сторонами одной и той же медали, поэтому наиболее правильным решением станет создание такого типа данных, который будет обладать возможностью хранить значения даты и времени вместе;
- во-вторых, на взгляд создателей языка Pascal наиболее подходящей единицей измерения для даты могут выступать сутки (с чем трудно не согласиться), а для времени — доли суток. На первый взгляд предложенный способ хранения времени может показаться необычным, ведь традиционно время измеряется в миллисекундах (секундах, минутах, часах). Однако при более глубоком рассмотрении идеи выясняется, что мы приобретаем одно очень важное преимущество — возможность осуществлять элементарные арифметические действия над датой и временем.

Учтя все вышеперечисленное, разработчики Delphi для работы с датой и временем стали использовать обычный вещественный тип данных `Double`. Правда, для того чтобы акцентировать внимание рядового программиста на том, что все-таки речь идет именно о дате и времени, а не о рядовом действительном числе, в модуле `System` был объявлен базовый тип данных

```
type TDateTime = type Double;
```

Теперь поговорим подробнее об идее хранения значений даты и времени в `TDateTime`. Целая часть числа предназначена для запоминания даты, а значение после запятой — времени. Нулевому значению переменной соответствует время: 30.12.1899 г. 00:00:00:000 (в последней четверке нулей указываются часы, минуты, секунды, миллисекунды). Для того чтобы "сдвинуть" дату ровно на сутки вперед и получить значение 31.12.1899 г. 00:00:00:000, достаточно прибавить к переменной типа `TDateTime` одну единицу. Вычитание единицы повернет дату вспять, и мы окажемся в 29.12.1899 г. Значению 36526,0417 соответствует 01.01.2000 г. 01:00:00. Значение `TDateTime` на момент написания этих строк книги вы обнаружите на экранном снимке, представленном на рис. 27.1.

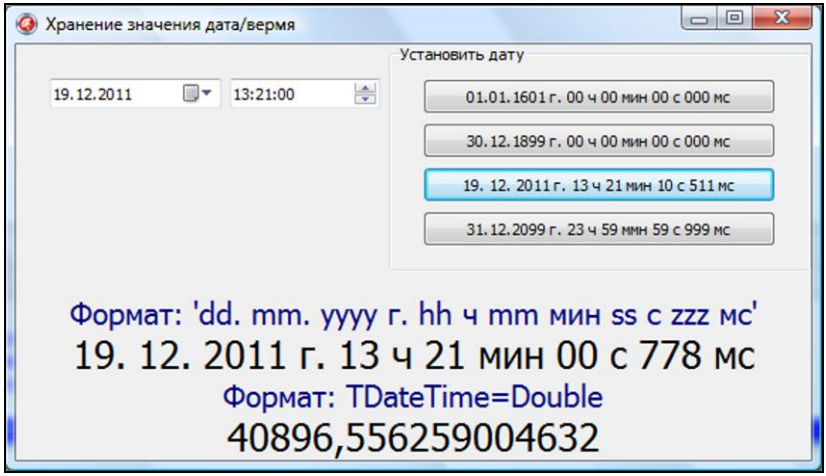


Рис. 27.1. Способ хранения значения даты/времени в `TDateTime`

Замечание

Верхний предел типа данных `TDateTime` соответствует значению 31.12.9999 г. 23:59:59:999. Кроме того, `TDateTime` допускает работу и с отрицательными значениями.

В дополнение к классу `TDateTime` в Delphi прилагаются производные типы данных:

```
type TDate = type TDateTime;//целая часть, только для хранения даты
type TTime = type TDateTime;//дробная часть, только для хранения времени
```

Тип данных `TDateTime` является основным, но далеко не единственным способом описания даты и времени. Например, для представления времени с точностью больше, чем `TDateTime` (до миллисекунд), предназначена запись

```
type TTimeStamp = record
     Time: Integer; //миллисекунды от полуночи
     Date: Integer; //количество дней, начиная с 01.01.0001 г.
end;
```

Отсчет времени, таймер *TTimer*

Невизуальный компонент `TTimer` мы обнаружим на странице **System** палитры компонентов. Этот компонент построен на базе класса `TComponent`. Таймер предназначен для генерации особых данных — сообщений об изменении времени. Для этой цели таймер с заданной периодичностью вызывает событие

```
property OnTimer: TNotifyEvent;
```

в котором следует описать реакцию нашей программы на изменение времени.

Ключевое свойство таймера

```
property Interval: Cardinal; //по умолчанию 1000 миллисекунд
```

определяет периодичность срабатывания таймера.

Таймер управляется единственным свойством

```
property Enabled: Boolean;
```

Установив `Enabled` в `true`, мы заставим компонент отсчитывать миллисекунды, перевод свойства в состояние `false` остановит таймер.

Расположив на поверхности формы метку `Label1` и таймер `Timer1`, мы с легкостью напишем программу Часы (листинг 27.1).

Листинг 27.1. Приложение Часы

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Timer1.Interval:=1000;
  Timer1.Enabled:=True;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption:=FormatDateTime('hh:nn:ss', Time);
end;
```

При создании формы устанавливается периодичность отсчета (1000 миллисекунд = 1 секунда) и производится включение таймера. Теперь с каждым вызовом события `OnTimer()` программа обновляет текст метки, передавая в него отформатированное значение системного времени.

Компоненты-календари, базовый класс *TCommonCalendar*

На странице **Win32** палитры компонентов Delphi давно обосновались два компонента, способные исполнять роль календарей. Это календарь `TMonthCalendar` и универсальный элемент управления `TDateTimePicker`, позволяющий осуществлять выбор не только даты, но и времени.

Оба наших компонента построены на фундаменте общего предка — классе `TCommonCalendar` (рис. 27.2). В `TCommonCalendar` объявлены общие ключевые свойства и методы, позволяющие его потомкам называться календарями.

Целевая функция календарей — разрешать пользователю производить выбор даты, поэтому большинство методов и свойств календарей направлено на обеспечение этой цели. Значение выбранной в календаре даты/времени мы обнаружим в свойстве

```
property DateTime: TDateTime;
```

Если нас интересует только дата, то воспользуемся услугами свойства

```
property Date: TDate;
```

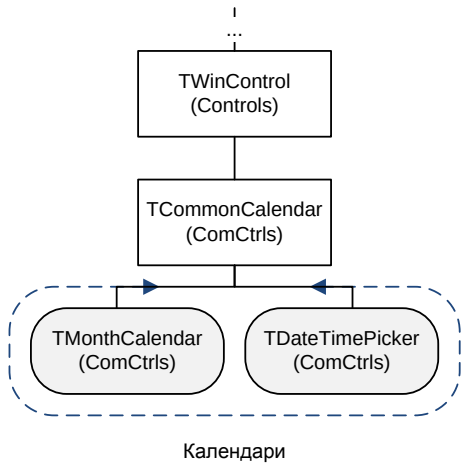


Рис. 27.2. Место календарей в иерархии VCL

Свойства `DateTime` и `Date` могут применяться и для установки текущей даты календаря из кода программы.

Программист имеет право ограничить диапазон доступных для выбора дат. Для этого предназначена пара свойств:

```
property MinDate: TDate;
property MaxDate: TDate;
```

Свойство `MinDate` определяет минимальную, а `MaxDate` — максимально возможную дату календаря. По умолчанию эти свойства пусты, что является признаком отсутствия ограничений на выбор даты.

В России трудовая неделя стартует с понедельника, в США и Англии — с воскресенья. Для того чтобы исключить путаницу, программисты Embarcadero предложили нам самостоятельно определиться, с какого дня должна начинаться неделя в наших программах. Для этого в праотце всех календарей объявлено свойство

```
property FirstDayOfWeek: TCalDayOfWeek;
type TCalDayOfWeek = (dowMonday, dowTuesday, dowWednesday, dowThursday, dowFriday,
dowSaturday, dowSunday, dowLocaleDefault);
```

Здесь: `dowMonday` — понедельник; `dowSunday` — воскресенье; `dowLocaleDefault` — день недели определяется автоматически локализацией версии Windows.

Установив в `true` свойство

```
property MultiSelect : Boolean; //по умолчанию false
```

мы разрешим пользователю в календаре выбирать одновременно некоторый диапазон дат. Выбор производится при нажатой клавише <Shift>. В этом случае первая выбранная дата диапазона окажется в свойстве `Data`, а последняя по очередности дата будет сохранена в свойстве

```
property EndDate: TDate;
```

Одновременно пользователь не сможет выбрать дней более, чем определено в свойстве

```
property MaxSelectRange: Integer; //по умолчанию 31
```

Ряд свойств существенно повышает наглядность представления данных календарем — влияют на его интерфейсную часть (табл. 27.1).

***Таблица 27.1.** Особенности оформления календаря*

Свойство	Описание
`property ShowToday: Boolean;`	Выводить внизу календаря сегодняшнюю дату
`property ShowTodayCircle: Boolean;`	Установив свойство в `true`, мы обяжем календарь выделять цветом сегодняшнюю дату
`property WeekNumbers: Boolean;`	Отображать (`true`) или нет (`false`) номера дней недели в левой колонке календаря
`property CalColors: TMonthCalColors;`	Многофункциональное свойство, устанавливающее цветовое оформление элементов календаря. Можно задать значения: `BackColor` — фон календаря; `MonthBackColor` — фон месяца; `TextColor` — цвет текста; `TitleBackColor` — фон заголовка; `TitleTextColor` — цвет текста заголовка; `TrailingTextColor` — цвет дат предыдущего и последующего месяцев

В базовом классе TCommonCalendar объявлены два специфичных события, вызываемые в момент смены месяца и решающие одну и ту же задачу — отображение утолщенным шрифтом определенных дней (например, праздничных).

```
property OnGetMonthInfo: TOnGetMonthInfoEvent;
type TOnGetMonthInfoEvent = procedure(Sender: TObject; Month: LongWord; var
MonthBoldInfo: LongWord) of object;
```

и

```
property OnGetMonthBoldInfo: TOnGetMonthBoldInfoEvent;
type TOnGetMonthBoldInfoEvent = procedure(Sender: TObject; Month: LongWord; Year:
LongWord; var MonthBoldInfo: LongWord) of object;
```

Разница между событиями всего в одном параметре. Так, в обеих версиях процедур есть параметры: Sender — ссылка на календарь; Month — номер месяца, который сейчас будет отображен; MonthBoldInfo определяет дни месяца, которые будут выделены утолщенным шрифтом. Наконец в событии OnGetMonthBoldInfo() появился еще один параметр, который позволяет контролировать год — Year.

Для инициализации параметра MonthBoldInfo требуется обратиться к специальному методу

```
procedure BoldDays(Days: array of LongWord; var MonthBoldInfo: LongWord);
```

В массив Days программист заносит интересующие его дни, результаты выполнения процедуры возвратятся в параметр MonthBoldInfo.

Листинг 27.2 демонстрирует способ оповещения календаря о праздничных днях.

Листинг 27.2. Пример обработки события OnGetMonthBoldInfo()

```
procedure TForm1.MonthCalendar1GetMonthBoldInfo(Sender: TObject; Month,
  Year: Cardinal; var MonthBoldInfo: Cardinal);
begin
  case Month of
    1: {праздники января}
          if Year>=2007 then MonthCalendar1.BoldDays(
                          [1,2,3, 4, 5, 6, 7,8,9], MonthBoldInfo)
          else {до 2007 года выходных дней было меньше}
             MonthCalendar1.BoldDays([1,2,7], MonthBoldInfo);
    2: MonthCalendar1.BoldDays([23], MonthBoldInfo);{праздники февраля}
    3: MonthCalendar1.BoldDays([8], MonthBoldInfo); {праздники марта}
    // и т. д.
  end;
end;
```

Календарь *TMonthCalendar*

Компонент TMonthCalendar целиком и полностью основывается на классе TCommonCalendar. В рамках исходного кода класса не объявлено ни одного нового свойства, несколько унаследованных методов подверглись незначительной переработке.

После размещения календаря на форме он по умолчанию начинает отображать текущий месяц. Для того чтобы элемент управления TMonthCalendar смог одновременно вывести

сведения о днях более одного месяца, достаточно увеличить его геометрические размеры, например, установив клиентский режим выравнивания по всей форме (рис. 27.3).

Замечание

Календарь `TMonthCalendar` может одновременно отображать от 1 до 12 месяцев года.

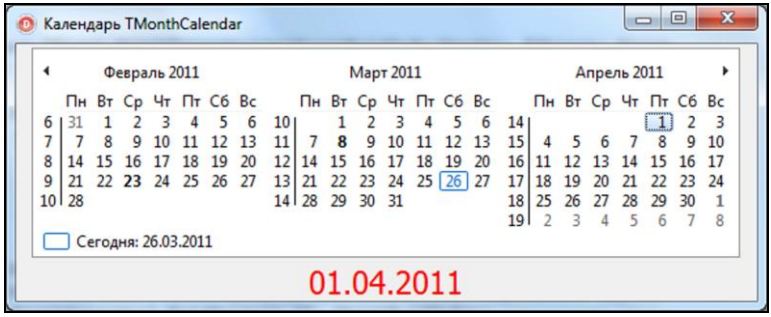


Рис. 27.3. Внешний вид календаря `TMonthCalendar`

Выбор даты/времени, компонент *TDateTimePicker*

Компонент `TDateTimePicker` обладает значительно большим набором возможностей, чем его "коллега" — обычный календарь. Первое важное отличие в том, что `TDateTimePicker` совмещает функционал строки ввода и все возможности обычного календаря, что позволяет пользователю вводить значение даты/времени с помощью клавиатуры. Второе отличие в том, что элемент управления буквально одним щелчком мыши превращается из календаря в строку выбора времени. Секрет превращения таится в свойстве

```
property Kind: TDateTimeKind;
type TDateTimeKind = (dtkDate, dtkTime); //по умолчанию dtkDate
```

По умолчанию свойство `Kind` установлено в состояние `dtkDate`. Это означает, что компонент предназначен для ввода даты. Значение даты может редактироваться непосредственно в строке ввода либо выбираться из связанного со строкой календаря. Если элемент управления установлен в состояние `dtkTime`, то он переключается в режим редактирования времени. С этого момента в строке ввода вместо даты выводится строка со временем.

Для удобства работы со временем компонент снабжен отдельным свойством

```
property Time: TTime;
```

Помимо этого, от предка `TCommonCalendar` унаследовано свойство `Date`, предназначенное для обслуживания даты.

Если вы допускаете, чтобы пользователь вводил значение даты (времени) с клавиатуры непосредственно в строку ввода, то проверьте, установлено ли в `true` свойство

```
property ParseInput: Boolean; //по умолчанию false
```

В этом случае в момент ввода пользователем даты (времени) генерируется событие `OnUserInput()`.

Особенности форматирования текста с датой (или временем) определяются в свойстве

```
property Format: String;
```

Листинг 27.3 демонстрирует порядок определения формата вывода даты, маска формата задается точно так же, как и для функции `FormatDateTime()` *(см. приложения 2 и 3)*.

Листинг 27.3. Определение порядка форматирования даты

```
DateTimePicker1.Kind:= dtkDate;
DateTimePicker1.Format:='DD/MMM/YYYY';
```

Если вы не хотите возиться со свойством `Format`, то более быстрый способ определения особенностей форматирования даты предоставит свойство

```
property DateFormat: TDTDateFormat; //варианты dfShort или dfLong
```

Значение `dfShort` соответствует сокращенной дате, например "24/03/12", `dfLong` — полной дате, например 24 марта 2012 г.

Для отображения в левой части строки ввода области для отметки флажком установите в `true` свойство

```
property ShowCheckbox: Boolean; //по умолчанию false
```

Установлен флажок или нет, знает свойство

```
property Checked: Boolean;
```

Основные события элемента управления представлены в табл. 27.2.

***Таблица 27.2.** Основные события `TDateTimePicker`*

Событие	Описание
`property OnChange: TNotifyEvent;`	Сопровождает любое изменение значения даты/времени
`property OnUserInput: TDTParseInputEvent;` `type TDTParseInputEvent = procedure(Sender: TObject; const UserString: string; var DateAndTime: TDateTime; var AllowChange: Boolean) of object;`	Генерируется при ручном вводе даты/времени пользователем. `Sender` — ссылка на компонент, `UserString` — текстовая строка, введенная пользователем, `DateAndTime` — новое значение даты (времени). Программист дает разрешение на изменение даты (времени), передав значение `true` в параметр `AllowChange`
`property OnDropDown: TNotifyEvent;`	Вызывается в момент показа панельки с календарем
`property OnCloseUp : TNotifyEvent;`	Панелька с календарем скрывается
`property OnGetMonthInfo: TOnGetMonthInfoEvent;` `property OnGetMonthBoldInfo: TOnGetMonthBoldInfoEvent;`	События, позволяющие изменить начертание отмеченных дат календаря

ГЛАВА 28

Диалоговые окна

Любая, претендующая на звание интерактивной, программа должна обладать возможностью общаться с пользователем: "Вот так откровение, — удивится читатель. — Для этого в распоряжении программы есть меню, команды и другие элементы управления!" Это действительно так, однако меню, команды и другие элементы пользовательского интерфейса просто передают приложению требования пользователя, в которых он определяет порядок поведения нашей программы. Но любая сложная система (в том числе и программный продукт) нуждается и в обратной связи со своим оператором. Как правило, этот контакт необходим для уведомления пользователя о завершении определенной операции или для подтверждения какой-либо команды.

В большинстве случаев для общения с оператором компьютера приложение выводит на экран диалоговые окна. Диалоговое окно обладает весьма навязчивым характером, и для того чтобы гарантированно достучаться до пользователя, в подавляющем числе случаев оно выводится на экран в модальном режиме поверх всех окон приложения. Такой на первый взгляд назойливый способ общения гарантирует, что даже самый невнимательный пользователь уделит диалоговому окну хотя бы немного своего драгоценного времени, тем более что пока пользователь не закроет окно, он не сможет вернуться к работе с программой.

В этой главе мы обсудим несколько категорий диалоговых окон, от простейших окон сообщений и окон ввода текстовой информации до сложных диалогов доступа к файлам, поиска текста и настройки печатающего устройства.

Окна вывода сообщений

Самое простое, что может сделать программа — оповестить пользователя о каком-либо событии текстовой строкой. Окно с текстовым сообщением выводится процедурой

```
procedure ShowMessage(const Msg: string);
```

В результате ровно в центре рабочего стола появится окно с текстом, который вы ранее передали в параметр `Msg`.

Более широкими возможностями может похвастаться процедура

```
procedure ShowMessagePos(const Msg: string; X, Y: Integer);
```

Теперь мы можем управлять местом вывода сообщения. Для этой цели в распоряжение программиста передаются параметры (`X`, `Y`), задающие координаты левого верхнего угла окна.

Если текстовое сообщение окна достаточно сложное, например, содержит разнотипные значения, то следует обратиться к процедуре

```
procedure ShowMessageFmt(const Msg: string; Params: array of const);
```

Форматирование осуществляется в соответствии с правилами форматирования строк, применяемыми в функции `Format()`, в первый параметр передается шаблон сообщения с форматирующими символами, во второй параметр направляется массив с данными *(см. приложение 2)*. Пример работы с процедурой предложен в листинге 28.1.

Листинг 28.1. Пример вывода сообщения со сложным форматированием

```
resourcestring  Msg='Сложное сообщение содержит:'+
          #13+'текст — %s'+#13+'целое — %d'+#13+'вещественное — %f';
var s:string;
    i:integer;
    r:real;
begin
  s:='строка текста';
  i:=555;
  r:=111/13;
  ShowMessageFmt(Msg,[s,i,r]);
end;
```

В результате выполнения листинга на экране компьютера появится диалоговое окно, представленное на рис. 28.1.

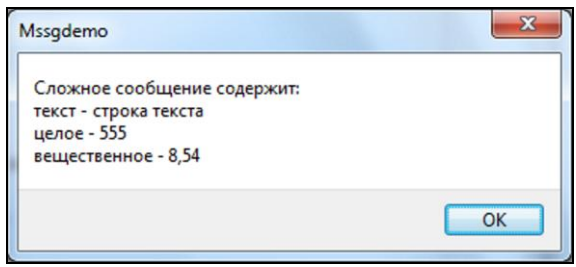


Рис. 28.1. Пример окна сообщения, полученного с помощью процедуры `ShowMessageFmt()`

ЗАМЕЧАНИЕ

К сожалению, при выводе текстового сообщения в заголовок окна диалога Delphi автоматически подставляет имя исполняемого файла приложения, что несколько снижает информативность окна.

Окна выбора действия

Все рассмотренные ранее процедуры `ShowMessage()`, `ShowMessagePos()`, `ShowMessageFmt()` предназначены лишь для уведомления пользователя о том или ином событии и никак не влияют на дальнейшую логику выполнения приложения. В том случае, когда операция требует, чтобы пользователь подтвердил ее выполнение, следует искать помощи у функции

`MessageDlg()`. В Delphi предусмотрены две перегружаемые версии этой функции, мы рассмотрим самую сложную:

```
function MessageDlg(const Msg: string; DlgType: TMsgDlgType;
              Buttons: TMsgDlgButtons; HelpCtx: Longint;
              DefaultButton: TMsgDlgBtn): Integer; overload;
```

В результате обращения к функции на экран выводится диалоговое окно, представленное на рис. 28.2.

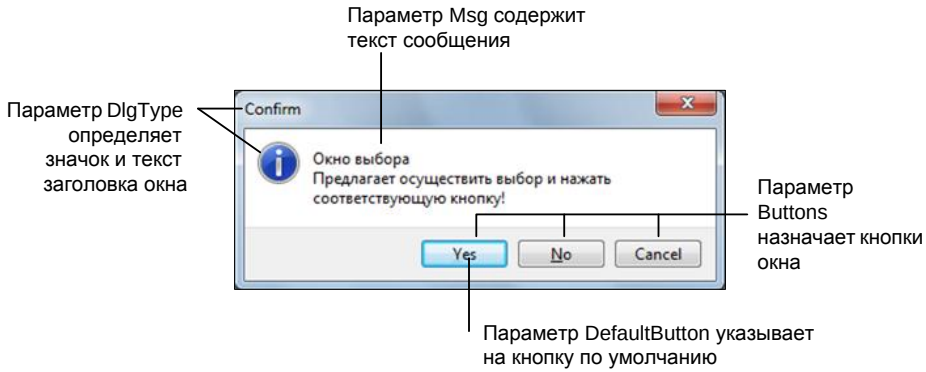


Рис. 28.2. Окно выбора действия, вызванное функцией `MessageDlg()`

В параметре `Msg` задается текст сообщения. Параметр `DlgType` определяет внешний вид диалогового окна в соответствии с типом `TMsgDlgType` (табл. 28.1).

Таблица 28.1. *Оформление окна выбора* `TMsgDlgType`

Значение	Особенности оформления окна
`mtWarning`	Тревожное оповещение, восклицательный знак в желтом треугольнике
`mtError`	Оповещение об ошибке, крест в красном кругу
`mtInformation`	Информационное сообщение, символ "i" в синем кругу
`mtConfirmation`	Запрос, зеленый знак вопроса
`mtCustom`	Сообщение без рисунка, заголовок окна содержит название проекта

Любое диалоговое окно как минимум содержит хотя бы одну кнопку, нажав которую пользователь известит программу, что он ознакомился с сообщением. Еще чаще в окне диалога располагается несколько кнопок, нажатие которых определяет дальнейшее поведение программы. Например, окно, спрашивающее у пользователя подтверждение на удаление файла, должно содержать кнопки **Да** (Yes) и **Нет** (No). Нажатие кнопки **Да** подтверждает операцию, кнопка **Нет** отвергнет. В код программы диалоговое окно возвращает модальный результат, соответствующий нажатой пользователем кнопке: кнопка **Да** — константа `mrYes`, кнопка **Нет** — `mrNo`, и т. д. Задачей программиста является обработка этого результата. Какие именно кнопки будут размещены в нижней части окна, определяет параметр `Buttons:TMsgDlgButtons`.

```
type TMsgDlgBtn = (mbYes, mbNo, mbOK, mbCancel, mbAbort, mbRetry,
                   mbIgnore, mbAll, mnNoToAll, mbYesToAll, mbHelp);
```

Для наиболее часто встречающихся комбинаций кнопок подготовлены константы

```
const  mbYesNoCancel = [mbYes, mbNo, mbCancel];
       mbOKCancel    = [mbOK, mbCancel];
       mbAbortRetryIgnore = [mbAbort, mbRetry, mbIgnore];
```

Параметр функции `HelpCtx` предназначен для взаимодействия со справкой приложения, в нем задается индекс соответствующей диалоговому окну страницы справки. Если справка отсутствует, то в параметр передается –1. Параметр `DefaultButton` определяет, какая из кнопок окна будет находиться в фокусе ввода в момент вывода окна на экран.

В листинге 28.2 приведен пример процедуры, удаляющей файл с именем `FileName`. Файл исчезнет с жесткого диска только в том случае, если пользователь подтвердит удаление нажатием кнопки **Да**.

Листинг 28.2. Пример вывода окна выбора действия

```
procedure FileDelete(FileName : string);
begin
  if MessageDlg('Удалить файл '+ExtractFileName(FileName)+' ?',
              mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    DeleteFile(FileName);
end;
```

По умолчанию диалоговое окно выбора позиционируется в центре экрана. Для того чтобы нарисовать окно в альтернативном месте, используйте функцию

```
function MessageDlgPos(const Msg: string; DlgType: TMsgDlgType;
       Buttons: TMsgDlgButtons; HelpCtx: Longint; X, Y: Integer): Word;
```

Все параметры метода нам уже знакомы, единственная новость — координаты левого верхнего угла окна определяются аргументами `X` и `Y`.

Еще одна похожая функция

```
function MessageDlgPosHelp(const Msg: string; DlgType: TMsgDlgType;
         Buttons: TMsgDlgButtons; HelpCtx: Longint; X, Y: Integer;
         const HelpFileName: string; DefaultButton: TMsgDlgBtn): Integer;
```

позволяет динамически подключать файл со справкой прямо к окну диалога в момент его создания.

Завершая разговор об окнах выбора действия, рассмотрим функцию

```
function TaskMessageDlg(const Title, Msg: string; DlgType: TMsgDlgType;
                        Buttons: TMsgDlgButtons; HelpCtx: Longint;
                        DefaultButton: TMsgDlgBtn): Integer;
```

В функции предусмотрен отдельный параметр `Title`, определяющий дополнительный заголовок сообщения, все остальные аргументы функции нам уже знакомы. Упомянем еще тот факт, что в наше распоряжение предоставлены: функция `TaskMessageDlgPos()`, управляющая местом вывода окна, и функция `TaskMessageDlgPosHelp()`, подключающая к окну файл со справкой.

Создание многоразового окна выбора действия

Если вы намерены создать собственное неповторимое диалоговое окно, то наиболее просто решить эту задачу, обратившись к функции

```
function CreateMessageDialog(const Msg: string;
               DlgType: TMsgDlgType; Buttons: TMsgDlgButtons): TForm;
```

Функция не выводит диалоговое окно на экран, а просто создает новую форму, которая может быть многократно задействована в коде приложения.

Ответственность за показ и уничтожение формы диалога ложится на плечи программиста (листинг 28.3).

Листинг 28.3. Создание пользовательской формы диалога

```
var NewDlg : TForm;
begin
  NewDlg:=CreateMessageDialog('Ошибка!', mtError, [mbYes, mbNo]);
  if NewDlg.ShowModal=idOK then ...;
  NewDlg.Release;
end;
```

Окна ввода данных

Очень часто для работы приложения недостаточно получать односложные ответы пользователя: "Да", "Нет" или "Отмена". Программе могут потребоваться фамилия пользователя, адрес электронной почты, пароль или другие данные. В таком случае окажется полезной функция

```
function InputBox(const ACaption, APrompt, ADefault: string): string;
```

В результате вызова `InputBox()` создается форма, содержащая строку ввода и две кнопки: **OK** и **Отмена** (Cancel). В параметре `ACaption` определяется заголовок диалогового окна, `APrompt` содержит надпись над строкой ввода, в `ADefault` передается значение по умолчанию. Пример работы с функцией предложен в листинге 28.4.

Листинг 28.4. Демонстрация функции `InputBox()`

```
var s: string;
begin
  s:= InputBox('Создание файла', 'Имя файла', 'Новый файл.txt);
  //...
```

Еще более удобная (на мой взгляд) функция

```
function InputQuery(const ACaption, APrompt: string;
                                  var Value: string ): Boolean;
```

возвращает не только содержимое текстовой строки (для этого предназначен параметр `Value`), а еще и логическое значение, соответствующее нажатой пользователем кнопки: **OK** — `true`, **Отмена** (Cancel) — `false`. Эта особенность функции позволяет улучшать код наших программ (листинг 28.5).

Листинг 28.5. Демонстрация функции `InputQuery()`

```
var s: string;
begin
  s := 'Новый пользователь';
  if InputQuery('Регистрация', 'Фамилия', s)=true then
    {действия, если нажата кнопка OK}
  else {действия, если нажата кнопка Cancel};
end;
```

Окна выбора файлов и папок

Одна из наиболее востребованных областей применения диалоговых окон связана с процессом выбора пользователем файлов и каталогов.

Для вызова стандартного диалогового окна открытия (сохранения) файла можно воспользоваться помощью функции

```
function PromptForFileName(var AFileName: string;
           const AFilter: string = ''; const ADefaultExt: string = '';
           const ATitle: string = '';  const AInitialDir: string = '';
           SaveDialog: Boolean = False): Boolean;
```

В простейшей нотации в функцию необходимо передать лишь одну переменную `AFileName`, в которой (после выполнения функции) окажется имя выбранного пользователем файла. В более сложной нотации можно заполнить параметр `AFilter` сведениями о фильтре, ограничивающем перечень доступных файлов. Строка фильтра состоит из двух частей, разделенных вертикальной чертой. В первой половине строки содержится текстовое пояснение о назначении фильтра, а во второй — собственно фильтр. Например, для определения фильтра для открытия текстовых файлов можно воспользоваться следующей строкой:

```
AFilter:='Текстовый файл (*.txt)|*.txt';
```

Параметр `ADefaultExt` хранит расширение имени файла по умолчанию. В аргумент `ATitle` следует передать заголовок, который будет отображен в окне диалога. Параметр `AInitialDir` определяет имя папки, которая по умолчанию будет открыта диалогом. Заключительный параметр `SaveDialog` конкретизирует, какую разновидность диалогового окна мы рассчитываем увидеть: открытия файла (состояние `false`) или сохранения файла (состояние `true`).

Хотя основной герой этой главы — модуль `Dialogs`, сейчас мы сделаем исключение и ненадолго переключим наше внимание на модуль `FileCtrl`. Именно здесь объявлена перегружаемая функция `SelectDirectory()`, вызывающая диалоговое окно выбора папки или файла. Передавая те или иные параметры, программист получит два различных по внешнему виду диалога. Наиболее востребован вариант функции, вызывающий диалоговое окно в духе современного Проводника Windows (рис. 28.3):

```
function SelectDirectory(const Caption: string; const Root: WideString;
                         var Directory: string;
          [Options: TSelectDirExtOpts]; [Parent: TWinControl]): Boolean;
```

В параметре `Caption` задается заголовок окна. Параметр `Root` ограничивает область выбора папок. Так, благодаря ему программист сможет задать корневую папку, покинуть которую пользователь не сможет (если в этом нет необходимости — передавайте в параметр пустую

строку). Особенность функции в том, что она возвращает два значения: если пользователь выбрал каталог и нажал кнопку **OK**, то в выходной параметр `Directory` передается полный путь к выбранному каталогу и функция возвращает `true`; нажатие кнопки **Отмена** заставит функцию вернуть значение `false`. Последние два аргумента функции необязательны. Параметр `Options` определяет дополнительные опции окна:

```
type TSelectDirExtOpt = (sdNewFolder, //разрешено создавать папку
     sdShowEdit, //выводится строка, содержащая путь к выбранному объекту
     sdShowShares,   //отображает разделяемые ресурсы
     sdNewUI,        //использование современного интерфейса
     sdShowFiles,    //включить показ файлов
     sdValidateDir); //проверка корректности имени папки/файла
```

В заключительный параметр `Parent` можно передавать ссылку диалога на родительское окно.

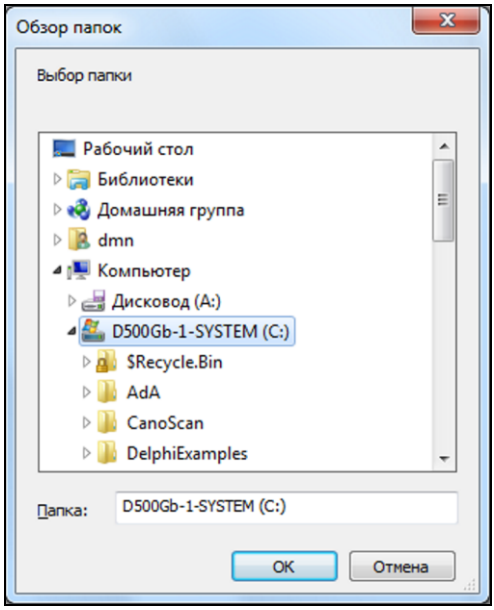


Рис. 28.3. Окно выбора папки с помощью `SelectDirectory()`

Пример работы с функцией представлен в листинге 28.6.

Листинг 28.6. Демонстрация функции `SelectDirectory()`

```
var s:string;
begin
  if SelectDirectory('Выбор папки','',s,[sdShowEdit,sdNewUI])
  then Edit1.Text:=s;
end;
```

Компоненты-диалоги

Открыв страницу **Dialogs** палитры компонентов Delphi, вы обнаружите больше десятка компонентов-диалогов, отвечающих за самые разные аспекты взаимодействия приложения и пользователя.

- Наиболее внушительная группа компонентов вызывает диалоговые окна открытия/сохранения файлов. Эта группа насчитывает полдюжины классов: `TOpenDialog`, `TSaveDialog`, `TOpenPictureDialog`, `TSavePictureDialog`, `TOpenTextFileDialog`, `TSaveTextFileDialog`. Диалоги `TOpen...` открывают файлы, а диалоги `TSave...` — сохраняют файлы.
- Компоненты `TFineDialog` и `TReplaceDialog` окажутся весьма полезными в приложениях, обрабатывающих текстовые данные. Благодаря этим диалогам программист сможет вызвать окна поиска фрагмента текста и замены найденного фрагмента на новый текст.
- Компонент `TFontDialog` вызовет на экран стандартный диалог настройки параметров шрифта.
- Помощь в выборе подходящего оттенка цвета окажет диалог `TColorDialog`.
- Настройку размера, полей и ориентации страницы документа осуществит компонент `TPageSetupDialog`.
- Диалоги `TPrinterSetupDialog` и `TPrintDialog` соответственно помогут подготовить принтер к печати и отправить задание на печать.

Прежде чем мы приступим к изучению каждого из представленных на рис. 28.4 диалогов, рассмотрим их наиболее важные, унаследованные от опорного класса `TCommonDialog` родовые черты. Основным методом класса `TCommonDialog` считается

```
function Execute: Boolean; virtual; abstract;
```

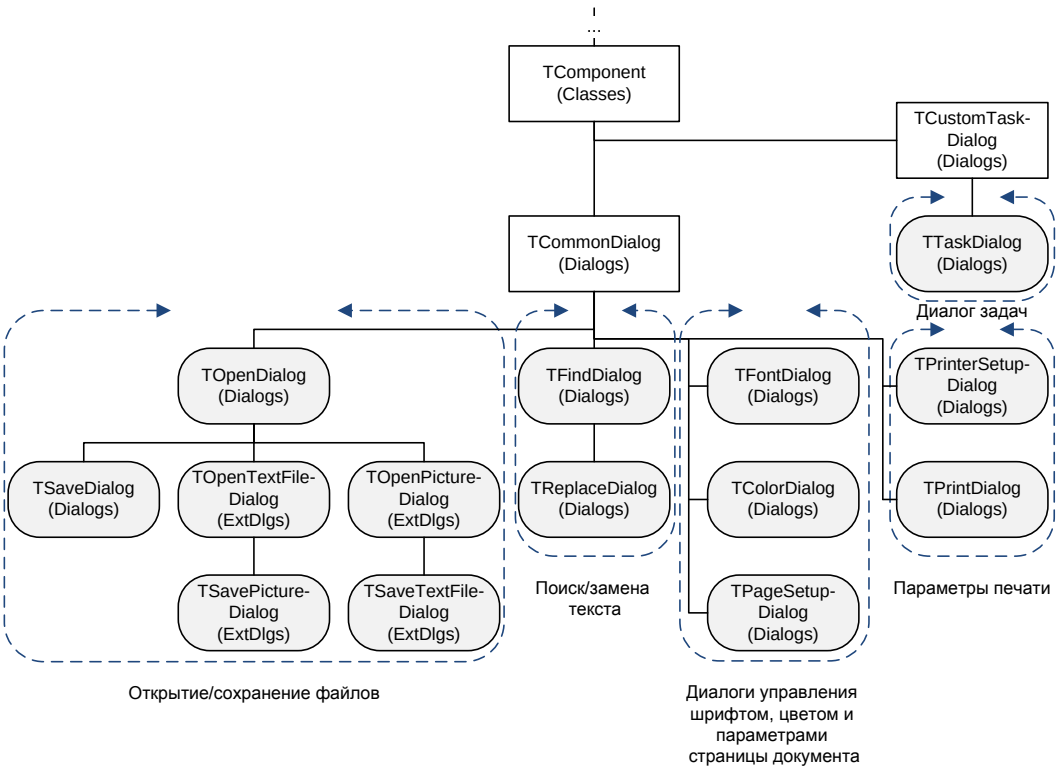


Рис. 28.4. Место компонентов-диалогов в иерархии классов VCL

Вызов потомками TCommonDialog функции Execute() (за исключением TFindDialog и TReplaceDialog) приводит к выводу на экран стандартного диалогового окна, работая с которым пользователь взаимодействует со своей программой или с Windows. При нажатии кнопки **OK** в окне диалога функция возвращает значение true, в противном случае — false.

В классе описаны всего два обработчика событий

```
property OnShow: TNotifyEvent;
property OnClose: TNotifyEvent;
```

вызываемых соответственно в момент показа и закрытия диалогового окна.

Диалоги открытия и сохранения файлов

В перечне компонентов наиболее широкое представительство получили диалоги, предназначенные для поддержки операций открытия и сохранения файлов. На странице **Dialogs** палитры Delphi вы обнаружите шесть компонентов, отвечающих за вызов диалогового окна открытия (диалоги TOpen...) и сохранения (диалоги TSave...). Наиболее универсальны компоненты TOpenDialog и TSaveDialog. Они способны работать с любым типом файла. Оставшиеся две пары компонентов имеют узкую специализацию. Так, за обслуживание файлов изображений отвечают TOpenPictureDialog и TSavePictureDialog, за доступ к текстовым файлам — TOpenTextFileDialog и TSaveTextFileDialog.

> ***ЗАМЕЧАНИЕ***
>
> Не стоит понимать названия диалогов открытия и сохранения файлов буквально. Все эти компоненты всего лишь позволяют пользователю выбрать имя файла для его открытия (сохранения), саму программную логику открытия (сохранения) файла следует реализовывать самостоятельно.

Универсальные диалоги *TOpenDialog* и *TSaveDialog*

Для вызова стандартных диалоговых окон Windows, осуществляющих выбор имени файла для обеспечения дальнейшего чтения или записи, предназначены компоненты TOpenDialog и TSaveDialog.

При описании диалога открытия или сохранения файла первым действием программиста должно быть определение ограничений на имя допустимых файлов. При отображении диалоговое окно на основе фильтра

```
property Filter: string;
```

произведет отбор только необходимых пользователю файлов. Щелчок по кнопке с многоточием напротив названия свойства в Инспекторе объектов выведет на экран окно **Filter Editor** (Редактор фильтра), позволяющее нам указать приемлемые для нас файлы (рис. 28.5). Самая общая маска включает две звездочки, разделенные точкой — *.*, этот фильтр допускает все файлы. Все остальные вариации масок зависят от назначения нашего приложения.

Фильтр может быть настроен и во время выполнения программы. Допустим, для приложения, повторяющего функционал блокнота, фильтр диалогов открытия и сохранения файлов можно подготовить в момент создания главной формы приложения (листинг 28.7).

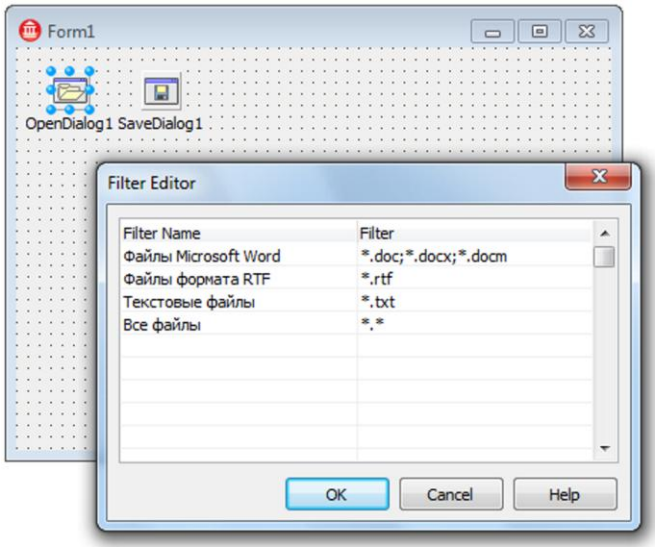


Рис. 28.5. Окно редактора фильтра диалогов `TOpenDialog` и `TSaveDialog`

Листинг 28.7. Настройка фильтра диалога открытия и сохранения файла

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  OpenDialog1.Filter:='Текстовые файлы|*.txt)';
  OpenDialog1.DefaultExt:='txt';
  SaveDialog1.Filter:=OpenDialog1.Filter;
  SaveDialog1.DefaultExt:= OpenDialog1.DefaultExt;
end;
```

Обратите внимание на то, что текстовая строка фильтра включает два раздела: текстовое описание фильтра и допустимое расширение в имени файла, а между разделами устанавливается специальный символ `|`.

Допустимо определение сразу нескольких типов файлов в одной строке фильтра:

```
OpenDialog1.Filter := 'Файлы Microsoft Word|*.doc;*.docx;*.docm';
```

Если требуется программным образом описать несколько строк фильтра, то отделяйте строку от строки все тем же символом-разделителем — вертикальной чертой (`|`):

```
OpenDialog1.Filter:='Текстовые файлы|*.txt|Все файлы|*.*)';
```

В листинге 28.7 упоминается еще одно свойство диалога:

```
property DefaultExt: string;
```

Предполагается, что при вводе имени файла в окне диалога пользователь должен ввести не только имя, но и расширение имени файла. В случае если этого не было сделано, то диалог подставляет расширение по умолчанию, заранее определяемое в свойстве `DefaultExt`.

> ***Внимание!***
> При вводе расширения имени по умолчанию помните, что оно включает только символы расширения, точку-разделитель указывать не следует.

Если используемый фильтр — многострочный, то следует определить, какая из строк фильтра станет использоваться по умолчанию при появлении диалогового окна. Индекс этой строки передается в свойство

```
property FilterIndex: Integer;
```

Для определения имени папки, открываемой в момент первого запуска диалога, воспользуйтесь свойством

```
property InitialDir: string;
```

При повторных запусках выбор каталога зависит от флага `ofNoChangeDir` в свойстве `Options` диалогового окна (табл. 28.2).

Для совместимости с ранними версиями Delphi в сохранены свойства

```
property FileEditStyle: TFileEditStyle; //fsEdit или fsComboBox
```

и

```
property HistoryList: TStrings;
```

Первое из них задает вид поля ввода имени файла — обычная строка ввода или комбинированный список. Второе свойство позволяет хранить историю о предыдущих прочитанных/записанных файлах. Оно играет роль при условии, что `FileEditStyle= fsComboBox`. В этом случае в строках комбинированного списка будет отображаться содержимое `HistoryList`.

Текст, подлежащий выводу в заголовке диалога, определяется в свойстве

```
property Title: string;
```

Наиболее широкий спектр по настройке внешнего вида диалогового окна предоставляет свойство (табл. 28.2)

```
property Options: TOpenOptions;
```

Таблица 28.2. *Основные опции диалогов открытия и сохранения файлов*

Значение	Описание
`ofReadOnly`	Открывает окно в режиме "только для чтения"
`ofHideReadOnly`	Скрывает флажок `ReadOnly`
`ofOverwritePrompt`	Играет роль в диалогах записи файлов, запрашивает разрешение на перезапись при совпадении имени сохраняемого и существующего файлов
`ofNoChangeDir`	Если флаг установлен, то при повторных запусках диалога он открывает папку, определенную в свойстве `InitialDir`. Иначе будет открыт каталог, с которым велась работа в последний раз
`ofShowHelp`	Дополняет диалог кнопкой помощи
`ofNoValidate`	Отключает контроль ввода недопустимого символа в имя файла
`ofAllowMultiSelect`	Допускает одновременный выбор нескольких фалов
`ofExtensionDifferent`	Наличие флага обычно контролируется программистом во время выполнения приложения. Он автоматически устанавливается приложением в случае, когда расширение выбранного файла отличается от заданного в свойстве `DefaultExt`

Таблица 28.2 *(окончание)*

Значение	Описание
ofPathMustExist	Вызывает сообщение об ошибке, если пользователь указал неверный путь к файлу
ofFileMustExist	Используется только в диалогах открытия файла. Вызывает сообщение об ошибке, если пользователь указал имя несуществующего файла
ofCreatePrompt	Работает совместно с ofFileMustExist, запросит подтверждение на создание несуществующего файла
ofShareAware	Разрешает совместный доступ к одному и тому же файлу. Если флаг отключен, то запрещает открывать файл, уже используемый другим процессом
ofNoReadOnlyReturn	Инициирует сообщение об ошибке при попытке обратиться к файлу с атрибутом "только для чтения"
ofNoTestFileCreate	Опция применяется при работе с разделяемым сетевым ресурсом, в котором пользователь обладает правом создания новых файлов, но не может модифицировать существующие. Флаг отключает проверку возможности записи в сетевом ресурсе
ofNoNetworkButton	Скрывает кнопку доступа к сетевому ресурсу (используется только совместно с флагом ofOldStyleDialog)
ofNoLongNames	Показывает файлы с форматом имени 8.3 (используется только совместно с флагом ofOldStyleDialog)
ofOldStyleDialog	Флаг обратной совместимости, создает диалоговое окно старого стиля
ofNoDereferenceLinks	Определяет порядок взаимодействия с ярлыками файлов. Если флаг отключен, то диалог возвращает реальное имя файла, определенное в ярлыке (файл с расширением lnk). Иначе диалог возвращает имя ярлыка
ofEnableIncludeNotify	Работает с Windows 2000 и более поздними версиями, используется для активизации обработчика события OnIncludeItem()
ofEnableSizing	Разрешает изменять размеры диалогового окна
ofDontAddToRecent	Пока не используется, в будущих версиях Delphi будет управлять добавлением ссылки на файл в список недавно использованных документов
ofForceShowHidden	Пока не используется, в будущих версиях Delphi будет принудительно включать показ скрытых файлов в окне диалога

У диалогов имеется еще одно свойство, управляющее дополнительными опциями:

```
property OptionsEx: TOpenOptionsEx;//по умолчанию []
```

Пока в TOpenOptionEx возможных вариантов прямо скажем не густо, доступна единственная опция ofExNoPlacesBar. Задача флага ofExNoPlacesBar — отключение панели быстрого доступа к рабочему столу, дискам компьютера и т. д.

Стоит напомнить, что фундаментальным методом диалогов открытия и сохранения файла является функция Execute(). Она осуществляет вызов диалогового окна, в котором пользователь производит выбор файла для открытия или указывает имя сохраняемого файла. Если после работы с диалогом была нажата кнопка **OK**, метод вернет true. После успешного вызова диалогового окна имя выбранного файла окажется в свойстве

```
property FileName: TFileName;
```

Допустим, что мы разрабатываем проект текстового редактора. На главной форме проекта frmMain расположены диалоги открытия и сохранения файлов, редактор многострочного текста Memo1:TMemo. За вызов диалога открытия файла отвечает событие-щелчок по кнопке btnOpen, диалог сохранения вызывается щелчком по кнопке btnSave (листинг 28.8).

Листинг 28.8. Управление диалогами открытия и сохранения файла

```
procedure TfrmMain.btnOpenClick(Sender: TObject); //открытие файла
begin
  if OpenDialog1.Execute then
  begin
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
    frmMain.Caption:=ExtractFileName(OpenDialog1.FileName);
  end;
end;

procedure TfrmMain.btnSaveClick(Sender: TObject); //сохранение файла
var fName:string;
begin
  if SaveDialog1.Execute then
  begin
    fName:=SaveDialog1.FileName;
    if fName<>'' then Memo1.Lines.SaveToFile(fName);
    frmMain.Caption:=ExtractFileName(fName);
  end;
end;
```

Щелчок по кнопке btnOpen вызывает метод Execute() диалога открытия файла, в результате на экране вашего компьютера отобразится окно, позволяющее пользователю выбрать текстовый файл (рис. 28.6).

> ***Внимание!***
>
> Чтобы увидеть открытое окно диалога, совсем необязательно писать программный код. Значительно проще во время визуального проектирования дважды щелкнуть левой кнопкой мыши по компоненту.

Теперь уделим немного времени обработке событий в компонентах открытия и сохранения файла. Оба компонента унаследовали события OnShow() и OnClose() от своего предка — класса TComonDialog. Помимо этого во всех диалогах имеется ряд вспомогательных событий (табл. 28.3).

Таблица 28.3. *События диалогов открытия и сохранения файлов*

Событие	Описание
`property OnFolderChange: TNotifyEvent;`	Происходит при открытии или закрытии папки в диалоговом окне
`property OnSelectionChange: TNotifyEvent;`	Возникает при выборе пользователем нового файла в списке файлов, применении нового фильтра, создании новой папки

Таблица 28.3 *(окончание)*

Событие	Описание
`property OnTypeChange: TNotifyEvent;`	Вызывается при установке пользователем нового фильтра
`property OnCanClose: TCloseQueryEvent;` `type TCloseQueryEvent = procedure(Sender: TObject; var CanClose: Boolean) of object;`	Происходит при попытке закрыть диалоговое окно (без отмены, т. е. без нажатия кнопки **Отмена** (Cancel)), параметр `CanClose` разрешает (`true`) или запрещает (`false`) закрытие окна
`property OnIncludeItem: TIncludeItemEvent;` `type TIncludeItemEvent = procedure( const OFN:TOFNotifyEx; var Include: Boolean) of object;`	Событие используется для организации дополнительной программной фильтрации списка папок. Оно возникает каждый раз перед тем, как к списку файлов окна добавляется новый элемент. Параметр `OFN` является особой структурой, описывающей папку в рамках интерфейса `IShellFolder`. Управляя параметром `Include`, программист разрешает (`true`) или запрещает (`false`) добавлять новый элемент в список

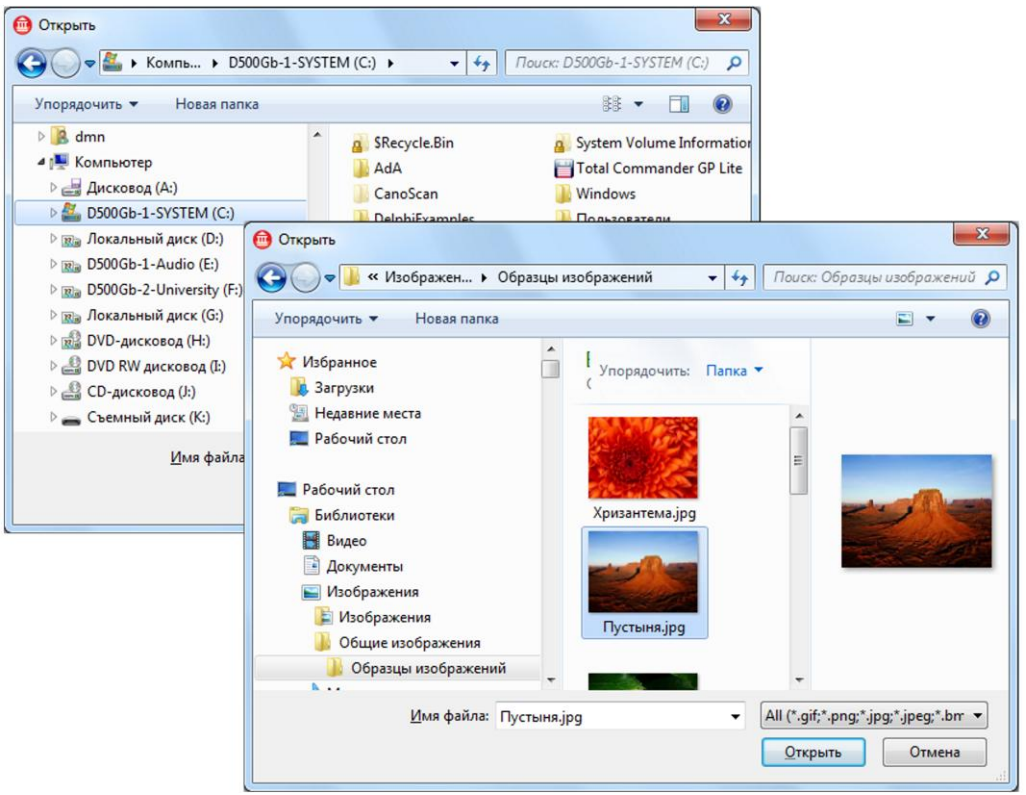


Рис. 28.6. Окна диалогов выбора файла `TOpenDialog` и изображения `TOpenPictureDialog`

Особенности графических диалогов *TOpenPictureDialog* и *TSavePictureDialog*

Классы `TOpenPictureDialog` и `TSavePictureDialog` опубликованы в модуле `ExtDlgs` и предназначены для поддержки операций открытия/сохранения файлов основных графических форматов Windows. К таким форматам относятся файлы растровой графики с расширениями имен gif, png, jpg, jpeg, bmp, tif, tiff, ico и метафайлы векторной графики с расширениями wmf и emf.

Как показано на рис. 28.4, классы `TOpenPictureDialog` и `TSavePictureDialog` являются прямыми потомками класса `TOpenDialog` и соответственно наследуют все свойства и методы, объявленные у своего родителя.

Первая, бросающаяся в глаза, особенность диалоговых окон для работы с графикой заключается в возможности просмотра выбранного файла в дополнительном окне (см. рис. 28.6). Второй отличительной чертой является ограничение списка допустимых к чтению/записи типов файлов — по умолчанию свойство `Filter` графических диалоговых окон уже заполнено списком автоматически поддерживаемых форматов.

Никто не запрещает дополнить перечень `Filter`, поместив в него расширения имен файлов альтернативных форматов. Но не забывайте, что диалоговые окна просто позволяют выбирать файл, а все задачи по его обработке в приложении ложатся на плечи программиста. Что-то решается достаточно просто, например, для работы со сжатыми растровыми изображениями формата JPEG к проекту достаточно подключить модуль `jpeg`. А есть и весьма сложные задачи, в которых придется самостоятельно разрабатывать декодер/кодер для специфичного формата файла.

Особенности текстовых диалогов *TOpenTextFileDialog* и *TSaveTextFileDialog*

Классы `TOpenTextFileDialog` и `TSaveTextFileDialog` опубликованы в модуле `ExtDlgs` и специализируются на обслуживании текстовых файлов. Текстовые диалоги построены на базе подробно изученного нами универсального диалога открытия файла `TOpenDialog` и вобрали в себя все имеющиеся в распоряжении опорного класса свойства и методы.

Основным нововведением диалогов, специализирующихся на работе с текстовыми файлами, стали свойства, помогающие управлять кодировкой символов текстового файла. Перечень приемлемых кодировок заносится в свойство

```
property Encodings: TStrings;
```

По умолчанию в список внесены кодировки: ANSI, ASCII, Unicode, Big Endian Unicode, UTF-8 и UTF-7.

Индекс кодировки по умолчанию содержится в свойстве

```
property EncodingIndex: Integer;
```

Диалоги поиска и замены текста

Компоненты `TFindDialog` и `TReplaceDialog` выводят на экран диалоговые окна, в которых пользователь должен указать текст, подлежащий поиску (или замене), и определиться с опциями поиска.

После вызова диалогового окна пользователь определяет текстовый фрагмент, который необходимо найти в тексте. Этот фрагмент записывается в свойство

```
property FindText: string;
```

Опции поиска зависят от установок свойства:

```
property Options: TFindOptions;
```

Возможные значения класса `TFindOptions` описаны в табл. 28.4.

Таблица 28.4. *Опции поиска и замены текста*

Значение	Описание
Учет регистра символов при поиске	
`frMatchCase`	Если флаг установлен, то поиск текстового фрагмента ведется с учетом регистра символов
`frDisableMatchCase`	Делает недоступным флажок учета регистра символов
`frHideMatchCase`	Прячет флажок учета регистра символов
Направление поиска	
`frDown`	Направление поиска (по умолчанию) вниз от текущего местоположения курсора
`frDisableUpDown`	Делает недоступными кнопки направления поиска (вверх и вниз)
`frHideUpDown`	Скрывает кнопки направления поиска в окне диалога
Поиск в виде целых слов	
`frWholeWord`	Если флаг установлен, то поиск заданного фрагмента осуществляется только в виде целых слов
`frDisableWholeWord`	Делает недоступным флажок поиска целыми словами
`frHideWholeWord`	Скрывает флажок поиска целыми словами
Прочее	
`frFindNext`	Флаг включается автоматически при щелчке пользователем по кнопке **Искать далее**, одновременно возникает событие `OnFind()`
`frShowHelp`	Показывает кнопку помощи в окне диалога
Опции замены текста (только для `TReplaceDialog`)	
`frReplace`	Флаг указывает, что осуществляется замена только первого найденного фрагмента в тексте
`frReplaceAll`	Флаг указывает, что должны быть заменены все найденные текстовые фрагменты `FindText` текстом замены `ReplaceText` во всем просматриваемом тексте

Для проверки, какие опции включены, а какие — нет, используют простую конструкцию `if frDown in FindDialog1.Options then...`.

В момент вывода диалога на экран допускается определить расположение окна:

```
property Position: TPoint;
```

Точка определяет левый верхний угол окна.

Допустим, что в нашем распоряжении есть проект текстового редактора, построенного на основе компонента `RichEdit1:TRichEdit` (который, я напомню, обладает собственным встроенным методом поиска текста).

Разместите на главной форме проекта компоненты:

- диалог поиска `FindDialog1:TFindDialog`;
- диалог замены `ReplaceDialog1:TReplaceDialog`;
- менеджер команд `ActionManager1:TActionManager`, в котором следует создать две команды — команду поиска `acFind:TAction` и команду замены `acReplace:TAction`.

Для настройки основных опций поиска и замены предлагаю воспользоваться услугами события `OnCreate()` главной формы проекта `frmMain`. Нам следует дополнить событие двумя строками кода (листинг 28.9).

Листинг 28.9. Настройка опций диалогов поиска и замены текста

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  . . .
  FindDialog1.Options:= FindDialog1.Options +
   [frHideWholeWord, frHideUpDown, frDisableMatchCase, frMatchCase];
  ReplaceDialog1.Options:=FindDialog1.Options;
end;
```

Листинг 28.10 содержит описание двух событий. Во-первых, обращение к команде `acFind()` инициирует вызов диалогового окна поиска текста. Во-вторых, событие `OnFind()` диалога `FindDialog1()` производит поиск и выделение в редакторе `RichEdit1` искомого текста.

Листинг 28.10. Организация поиска текста

```
procedure TfrmMain.acFindExecute(Sender: TObject);
begin
  FindDialog1.Execute(); //вызов диалога поиска
end;

procedure TfrmMain.FindDialog1Find(Sender: TObject);
var StartPos, ToEnd, ResultPos: Integer;
    SearchTypes: TSearchTypes;
begin
  //проверим опции поиска
  if frMatchCase in (Sender as TFindDialog).Options then
       SearchTypes :=[stMatchCase];
  if frWholeWord in (Sender as TFindDialog).Options then
       SearchTypes := SearchTypes + [stWholeWord];
  with RichEdit1 do
  begin
    //отправная точка для поиска
    if SelLength <> 0 then StartPos := SelStart + SelLength
                      else StartPos := 0;
    ToEnd := Length(Text) - StartPos;
    ResultPos := FindText((Sender as TFindDialog).FindText,
                          StartPos, ToEnd, SearchTypes);
```

```
    if ResultPos <> -1 then //запрашиваемый текст обнаружен
      begin
        SelStart := ResultPos;
        SelLength := Length((Sender as TFindDialog).FindText);
        SetFocus;
      end
    else MessageBeep(0);//образец не найден
  end;
end;
```

Код обработки события `OnFind()` в листинге 28.10 нуждается еще в одном важном комментарии. Обратите внимание, что в строках листинга события мы избегали прямого упоминания компонента `FindDialog1`, а вместо этого применяли конструкцию `(Sender as TFindDialog)`. Такое решение позволит нам использовать событие `OnFind()` и в интересах компонента замены текста. Поэтому сразу сейчас в Инспекторе объектов найдите событие `OnFind()` компонента `ReplaceDialog1` и свяжите его с только что описанным событием для `FindDialog1`.

Теперь приступим к решению задачи замены текста. Изучив класс `TReplaceDialog`, вы увидите, что для реакции на замену текста в компоненте объявлено событие

```
property OnReplace: TNotifyEvent;
```

Еще одна отличительная черта компонента — свойство

```
property ReplaceText: string;
```

В этом свойстве находится текст, заменяющий найденный.

Наших знаний вполне достаточно, чтобы дополнить текстовый редактор сервисом замены текста (листинг 28.11). Листинг вновь содержит код двух событий. Событие `OnExecute()` команды `acReplace()` выводит на экран диалог замены, а событие `OnReplace()` содержит строки, собственно осуществляющие замену.

Листинг 28.11. Организация замены текста

```
procedure TfrmMain.acReplaceExecute(Sender: TObject);
begin
  ReplaceDialog1.Execute; //запуск диалога замены текста
end;

procedure TfrmMain.ReplaceDialog1Replace(Sender: TObject);
var X : integer;
    s : string;
begin
  x:=0;
  while True do
    begin
      if RichEdit1.SelText<>ReplaceDialog1.FindText then
        FindDialog1.OnFind(Sender);
      If RichEdit1.SelLength=0 then Break;
      RichEdit1.SelText:=ReplaceDialog1.ReplaceText;
```

```
      if NOT (frReplaceAll in ReplaceDialog1.Options) then Break;
      x:=x+1;
    end;
  if X>1 then
    begin
      s:=Format('Произведено %d замен',[X]);
      MessageBox(Application.MainForm.Handle, pWideChar(s),
                 'Сообщение', MB_OK or MB_ICONINFORMATION);
    end;
end;
```

Замена текста производится по следующему алгоритму. Если выделенный в `RichEdit1` текст не соответствует искомому фрагменту, то событие `OnReplace()` вызывает процедуру поиска текста. В случае если поиск завершился успешно, то производится замена выделенного текста на текст замены. В рамках цикла `while...do` осуществляются две проверки на завершение цикла. Первая проверка контролирует длину выделенного текста. Если выделено `0` символов (`SelLength=0` — текст не найден), цикл прерывается. Вторая проверка контролирует режим замены. Если пользователь не нажал кнопку диалога **Заменить все**, то цикл прерывается после одного прохода, иначе цикл будет продолжаться до тех пор, пока не произведутся все замены.

Выбор шрифта *TFontDialog*

За вывод на экран стандартного диалогового окна выбора шрифта отвечает класс `TFontDialog`. Диалог позволяет выбирать один из установленных в операционной системе шрифтов и настроить его ключевые параметры.

Вызов стандартного диалога выбора шрифта осуществляется при помощи общего для большинства диалогов метода `Execute()`. В появившемся окне пользователь устанавливает предпочтительные для себя параметры шрифта и нажимает кнопку **OK**. В этом случае все указанные пользователем атрибуты шрифта передаются в основное свойство диалога

```
property Font: TFont;
```

Внешний вид и возможности диалогового окна выбора шрифта в первую очередь зависят от выбранных программистом опций

```
property Options: TFontDialogOptions;//по умолчанию [fdEffects]
```

Возможные значения опций описаны в табл. 28.5.

Таблица 28.5. *Опции диалога* `TFontDialog`

Значение	Описание
`fdAnsiOnly`	Отображать только шрифты в кодировке ANSI
`fdApplyButton`	Выводить в окне диалога кнопку **Применить**. Наличие этой кнопки предоставит возможность воспользоваться событием диалога `OnApply()`
`fdEffects`	Выводить в окне диалога флажки **Зачеркнутый** и **Подчеркнутый** и список выбора цвета шрифта
`fdFixedPitchOnly`	Включать в набор шрифтов только шрифты, у которых ширина всех символов одинакова (Courier и т. п.)

Таблица 28.5 *(окончание)*

Значение	Описание
fdForceFontExist	Допускает ввод имени гарнитуры шрифта в строке ввода. Если пользователь наберет имя, отсутствующее в списке, будет выведено сообщение об ошибке
fdLimitSize	Работает совместно со свойствами диалога MaxFontSize и MinFontSize, накладывает ограничение на диапазон размеров шрифта
fdNoFaceSel	При вызове диалога в списке выбора не задается шрифт по умолчанию
fdNoOEMFonts	Из списка отображаемых шрифтов исключаются шрифты с набором символов OEM
fdNoSimulations	Показывает только шрифты и начертания шрифта, которые непосредственно обеспечиваются файлом формирования рисунка шрифта. В список доступных шрифтов не включаются стили, синтезированные GDI Windows
fdNoSizeSel	При вызове диалога не задается размер шрифта по умолчанию
fdNoStyleSel	При вызове диалога не задается стиль начертания шрифта
fdNoVectorFonts	Изымает из списка выбора все векторные шрифты
fdScalableOnly	Показывать только масштабируемые шрифты, растровые шрифты удаляются из списка
fdShowHelp	Выводит кнопку помощи
fdTrueTypeOnly	Выводит только TrueType-шрифты
fdWysiwyg	Выводит только шрифты, доступные принтеру и экрану

Операционная система Windows обеспечивает работу с двумя категориями шрифтов: шрифты GDI и шрифты устройства. К первым относятся шрифты, хранящиеся в файлах и установленные в системе. Шрифты устройства соответствуют конкретному устройству вывода, например вашему принтеру. За выбор категории шрифтов отвечает свойство

```
property Device: TFontDialogDevice; //по умолчанию fdScreen
```

При выборе fdScreen в множество набора будут включены только шрифты GDI, fdPrinter — набор ограничится шрифтами принтера, fdBoth — обоими устройствами. В современных проектах целесообразно оставлять это свойство в состоянии по умолчанию.

Как уже отмечалось в табл. 28.5, при включенном флаге fdLimitSize программист способен ограничить высоту символов выбираемого шрифта минимальным и максимальным пределом:

```
property MinFontSize: Integer;
property MaxFontSize: Integer;
```

Выбор цвета *TColorDialog*

Класс TColorDialog формирует диалоговое окно выбора цвета пользователем. Настройка внешнего вида и поведения диалога, как и в предыдущих случаях, осуществляется через опции компонента

```
property Options: TColorDialogOptions;
```

Описание доступных опций представлено в табл. 28.6.

Таблица 28.6. *Опции диалога* `TColorDialog`

Значение	Описание
`cdAnyColor`	Допускает выбор не только так называемых чистых цветов, но и полутонов
`cdSolidColor`	При работе с цветовыми палитрами требует найти в палитре цвет, наиболее похожий на цвет, выбранный пользователем
`cdFullOpen`	При вызове диалога сразу предоставляет возможность выбора дополнительных цветов
`cdPreventFullOpen`	Запрещает выбор дополнительных цветов, отключая кнопку выбора цветов
`cdShowHelp`	Дополняет диалог кнопкой помощи

Для вызова диалога выбора цвета воспользуйтесь методом `Execute()`. Выбранный пользователем цвет можно получить, прочитав значение свойства

```
property Color: TColor;
```

Диалог поддерживает до 16 дополнительных (определенных пользователем) цветов. Доступ к ним осуществляется при помощи свойства:

```
property CustomColors: TStrings;
```

Каждая строка в наборе представляет собой пару вида: `ColorX = шестнадцатеричный код` цвета в формате RGB. Например: `ColorA = 808022`.

Параметры страницы *TPageSetupDialog*

Диалог настройки параметров страницы предоставляет удобный интерфейс управления основными параметрами бумажной страницы и может пригодиться в проектах текстовых и графических редакторов, при определении основных характеристик бумажных отчетов в проектах баз данных и при подготовке документа к печати. Обращение к стандартному для всех диалогов методу `Execute()` вызывает окно настройки страницы перед печатью (рис. 28.7).

Работу с параметрами страницы следует начинать с определения размеров страницы. Для удобства пользователя диалоговое окно обладает раскрывающимся списком **Размер** (см. рис. 28.8), в котором хранятся элементы с заранее предустановленными размерами (A3, A4, A5 и т. п.), ориентация страницы выбирается в группе **Ориентация**. Настроенные пользователем высота и ширина страницы передаются в пару свойств

```
property PageHeight: Integer;
property PageWidth: Integer;
```

Используемые в диалоге единицы измерения определяются свойством

```
property Units: TPageMeasureUnits; //по умолчанию pmDefault
type TPageMeasureUnits = (pmDefault,     //локальные установки системы
                          pmMillimeters, //миллиметры
                          pmInches);     //дюймы
```

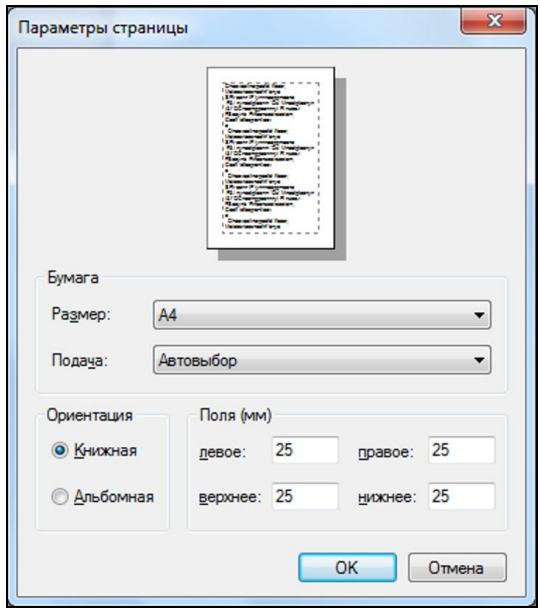


Рис. 28.7. Внешний вид окна диалога `TPageSetupDialog`

Кроме размеров страницы диалог позволяет пользователю определять поля. Для этого предназначена четверка свойств

```
property MarginLeft: Integer;  {левый отступ}
property MarginRight: Integer; {правый отступ}
property MarginTop: Integer;   {верхний отступ}
property MarginBottom: Integer;{нижний отступ}
```

Минимально допустимые значения полей ограничиваются соответствующими свойствами:

```
property MinMarginLeft: Integer;
property MinMarginRight: Integer;
property MinMarginTop: Integer;
property MinMarginBottom: Integer;
```

Особенности диалогового окна определяет классическое свойство

```
property Options: TPageSetupDialogOptions;
```

Опции представлены в табл. 28.7.

***Таблица 28.7.** Опции диалога `TPageSetupDialogOptions`*

Опция	Описание
`psoDefaultMinMargins`	Назначает ограничение на минимальное значение полей страницы, которые могут быть назначены пользователем. Значения определяются установками текущего принтера
`psoDisableMargins`	Запрещает пользователю настраивать поля страницы
`psoDisableOrientation`	Запрещает изменять ориентацию страницы
`psoDisablePagePainting`	Отменяет режим прорисовки диалога по умолчанию

Таблица 28.7 *(окончание)*

Опция	Описание
`psoDisablePaper`	Запрещает изменять размер бумаги и настраивать особенности ее подачи
`psoDisablePrinter`	Отключает кнопку вызова дополнительного диалога настройки принтера
`psoMargins`	Устанавливает все поля страницы в 1 дюйм
`psoMinMargins`	Устанавливает минимальные поля из свойств `MinMarginLeft`, `MinMarginRight`, `MinMarginBottom` и `MinMarginTop`, в противном случае эти поля определяются параметрами принтера
`psoShowHelp`	Показывает кнопку помощи
`psoWarning`	Отключает сообщение об ошибке при отсутствии установленного принтера
`psoNoNetworkButton`	Скрывает и отключает кнопку сетевых устройств

Настройка печати *TPrinterSetupDialog*

Класс `TPrinterSetupDialog` вызывает стандартное окно настройки параметров принтера. Из всех существующих в Delphi компонентов-диалогов диалог настройки принтера самый неприхотливый в программировании. После выбора компонента в Инспекторе объектов мы обнаружим 4 опубликованных свойства (`Name`, `Tag`, `HelpContext` и `Ctl3d`) и 2 события (`OnClose` и `OnShow`). Столь скромный список свойств объясняется тем, что все сделанные пользователем настройки принтера не возвращаются в наше приложение, а передаются в системный реестр Windows, откуда они будут считаны при отправке задания на печать.

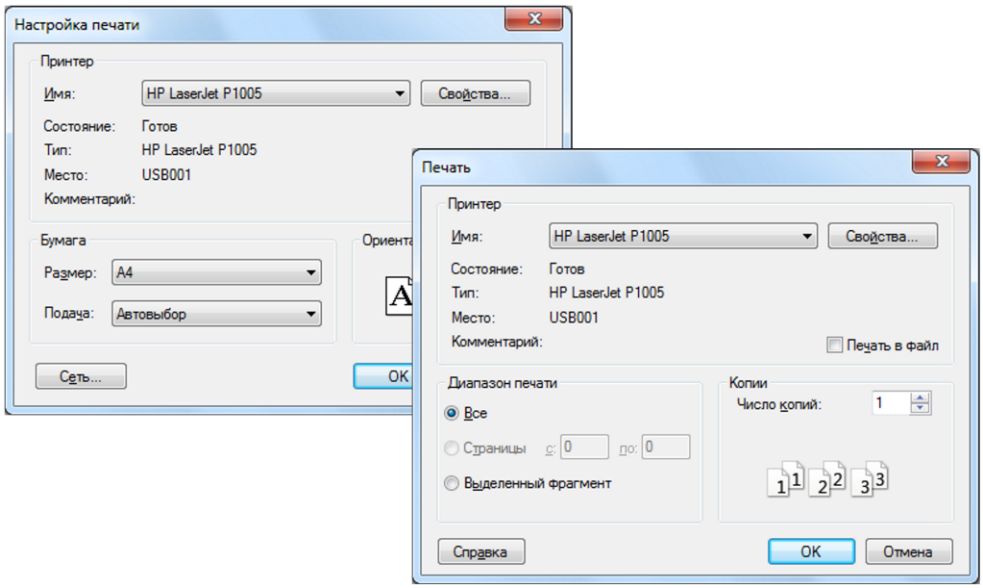


Рис. 28.8. Внешний вид диалогов `TPrinterSetupDialog` и `TPrintDialog`

Для активизации диалогового окна предназначена функция `Execute()`, в результате на экране компьютера будет отображено диалоговое окно, представленное на рис. 28.8.

Отправка задания на печать *TPrintDialog*

В сравнении с диалогом настройки принтера диалог печати задания отличается существенным спектром параметров. Как и у всех диалогов, любимым методом диалога печати является функция `Execute()`. В результате на экране компьютера возникнет стандартное диалоговое окно, позволяющее выбрать принтер, установить диапазон распечатываемых листов и число копий (см. рис. 28.8).

Особенности поведения диалогового окна определяются опциями, устанавливаемыми в свойстве

```
property Options: TPrintDialogOptions;
```

Значения флагов свойства приведены в табл. 28.8.

Таблица 28.8. *Опции диалога печати* `TPrintDialogOptions`

Значение	Описание
`poPrintToFile`	Выводит флажок перенаправления задания печати в файл
`poDisablePrintToFile`	Запрещает печать в файл (делая флажок **Печать в файл** неактивным)
`poHelp`	Отображает кнопку помощи в окне диалога
`poPageNums`	Разрешает пользователю выбирать диапазон страниц, отправляемых на печать. В противном случае будет отправлен весь перечень страниц
`poSelection`	Разрешает пользователю печатать только выделенный фрагмент текста
`poWarning`	Генерирует сообщение об ошибке при попытке отправить задание на неустановленный принтер

Большинство свойств диалога тем или иным образом взаимодействует с опциями окна печати. При использовании флагов `poPageNums` и `poSelection` пользователь получает право отправлять на печать несколько страниц или фрагмент задания. Результат выбора пользователя отразится в свойстве

```
property PrintRange: TPrintRange;
```

При выборе для печати только фрагмента текста свойство `PrintRange` примет значение `prSelection`, при выборе нескольких страниц — `prPageNums`. Если на печать отправляется весь документ, то результат будет соответствовать `prAllPages`.

Совместно с опцией диалога печати `poPageNums` трудятся свойства

```
property FromPage: Integer;
property ToPage: Integer;
```

Значения свойств определяют начальную и конечную страницы диапазона печати. В случае если оба свойства установлены в 0, то на печать направляется весь документ.

При особом желании допустимо ограничить возможности пользователя по определению диапазона направляемых на печать страниц.

```
property MinPage: Integer;
property MaxPage: Integer;
```

По умолчанию значения свойств равны 0, т. е. ограничения отсутствуют. При выходе за границы генерируется сообщение об ошибке.

Если допустимо перенаправление задания печати в файл и пользователь воспользовался такой возможностью, то свойство

```
property PrintToFile: Boolean;
```

примет значение `true`.

Количество копий предназначенного для печати документа вы обнаружите в свойстве

```
property Copies: Integer;
```

Если число соответствует 0 или 1, то будет отпечатан только один экземпляр.

Диалог управления задачей *TTaskDialog*

Компонент `TTaskDialog` инкапсулирует в себе возможности диалогового окна современных версий Windows Vista/7. В первую очередь диалог призван заместить имеющийся в распоряжении приложения `Application` метод `MessageBox()`, выводящий невыразительное диалоговое окно.

Благодаря широкому спектру свойств у диалога `TTaskDialog` появляются до сих пор недоступные возможности как по управлению внешним видом окна, так и по предоставляемым пользователю функциональным возможностям.

Работу с компонентом следует начинать с настройки его опций. Эта операция осуществляется при посредничестве свойства

```
property Flags: TTaskDialogFlags;
```

предоставляющего доступ к набору флагов (табл. 28.9).

Таблица 28.9. *Опции диалога* `TTaskDialog`

Опция	Описание
`tfEnableHyperlinks`	Текст может содержать гиперссылки
`tfUseHiconMain`	Допускается переназначение главного значка окна
`tfUseHiconFooter`	Допускается назначение дополнительного значка
`tfAllowDialogCancellation`	Диалог может быть закрыт без помощи кнопки **Отмена**
`tfUseCommandLinks`	Кнопки отображаются в виде командных ссылок
`tfUseCommandLinksNoIcon`	При отображении командных ссылок не используются значки
`tfExpandFooterArea`	Отображает текст в нижней части окна
`tfExpandedByDefault`	По умолчанию область, содержащая дополнительный текст, развернута (свойство `ExpandedText`)
`tfVerificationFlagChecked`	Флажок верификации (свойство `VerificationText`) по умолчанию отмечен галочкой
`tfShowProgressBar`	Включен показ шкалы прогресса выполнения
`tfShowMarqueeProgressBar`	Создает дополнительный эффект на шкале прогресса
`tfCallbackTimer`	К окну диалога каждые 200 миллисекунд посылается обращение

Таблица 28.9 *(окончание)*

Опция	Описание
tfPositionRelativeToWindow	Диалог цитируется относительно родительского окна
tfRtlLayout	Для арабского письма выводит текст справа налево
tfNoDefaultRadioButton	Во множестве радиокнопок отсутствует переключатель по умолчанию
tfCanBeMinimized	Окно может быть свернуто

На поверхности диалогового окна может быть отображено шесть элементов текстовой информации, определяемых в свойствах

```
property Caption: string;           //заголовок окна
property Title : string;            //заголовок основной области
property Text : string;             //основной текст диалога
property ExpandedText: string;      //текст с подробностями
property FooterText: string;        //текст нижнего колонтитула
property VerificationText : string; //текст подтверждения
```

Заметим, что при заполнении свойства с текстом верификации автоматически создается кнопка-флажок, щелчок по которой вызывает событие

```
property OnVerificationClicked : TNotifyEvent;
```

Место вывода той или иной строки текста отражает рис. 28.9.

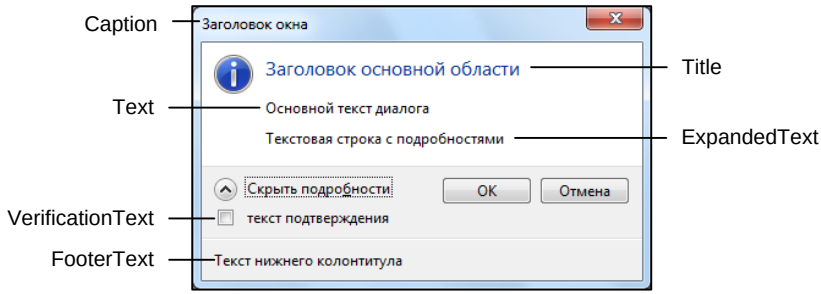


Рис. 28.9. Текстовая информация на окне `TTaskDialog`

Управление набором кнопок, отображаемых на поверхности окна, осуществляет свойство

```
property CommonButtons : TTaskDialogCommonButtons; //[tcbOk,tcbCancel]
```

Какая из кнопок окажется в фокусе ввода в момент вывода диалога, отвечает свойство

```
property DefaultButton : TTaskDialogCommonButton;
```

К перечню стандартных кнопок программист имеет право добавить несколько дополнительных с помощью свойства

```
property Button: TTaskDialogButtonItem;
```

Еще одну изюминку диалогу добавляет свойство

```
property RadioButtons : TTaskDialogButtons;
```

позволяющее размещать на поверхности окна группу кнопок выбора.

Щелчки по кнопкам могут быть обработаны в рамках событий

```
property OnButtonClicked : TTaskDlgClickEvent;//стандартные кнопки
property OnRadioButtonClicked : TNotifyEvent; //группа кнопок выбора
```

При необходимости диалог может быть дополнен шкалой прогресса выполнения (см. флаг `tfShowProgressBar`). Управление шкалой осуществляется с помощью свойства

```
property ProgressBar : TTaskDialogProgressBar;
```

Одним из способов управления шкалой может стать событие встроенного таймера, обрабатываемого в событии

```
property OnTimer : TTaskDlgTimerEvent;
```

ГЛАВА 29

Технология естественного ввода

Если еще три десятилетия назад персональная ЭВМ для большинства из нас была невиданной диковинкой, до которой боялись даже прикоснуться, то сегодня перед компьютером не спасует даже ребенок. Почему так произошло? Неужели нынешние дети умнее детей, появившихся на свет четверть века назад? Надеюсь, что да. Но скорее всего, отсутствие боязни перед электронной техникой объясняется проще — современные компьютеры очень дружелюбны.

В первую очередь дружелюбие компьютера проявляется через развитый пользовательский интерфейс. И с каждым десятилетием этот интерфейс приобретает новое качество. Судите сами. В середине XX века основным способом ввода данных были перфокарты и перфоленты, полагаю, что об удобстве здесь можно даже не говорить. Позднее появились клавиатуры, еще позднее — манипуляторы-мыши. Первые монохромные устройства отображения на электронно-лучевых трубках постепенно сменились цветными мониторами, а сегодня они уже почти полностью вытеснены жидкокристаллическими дисплеями. Но прогресс не стоит на месте. Пользователи проявляют все возрастающий интерес к интеллектуальным устройствам, обладающим сенсорными функциями ввода данных. Это в первую очередь многочисленные платежные терминалы, планшетные компьютеры, электронные доски и мобильные телефоны. Как правило, перечисленные устройства не снабжены отдельной клавиатурой или же возможности их клавиатуры сильно ограничены. Но это не беда, программисты научили эти устройства воспринимать естественно ввод или, говоря проще, жесты человека.

> ***Замечание***
>
> Под *жестом* (gesture) мы станем понимать произвольную геометрическую фигуру, нарисованную пользователем на устройстве сенсорного ввода с помощью электронного пера или мышью.

В корпорации Microsoft полноценная поддержка механизма естественного ввода реализована, начиная с операционной системы Windows 7. Компания Embarcadero пошла еще дальше — ее технология распознавания жестов (кроме безусловной поддержки возможностей Windows 7) сохраняет работоспособность и в более ранних версиях ОС (Windows Vista/XP).

Описание жеста

Большинство построенных на базе класса `TControl` элементов управления Delphi способно реагировать на некоторый набор заранее предопределенных жестов. Жест представляет

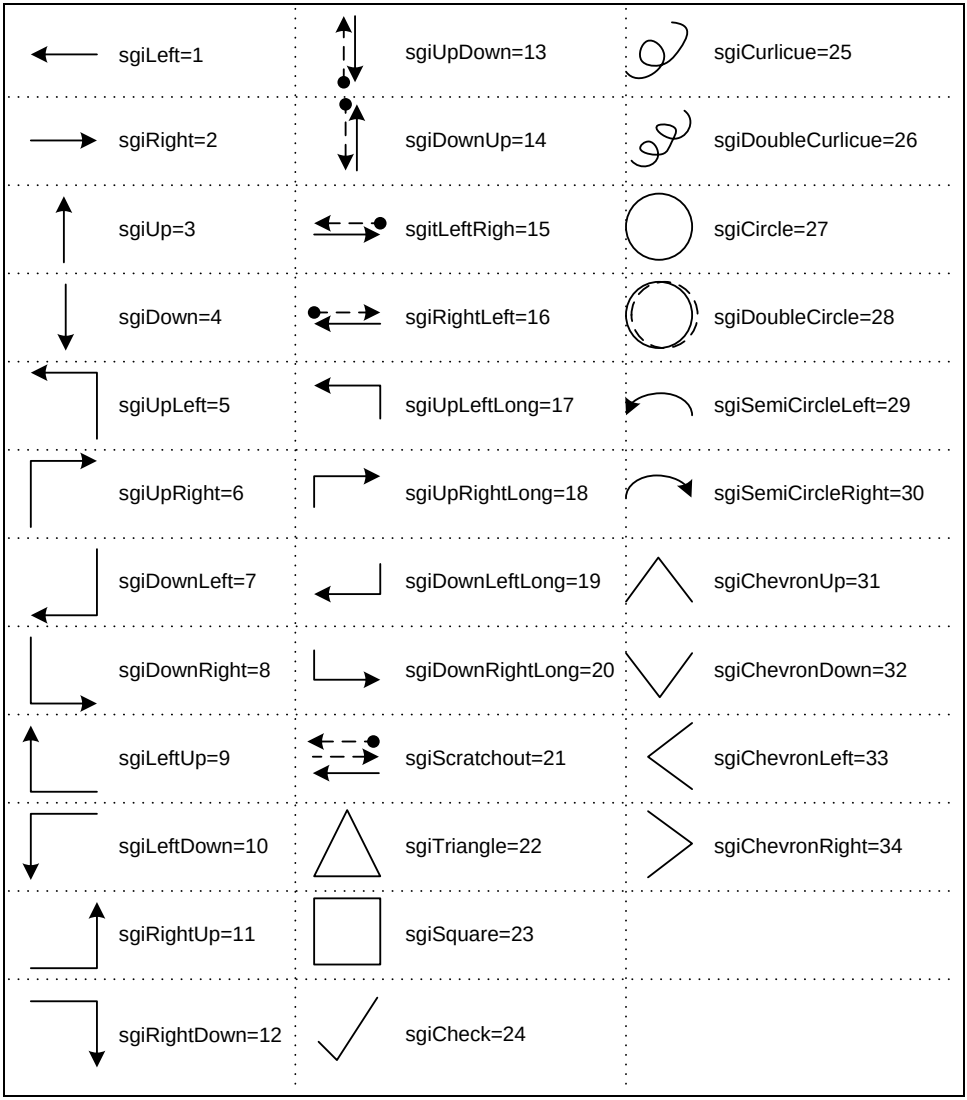


Рис. 29.1. Стандартные жесты и их константы

собой некую геометрическую фигуру, хранимую в памяти в формате структуры `TStandardGestureData` (листинг 29.1).

Листинг 29.1. Состав полей записи `TStandardGestureData`

```
type TStandardGestureData = record
   Points: TGesturePointArray; {массив точек TPoint, описывающих жест}
   GestureID: TGestureID;      {идентификатор жеста}
   Options: TGestureOptions;   {опции}
   Deviation: Integer;     {допустимое отклонение жеста от стандартного}
   ErrorMargin: Integer;   {максимальное число ошибок}
end;
```

Ключевое поле структуры — `Points`. Это динамический массив, хранящий координаты точек геометрической фигуры жеста. Второе поле содержит идентификатор жеста. В Delphi имеется более трех десятков предустановленных жестов (рис. 29.1). При анализе жеста учитывается не только его соответствие геометрической фигуре, но и ряд других характеристик. В частности, поле опций `Options` контролирует направление `goDirectional`, наклон `goSkew` и факт совпадения начальной и конечной точек `goEndpoint`.

Реакция элементов управления на жест

Начиная с Delphi 2010, ввод данных с помощью жестов поддерживается большинством элементов управления (потомков класса `TControl`). Для того чтобы элемент управления приобрел способность реагировать на предопределенные жесты, ему понадобится помощник — менеджер жестов, компонент `TGestureManager`. Для подключения менеджера жестов к элементу управления необходимо сделать несколько шагов. Сначала следует обратиться к свойству

```
property Touch: TTouchManager;
```

Свойство предоставляет доступ к инкапсулированному в элемент управления объекту — менеджеру прикосновений (экземпляру класса `TTouchManager`). В свою очередь у менеджера прикосновений имеется свойство

```
property GestureManager: TGestureManager;
```

позволяющее ассоциировать с элементом управления интересующий нас менеджер жестов `TGestureManager`.

Процесс подключения компонента `TGestureManager` к форме проекта отражает экранный снимок Инспектора объектов (рис. 29.2). Обратите внимание на то, что воспользовавшись разделом **Gestures | Standard**, программист определяет, на какие именно жесты должен реагировать элемент управления. Для этого достаточно поставить "галочку" рядом с соответствующим изображением.

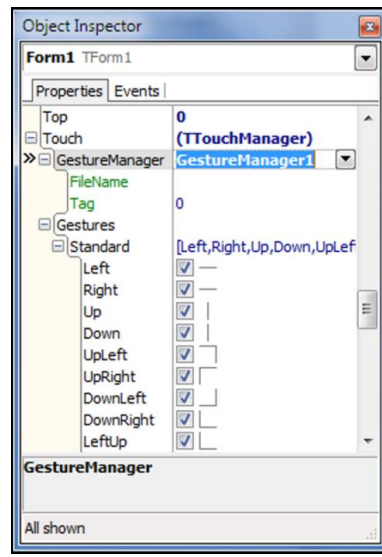


Рис. 29.2. Присоединение менеджера `TGestureManager` к форме

Для описания реакции на жест программисту следует воспользоваться событием

```
property OnGesture: TGestureEvent;
type TGestureEvent = procedure(Sender: TObject;
     const EventInfo: TGestureEventInfo; var Handled: Boolean) of object;
```

Событие генерируется в тот момент, когда пользователь (воспользовавшись сенсорным экраном, электронным пером или просто мышью) нарисовал над поверхностью элемента управления какую-то геометрическую фигуру.

Основная задача обработчика события заключается в двух вещах:

- провести анализ введенного жеста и найти ему наиболее точное соответствие среди жестов, имеющихся в распоряжении элемента управления (свойство `Touch`);
- выполнить соответствующую жесту операцию.

Ключевой параметр события — `EventInfo`, именно он уведомляет нас о том, какая фигура была нарисована пользователем. Параметр представляет собой запись `TGestureEventInfo`, объявление которой вы найдете в листинге 29.2. Установив последний параметр события `Handled` в состояние `true`, мы уведомим систему, что жест в обработке более не нуждается.

Листинг 29.2. Состав полей записи `TGestureEventInfo`

```
type TGestureEventInfo = record
   GestureID: TGestureID;
   Location: TPoint;
   Flags: TInteractiveGestureFlags;
   Angle: Double;
   InertiaVector: TSmallPoint;
   case Integer of
     0: (Distance: Integer);
     1: (TapLocation: TSmallPoint);
   end;
end;
```

Назначение полей структуры `TGestureEventInfo` предложено в табл. 29.1.

***Таблица 29.1.** Описание полей записи `TGestureEventInfo`*

Поле записи	Описание
`GestureID`	Идентификатор жеста
`Location`	Координаты текущей точки на поверхности устройства ввода
`Flags`	Набор флагов (`gfBegin`, `gfInertia`, `gfEnd`) доступных только в момент ввода жеста
`Angle`	Угол движения электронного пера (курсора мыши, пальца пользователя) относительно координатных осей устройства ввода
`InertiaVector`	Пара значений `X` и `Y`, благодаря которым можно идентифицировать направление движения электронного пера. Положительное значение `X` свидетельствует о движении пера к правой границе, отрицательное — к левой. Положительное значение `Y` говорит о том, что перо опускается вниз экрана, отрицательное — поднимается вверх
`Distance`	Расстояние в пикселах между текущей (`Location`) и предыдущей точками
`TapLocation`	Местоположение начальной точки фигуры жеста

Замечание

Стоит заметить, что значения идентификаторов стандартных жестов имеют положительные значения (от 1 и далее), значения идентификаторов пользовательских жестов всегда отрицательные (от –1 и далее).

Пример обработки стандартных жестов

Специалисты Embarcadero приложили все усилия для того, чтобы программист Delphi чувствовал себя максимально комфортно при использовании в проекте механизма естественного ввода. Подтверждение тому — простота создания приложения, способного общаться с пользователем с помощью жестов.

Для нашего примера понадобится новый проект и компонент `TGestureManager`, который следует подключить к форме `Form1` с помощью свойств `Touch—GestureManager`. Отметьте те жесты, на которые должна реагировать форма (см. рис. 29.2). Разместите на форме метку `Label1`, этот компонент проинформирует нас о жесте пользователя.

Собственно обработка жеста будет осуществлена в рамках события `OnGesture()` единственной формы проекта (листинг 29.3).

Листинг 29.3. Обработка события `OnGesture()` формой проекта

```
procedure TForm1.FormGesture(Sender: TObject;
  const EventInfo: TGestureEventInfo; var Handled: Boolean);
begin
  if EventInfo.GestureID>-1 then
        Label1.Caption:=IntToStr(EventInfo.GestureID)
  else Label1.Caption:= '?'
end;
```

Теперь, нарисовав одну из стандартных геометрических фигур в клиентской области формы, вы моментально узнаете ее идентификационный номер. Если вы "жестикулируете" с помощью мыши, то можно сделать так, чтобы жест стал видимым. Для этого стоит воспользоваться событием `OnMouseMove()` формы (листинг 29.4).

Листинг 29.4. Визуализация жеста пользователя

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if ssLeft in Shift then
    with Form1.Canvas do
      begin
        Pen.Color:=clBlue;
        Brush.Color:=clBlue;
        Ellipse(x-2,y-2,x+2,y+2);
      end;
end;
```

Результаты работы приложения, способного распознавать стандартные жесты, представлены на рис. 29.3. В данном случае форма распознала состоящий из двух петель стандартный жест с номером 26.

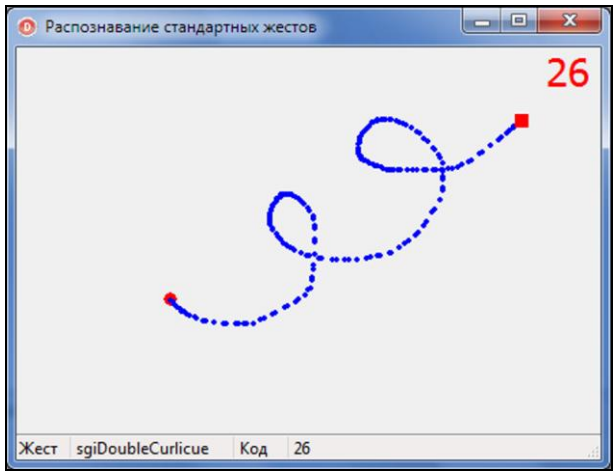


Рис. 29.3. Экранный снимок формы распознавания жестов

Компоненты поддержки естественного ввода

Просмотрев палитру компонентов Delphi, вы обнаружите специализированную страницу **Gestures**, на которой расположился квартет компонентов, нацеленных на обслуживание механизма естественного ввода (рис. 29.4). В этом перечне один невизуальный и три визуальных элемента управления:

- на первых страницах главы нам уже встречался менеджер жестов `TGestureManager`, этот компонент отвечает за хранение перечня жестов;
- список жестов `TGestureListView` предоставляет удобный доступ к хранимым в менеджере `TGestureManager` жестам;
- элемент управления `TGesturePreview` предназначен для просмотра жеста;
- область ввода жеста `TGestureRecorder` предоставляет сервис записи дополнительных жестов.

Менеджер жестов *TGestureManager*

Центральным узлом приложения, поддерживающего механизм естественного ввода, выступает менеджер жестов `TGestureManager`. Основная задача элемента управления — хранение зарегистрированных в приложении жестов и установка ассоциации между жестами и элементами управления приложения.

Обслуживаемые менеджером жесты разделяются на две категории:

- предустановленные в системе жесты;
- введенные программистом или пользователем дополнительные жесты.

Для создания (редактирования или удаления) жестов во время визуального проектирования достаточно воспользоваться встроенным редактором компонента. Для его вызова достаточно дважды щелкнуть по компоненту `TGestureManager` или воспользоваться пунктом **Custom Gestures** его контекстного меню. Редактор **Custom Gesture Designer** предложит вам нарисовать фигуру и ввести ее имя (рис. 29.5).

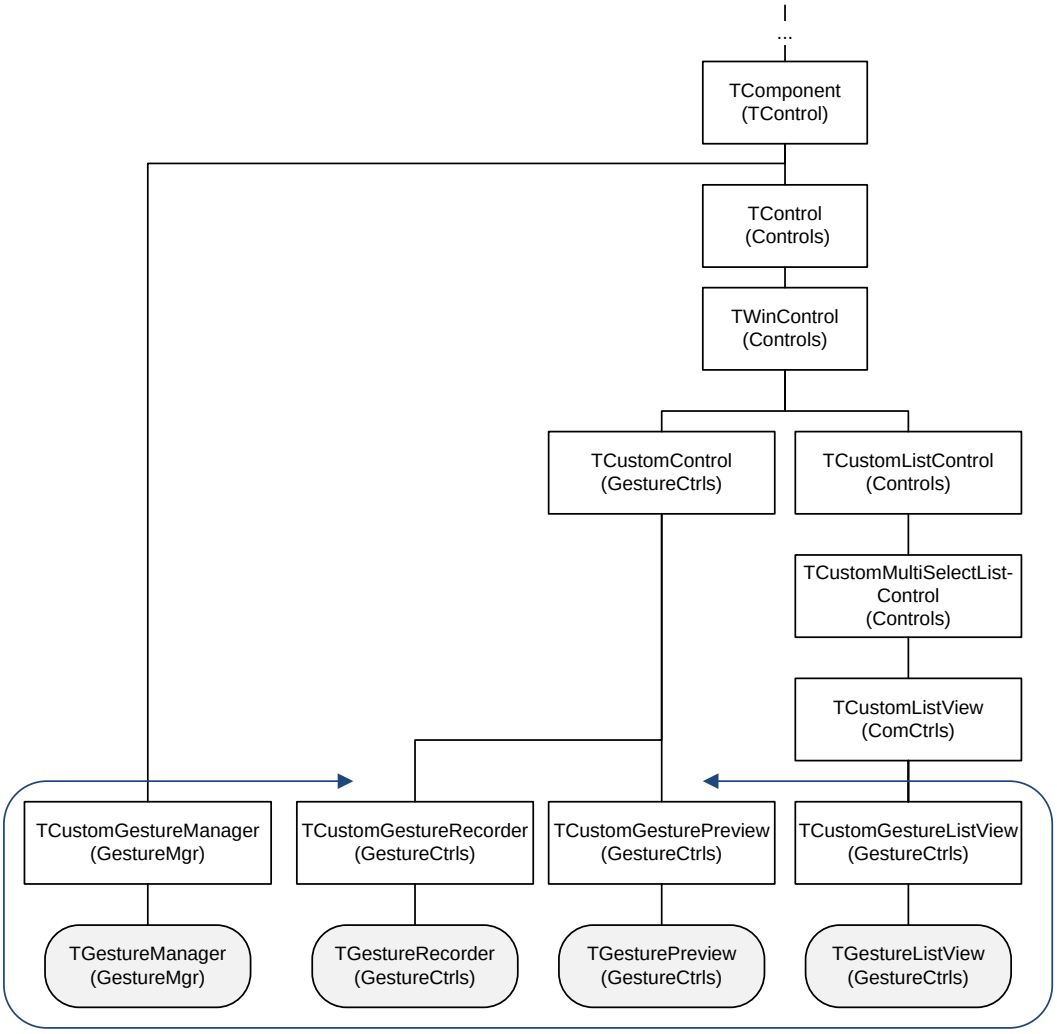


Рис. 29.4. Место элементов управления естественного ввода в иерархии VCL

Доступ к жестам и их сохранение

Вне зависимости от настроек менеджера жестов в операционной системе зарегистрирован некоторый перечень стандартных жестов (см. рис. 29.1). Доступ к зарегистрированным в системе жестам осуществляется при посредничестве свойств

```
class property RegisteredGestureCount: Integer;   {число жестов}
class property RegisteredGestures: TGestureArray; {массив жестов}
```

Обратите внимание на то, что доступ реализован при помощи свойств класса, т. е. свойств, существование которых не зависит от физического наличия экземпляра класса `TGestureManager`.

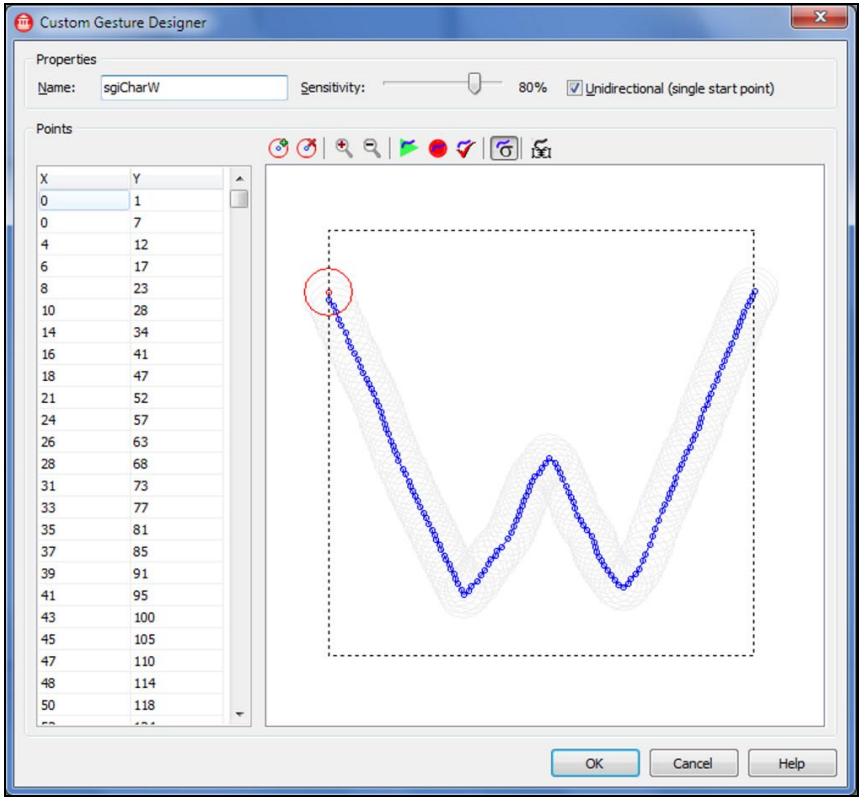


Рис. 29.5. Редактор пользовательских жестов

Записанные во время выполнения приложения (с помощью подключенной к менеджеру области ввода `TGestureRecorder`) жесты обслуживаются парой свойств

```
property RecordedGestureCount: Integer;  //число пользовательских жестов
property RecordedGestures: TGestureArray;//массив пользовательских жестов
```

Жесты хранятся в коллекции `TGestureArray`, каждый элемент которой описывается классом `TCustomGestureCollectionItem`.

Одновременный доступ и к пользовательским, и к стандартным жестам обеспечивают свойства

```
property CustomGestureCount: Integer;   //всего жестов
property CustomGestures: TGestureArray; //массив жестов
```

Все шесть представленных выше свойств доступны только для чтения.

Менеджер поддерживает два способа хранения жестов: в файле и в потоке. Загрузка данных осуществляется с помощью методов

```
procedure LoadFromFile(const Filename: string); //загрузка из файла
procedure LoadFromStream(S: TStream); //загрузка из потока
```

Сохранение жестов обеспечивают методы

```
procedure SaveToFile(const Filename: string);
procedure SaveToStream(S: TStream);
```

Замечание

В качестве расширения имени файла, хранящего жесты в Delphi, принято gdf.

Группа одноименных перегружаемых методов специализируется на поиске жеста в коллекции:

```
function FindCustomGesture(
         AGestureID: TGestureID): TCustomGestureCollectionItem; override;
function FindCustomGesture(
         const AName: string): TCustomGestureCollectionItem; override;
function FindGesture(AControl: TControl;
       AGestureID: TGestureID): TCustomGestureCollectionItem; override;
function FindGesture(AControl: TControl;
          const AName: string): TCustomGestureCollectionItem; override;
```

В качестве параметров поиска могут использоваться идентификатор жеста `AGestureID`, название жеста `AName`, ссылка на элемент управления `AControl`, реагирующего на жест.

Для удаления жеста из коллекции следует воспользоваться любым из предложенных далее методов:

```
procedure RemoveRecordedGesture(AGestureID: TGestureID); override;
procedure RemoveRecordedGesture(AGesture: TCustomGestureCollectionItem); override;
```

В первом случае нам потребуется идентификатор жеста, а во втором — ссылка на экземпляр жеста.

Просмотр жестов, *TGestureListView* и *TGesturePreview*

Если разрабатываемое приложение предполагает предоставление возможности просмотра имеющихся в его распоряжении жестов, то на помощь менеджеру жестов могут прийти еще два элемента управления: список жестов `TGestureListView` и просмотр жеста `TGesturePreview`.

Список жестов способен автоматически получить от менеджера `TGestureManager` перечень доступных жестов, для этого достаточно лишь подключить список к менеджеру с помощью свойства

```
property GestureManager:TCustomGestureManager;
```

После присоединения к менеджеру в списке отобразятся уменьшенные копии геометрических фигур жестов и их названия.

Для того чтобы пользователь смог более подробно разобраться с особенностями начертания жеста, к списку жестов можно подключить еще один элемент управления — компонент `TGesturePreview`.

Элемент управления `TGesturePreview` предоставляет услугу по просмотру жеста, для этого его достаточно подключить к списку с помощью свойства

```
property GestureProvider: IGestureProvider
```

Область ввода жеста *TGestureRecorder*

Элемент управления `TGestureRecorder` понадобится программисту, разрабатывающему приложение, которое способно работать не только со стандартными жестами, но и допускает создание пользовательских жестов.

Сразу после размещения области ввода на форме приложения следует подключиться к обслуживающему приложение менеджеру, для этого потребуется помощь свойства

```
property GestureManager: TGestureManager;
```

Сразу после подключения область ввода сможет самостоятельно отправлять нарисованные пользователем жесты в свой менеджер.

Ключевое событие элемента управления

```
property OnGestureRecorded: TGestureRecordedEvent;
type TGestureRecordedEvent = procedure(Sender: TObject;
                        RecordedGesture: TGestureCollectionItem) of object;
```

генерируется в тот момент, когда пользователь успешно завершил ввод нового жеста. Кроме ссылки на собственно область ввода (ссылку вы обнаружите в первом параметре события `Sender`), в коде события мы получим доступ к только что созданному экземпляру жеста `RecordedGesture`.

Виртуальная клавиатура *TTouchKeyboard*

Компонент `TTouchKeyboard` незаменим при проектировании приложений для планшетных компьютеров и других устройств, не обладающих собственной клавиатурой. Компонент самодостаточен и не требует от программиста какого-то дополнительного вмешательства. Можете в этом легко убедиться: перенесите на главную форму пустого проекта клавиатуру и многострочный редактор `Memo1`. "Программирование" закончилось, смело запускайте проект на выполнение (рис. 29.6).

Рис. 29.6. Внешний вид виртуальной клавиатуры

Хотя при обращении к Инспектору объектов Delphi раскроется внушительный список свойств компонента `TTouchKeyboard`, в подавляющем большинстве они унаследованы от хорошо знакомого нам класса `TControl` и в дополнительных комментариях не нуждаются.

Свойство

```
property Layout: TKeyboardLayout;
```

позволяет переключаться между стандартной и дополнительной цифровой клавиатурой.

Пара свойств отвечает за настройку повторов нажатия обычных клавиш и клавиши удаления:

```
property RepeatRate: Cardinal; //по умолчанию 50 миллисекунд
property RepeatDelay: Cardinal; //по умолчанию 300 миллисекунд
```

В конце обсуждения упомянем свойство

```
property KeyCaptions: TKeyCaptions;
```

позволяющее изменить заголовки служебных клавиш (<Esc>, <Backspace>, <Tab> и т. д.).

ГЛАВА 30

Управление графическим выводом

Вплоть до начала 1990-х годов подавляющее большинство операционных систем не имело графической оболочки. Множество разнообразных ОС (в том числе и MS-DOS от фирмы Microsoft) предоставляли в распоряжение пользователя командную строку, из которой тот управлял своим компьютером и запускал необходимые в данный момент приложения. Не стоит говорить о том, что знания аппаратной и программной части компьютера обычного у пользователя тех старозаветных времен на порядок превосходили знания современного "продвинутого" пользователя, с трудом находящего нужную клавишу.

Отсутствие хоть сколько-нибудь значимой графической среды объяснялось не ленью разработчиков операционных систем, а более прозаичной причиной — отсутствием недорогих видеокарт и дисплеев. Ведь более-менее приемлемый видеоадаптер VGA (Video Adapter Array) появился на свет только в 1987 году. Он работал с 256 цветами при разрешении 320×200 пикселов. Кроме того, VGA был способен обслуживать растр размером 640×480 пикселов, но на этот раз его предел составлял 16 цветов. Сегодня эти параметры вызывают улыбку. Но тогда это был настоящий прорыв, ведь появление видеоадаптеров VGA и немного позднее SVGA позволило разработчикам операционных систем начать работу по совершенствованию их графического интерфейса.

> ***Замечание***
>
> Стандарт VGA просуществовал более 25 лет, и только сейчас во втором десятилетии XXI века основные производители компьютерной техники стали смещать акценты в пользу цифровых интерфейсов HDMI и DisplayPort.

Получение сведений об устройствах видеовывода

Очень часто работа современного приложения не замыкается на выводе графической информации на единственный монитор, операционная система Windows поддерживает одновременную работу на двух дисплеях (дисплее и проекторе, электронных досках и других устройствах представления графической информации). Если логика поведения программы должна изменяться в зависимости от того, на каком устройстве отображения ее придется выводить, стоит познакомиться с функцией, способной собрать данные о доступных в системе устройствах видеовывода:

```
function EnumDisplayDevices(Device : Pointer;
                            DevNum : Cardinal;
                            var DD : TDisplay_DeviceW;
                            Flags : Cardinal) : boolean;
```

Здесь `Device` — указатель на название конкретного устройства, допускается вместо имени направлять в параметр неопределенный указатель `nil`. В таком случае функцией собираются данные о доступных видеоадаптерах компьютера, порядковые номера которых передаются в параметр `DevNum`. Для построения списка всех адаптеров необходимо организовать цикл вызовов `EnumDisplayDevices()`, последовательно (начиная с нуля) увеличивая значение `DevNum` на единицу. Как только устройства закончатся, функция `EnumDisplayDevices()` отрапортует нам значением `false`. Результаты трудов функции мы обнаружим в третьем по счету параметре `DD` — это структура `TDisplayDeviceW`. Последний аргумент функции пока не используется, поэтому `EnumDisplayDevices()` никоим образом не обидится, получив в параметр `Flags` нулевое значение.

```
type TDisplayDeviceW = packed record
   cb: DWORD;                                  //размер структуры
   DeviceName: array[0..31] of AnsiChar;       //имя устройства
   DeviceString: array[0..127] of AnsiChar;    //описание устройства
   StateFlags: DWORD;   //флаги состояния (см. табл. 30.1)
end;
```

Таблица 30.1. *Флаги состояния записи* `TDisplayDeviceW`

Константа	Описание
DISPLAY_DEVICE_ATTACHED_TO_DESKTOP	Устройство отображает рабочий стол
DISPLAY_DEVICE_MIRRORING_DRIVER	Виртуальное устройство
DISPLAY_DEVICE_MODESPRUNED	Набор видеорежимов устройства превышает возможности оконечного устройства. Другими словами, дисплей не способен поддержать все режимы видеоадаптера
DISPLAY_DEVICE_PRIMARY_DEVICE	Основное устройство видеовывода
DISPLAY_DEVICE_REMOVABLE	Съемное устройство, которое не может являться основным дисплеем
DISPLAY_DEVICE_VGA_COMPATIBLE	VGA-совместимое устройство

В листинге 30.1 представлен пример, демонстрирующий способ сбора данных о видеоадаптерах в комбинированный список `ComboBox1`.

Листинг 30.1. Построение списка доступных видеоадаптеров

```
type pDisplayDeviceW=^TDisplayDeviceW;
var  pDD:pDisplayDeviceW;
     DevNum:longword;
begin
  DevNum:=0;
  New(pDD);
  pDD^.cb:=SizeOf(_Display_DeviceW); {!!! инициализация структуры !!!}
```

```
  while EnumDisplayDevices(Nil,DevNum,pDD^,0)=true do
  begin
     {не собираем сведения о виртуальных устройствах}
     if (pDD^.StateFlags AND DISPLAY_DEVICE_MIRRORING_DRIVER)<>
       DISPLAY_DEVICE_MIRRORING_DRIVER then
              ComboBox1.AddItem(pDD^.DeviceString,TObject(pDD));
     INC(DevNum); //готовим новую структуру
     New(pDD);
     pDD^.cb:=SizeOf(_Display_DeviceW); {!!! инициализация структуры !!!}
  end;
  Dispose(pDD); //последняя структура лишняя
end;
```

В результате выполнения кода в комбинированный список `ComboBox1` помещаются не только названия всех устройств, но и указатели на области памяти, в которых хранятся соответствующие устройствам структуры `TDisplayDeviceW`.

Получив сведения об установленных в компьютере устройствах видеовывода, стоит задуматься о получении их технических характеристик. В первую очередь следует собрать данные обо всех потенциально поддерживаемых устройством режимах работы. Для такой работы понадобится функция

```
function EnumDisplaySettings(DeviceName : PWideChar; ModeNum  : Cardinal;
                             var DevMode: TDeviceModeW) : boolean;
```

Параметр `DeviceName` представляет собой указатель на строку с именем устройства отображения (первичный адаптер обычно называется "`\\.\DISPLAY1`"). Впрочем, если логика программы предполагает только работу с основным видеоадаптером, то вместо имени в параметр передавайте `nil`. Кроме того, `DeviceName` вправе ссылаться на строку, полученную при посредничестве функции `EnumDisplayDevices()`. Второй по счету параметр `ModeNum` определяет порядковый номер интересующего нас режима. Для сбора всех имеющихся режимов отображения инициируйте `ModeNum` нулевым значением и (осуществляя пошаговое приращение) вызывайте функцию до тех пор, пока она не возвратит `false`. Как показывает опыт, для создания полной картины потребуется больше сотни обращений к функции `EnumDisplaySettings()`. Вся информация о режиме накапливается в параметре `DevMode`, описываемом структурой `TDeviceModeW`. В записи `TDeviceMode` более чем три десятка полей, но сейчас для нас наиболее значимы поля, отвечающие за глубину цвета, размеры области вывода и частоту смены кадров (листинг 30.2).

Листинг 30.2. Фрагмент объявления записи `TDeviceModeW`

```
type TDeviceModeW= packed record
 dmDeviceName: array[0..CCHDEVICENAME – 1] of WideChar; //имя устройства
 dmBitsPerPel: DWORD;        //глубина цветов в битах
 dmPelsWidth: DWORD;         //ширина
 dmPelsHeight: DWORD;        //высота
 dmDisplayFrequency: DWORD; //частота кадровой развертки
 dmOrientation: SHORT;       //угол поворота
 dmPosition: DWORD;{номер устройства при работе с несколькими мониторами}
 //... другие поля записи
end;
```

В листинге 30.3 предложен пример, демонстрирующий способ сбора допустимых режимов работы видеоустройства с именем DeviceName в список выбора ListBox1:TListBox.

Листинг 30.3. Сбор сведений о поддерживаемых режимах работы

```
procedure TForm1.GetVideoAdapterSettings(DeviceName: string);
var ModeNum :cardinal;
    DM:TDeviceMode;
    s:string;
begin
  ListBox1.Clear;
  ModeNum :=0;
  while EnumDisplaySettings(pWideChar(DeviceName),ModeNum ,DM)=true do
  begin
    s:=Format('%d x %d пикс. %d бит %d Гц',[DM.dmPelsWidth,
            DM.dmPelsHeight,DM.dmBitsPerPel,DM.dmDisplayFrequency]);
    ListBox1.AddItem(s,nil);
    INC(ModeNum);
  end;
end;
```

Изменение настроек дисплея

Обладая сведениями о поддерживаемых видеоадаптером режимах работы, мы без особых проблем сможем перевести дисплей в наиболее подходящее для нашей программы состояние. Для управления режимом работы основного устройства видеовывода компьютера предназначена функция

```
function ChangeDisplaySettings(DevMode: TDeviceMode;
                               Flags : Cardinal) : Integer;
```

Функция изменяет настройки дисплея, переводя его в назначенный с помощью первого аргумента режим работы. Значения описываются в формате структуры TDeviceMode, уже знакомой нам по функции EnumDisplaySettings(). Особо важен параметр Flags. Он представляет собой комбинацию флагов, приведенных в табл. 30.2. Если работа функции завершилась корректно, то она возвратит значение DISP_CHANGE_SUCCESSFUL.

***Таблица 30.2.** Значения флагов функции ChangeDisplaySettings()*

Значение	Описание
0	Дисплей динамически переходит в новый графический режим
CDS_FULLSCREEN	Временный полноэкранный режим. После завершения работы приложения, сменившего графический режим, система постарается возвратиться в исходное состояние
CDS_GLOBAL	Новые установки будут применены для всех пользователей компьютера. Флаг должен использоваться совместно с флагом CDS_UPDATEREGISTRY
CDS_NORESET	Данные о предлагаемом графическом режиме сохраняются в реестре, но экран продолжит работу в старом режиме до перезагрузки. Флаг используется только совместно с CDS_UPDATEREGISTRY
CDS_RESET	Новый режим применяется принудительно

Таблица 30.2 *(окончание)*

Значение	Описание
CDS_SET_PRIMARY	Выбранное устройство назначается первичным
CDS_TEST	Система только тестирует корректность графического режима без изменений работы видеоадаптера
CDS_UPDATEREGISTRY	Предложенный режим применяется немедленно с записью пользовательских настроек в реестре Windows

Предположим (рис. 30.1), что список ListBox1 заполнен всеми доступными для устройства режимами (в соответствии с примером работы с функцией EnumDisplaySettings() из листинга 30.3). В этом случае код, позволяющий изменять текущие настройки первичного видеоадаптера и подключенного к нему дисплея, может выглядеть так, как предложено в листинге 30.4.

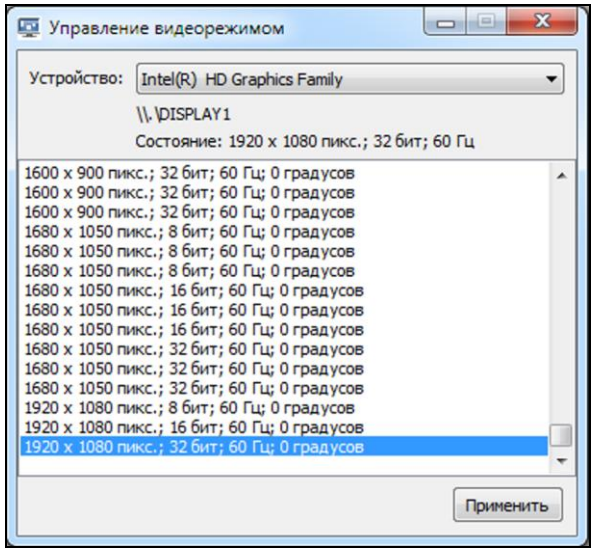


Рис. 30.1. Внешний вид приложения, управляющего видеорежимами

Листинг 30.4. Изменение режима работы видеоадаптера

```
var pDD : pDisplayDeviceW;
    DM  : TDeviceMode;
    s   : string;
begin
  pDD:=pDisplayDeviceW(ComboBox1.Items.Objects[ComboBox1.ItemIndex]);
  //индекс выбранного режима соответствует ListBox1.ItemIndex
  EnumDisplaySettings(pDD^.DeviceName,ListBox1.ItemIndex ,DM);

  s:=Format('Установить режим:'+#13+'%d x %d пикс. %d бит %d Гц ?',
                  [DM.dmPelsWidth,DM.dmPelsHeight,
                   DM.dmBitsPerPel, DM.dmDisplayFrequency]);
```

```
  if MessageBox(Form1.Handle,PChar(s),
      PChar('Применение режима'),MB_YESNO+MB_ICONQUESTION)=idYes then
      ChangeDisplaySettings(DM,0);
end;
```

Исследование текущего состояния устройства

Для того чтобы наше приложение смогло корректно работать с монитором, проектором, печатающим устройством или любым другим графическим устройством, следует научить его получать основные сведения об устройстве. В составе VCL Delphi предусмотрены классы, позволяющие собрать наиболее общие характеристики устройств видеовывода. Так, классы `TMonitor` и `TScreen` предоставят информацию о мониторе и экране компьютера, а класс `TPrinter` упростит работу с принтером. Но еще более подробную информацию предоставит функция

```
function GetDeviceCaps(DC : HDC; Index : Integer) : Integer;
```

Функция способна рассказать о трех десятках ключевых характеристиках устройства вывода (принтера, плоттера или экрана дисплея). Для этого в `GetDeviceCaps()` передается дескриптор контекста графического устройства `DC`. Второй параметр `Index` представляет собой один из почти трех десятков идентификаторов. В зависимости от его значения функция возвратит то или иное описание устройства.

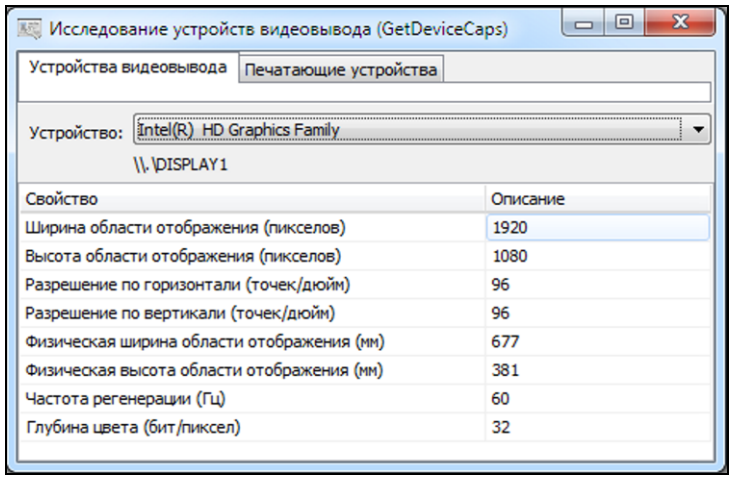


Рис. 30.2. Исследование характеристик видеоадаптера

Изучите рис. 30.2. На нем представлены основные технические характеристики видеоадаптера, полученные с помощью функции `GetDeviceCaps()`, в их числе: ширина и высота области вывода, разрешающая способность, глубина цвета и частота регенерации. Для того чтобы получить перечисленные данные, нам потребуется узнать дескриптор контекста графического устройства и передать его в функцию `GetDeviceCaps()`. Все это делает представленная в листинге 30.5 функция `GetVideoAdapterContextInfo()`.

Листинг 30.5. Пример работы получения параметров видеоадаптера

```
procedure TForm1.GetVideoAdapterContextInfo(DeviceName: string);
var IC:HDC; //дескриптор информационного контекста устройства
    i : integer;
begin
  IC:=CreateIC(pChar(DeviceName), nil, nil, nil);//информационный контекст

  with ValueListEditor1 do // ValueListEditor1:TValueListEditor
  begin
    Strings.BeginUpdate;
    Strings.Clear;
    i:=GetDeviceCaps(IC,HORZRES);
    InsertRow('Ширина области отображения (пикселов)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,VERTRES);
    InsertRow('Высота области отображения (пикселов)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,LOGPIXELSX);
    InsertRow('Разрешение по горизонтали (точек/дюйм)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,LOGPIXELSY);
    InsertRow('Разрешение по вертикали (точек/дюйм)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,HORZSIZE);
    InsertRow('Физическая ширина области (мм)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,VERTSIZE);
    InsertRow('Физическая высота области (мм)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,VREFRESH);
    InsertRow('Частота регенерации (Гц)',IntToStr(i),true);
    i:=GetDeviceCaps(IC,BITSPIXEL);
    InsertRow('Глубина цвета (бит/пиксел)',IntToStr(i),true);
    Strings.EndUpdate;
  end;
  DeleteDC(IC);
end;
```

Комментируя листинг 30.6, отметим, что на вход процедуры `GetVideoAdapterContextInfo()` должно поступать не физическое название адаптера, а заранее оговоренная аббревиатура. Например, для основного дисплея это "`\\.\DISPLAY1`", для дополнительного — "`\\.\DISPLAY2`".

Взаимодействие с экраном, класс *TScreen*

Процесс проектирования графического интерфейса пользователя в приложениях VCL Forms значительно упрощает класс `TScreen`. Экземпляр класса создается автоматически при запуске приложения и самостоятельно собирает информацию о ключевых характеристиках экрана, списке доступных шрифтов и курсоров, составе форм приложения, активной форме приложения и элементе управления, находящемся в фокусе ввода. Доступ к объекту производится благодаря глобальной переменной `Screen`. Практически все свойства экрана доступны только в режиме для чтения.

Информация о рабочем столе

Высота, ширина и координаты левого верхнего угла рабочего стола хранятся в свойствах:

```
property DesktopHeight: Integer;
property DesktopWidth: Integer;
property DesktopLeft: Integer;
property DesktopTop: Integer;
```

Высота экрана в пикселах:

```
property Height: Integer;
```

Ширина экрана в пикселах:

```
property Width: Integer;
```

Количество пикселов в одном дюйме по вертикали:

```
property PixelsPerInch: Integer;
```

Управление видом указателя мыши

Класс `TScreen` содержит информацию обо всех доступных системе указателях мыши.

```
property Cursors[Index: Integer]: HCursor;
```

Свойство возвратит совместимый с функциями Win API дескриптор курсора.

Для изменения текущего указателя стоит обратиться к свойству

```
property Cursor: TCursor;
```

Как правило, изменение вида указателя должно уведомлять пользователя о происходящих в системе событиях. Например, о занятости процессора красноречиво свидетельствуют песочные часы. Предположим, что одна из функций программы выполняет достаточно высокий объем вычислений и поэтому "притормаживает" систему. Для того чтобы пользователь понял это, просто измените вид указателя мыши на песочные часы, пока не отработала ресурсоемкая функция (листинг 30.6).

Листинг 30.6. Смена указателя мыши

```
var OldCursor : TCursor;
begin
  OldCursor:=Screen.Cursor;   //запомним текущий указатель
  Screen.Cursor:=crHourGlass; //песочные часы
  //...  действия программы
  Screen.Cursor:= OldCursor;  //восстановим указатель
end;
```

Информация о шрифтах системы

Список всех доступных шрифтов системы можно получить, обратившись к свойству

```
property Fonts: TStrings;
```

Так как шрифты хранятся в виде набора строк, то не представляет особого труда передать набор любому компоненту, обладающему свойством `Items` или `Lines`, например `ComboBox1.Items:=Screen.Fonts`.

Для обновления списка шрифтов лучше всего подойдет метод

```
procedure ResetFonts;
```

Для изменения шрифта, применяемого при выводе оперативной подсказки, меню и имен файлов в стандартных диалоговых окнах открытия/сохранения файла, предназначены свойства:

```
property HintFont: TFont;
property MenuFont: TFont;
property IconFont: TFont;
```

Информация о формах проекта

Экземпляр класса `TScreen` держит на контроле все отображаемые на экране компьютера формы.

Общее количество форм содержится в свойстве:

```
property FormCount: Integer;
```

Для обращения к конкретной форме передайте ее индекс в свойство

```
property Forms[Index: Integer]: TForm;
```

Для доступа к активной в данный момент форме и ее активному элементу управления посмотрите на содержимое свойств:

```
property ActiveForm: TForm;
property ActiveControl: TWinControl;
```

В момент смены активной формы и в момент смены активного элемента управления генерируются события

```
property OnActiveFormChange: TNotifyEvent;   //смена активной формы
property OnActiveControlChange: TNotifyEvent; //смена элемента управления
```

Если приложение содержит модули данных, то обратиться к ним возможно при помощи свойства

```
property DataModules[Index: Integer]: TDataModule;
```

Число модулей данных находится в свойстве

```
property DataModuleCount: Integer;
```

Вполне возможно, что проект вместо классических форм (экземпляров класса `TForm`) работает с другими потомками класса `TCustomForm`. В такой ситуации для доступа к формам стороннего разработчика понадобятся свойства

```
property ActiveCustomForm: TCustomForm;
property CustomForms[Index: Integer]: TCustomForm;
property CustomFormCount: Integer;
```

Если для размещения форм на рабочем столе были задействованы их свойства `Align`, то для временного отключения выравнивания можно воспользоваться методом

```
procedure DisableAlign;
```

Восстановление режима выравнивания осуществит метод

```
procedure EnableAlign;
```

Информация об устройствах видеовывода

Если компьютер сконфигурирован для одновременной работы с несколькими устройствами видеовывода (например, монитором и видеопроектором или с двумя мониторами), то объект `Screen` предоставит приложению все основные сведения об этих устройствах. О количестве устройств проинформирует свойство

```
property MonitorCount: Integer;
```

Доступ к каждому из мониторов согласно их индексу предложит свойство

```
property Monitors[Index: Integer]: TMonitor;
```

Группа однотипных методов

```
function MonitorFromPoint(const Point: TPoint;
              MonitorDefault: TMonitorDefaultTo = mdNearest): TMonitor;
function MonitorFromRect(const Rect: TRect;
              MonitorDefault: TMonitorDefaultTo = mdNearest): TMonitor;
function MonitorFromWindow(const Handle: THandle;
              MonitorDefault: TMonitorDefaultTo = mdNearest): TMonitor;
```

позволит дать ссылку на экземпляр монитора по координатам точки `Point`, или координатам прямоугольной области `Rect`, или по дескриптору `Handle` окна.

Реакция на события

Полезное достоинство экземпляра класса `TScreen` заключается в наличии двух обработчиков событий, значительно упрощающих разработку программ, в которых надо иметь возможность централизованного контроля изменений активной формы и активного элемента управления. Первую задачу решает событие

```
property OnActiveFormChange: TNotifyEvent;
```

Контроль изменений активного элемента управления стоит поручить событию

```
property OnActiveControlChange: TNotifyEvent;
```

Взаимодействие с дисплеем, класс *TMonitor*

Класс `TMonitor` описан в модуле `Forms`. Экземпляр класса упрощает организацию взаимодействия приложения и монитора (мониторов, если их несколько). Объект создается автоматически вместе со стартом приложения, для обращения к нему необходимо воспользоваться свойством `Monitors` глобального объекта `Screen`.

Все методы монитора унаследованы от класса `TObject`, поэтому с ними мы знакомиться не станем и сразу перейдем к свойствам (табл. 30.3).

***Таблица 30.3.** Основные параметры класса `TMonitor`*

Свойство	Описание
`property Handle: HMONITOR;`	Дескриптор монитора
`property MonitorNum: Integer;`	Номер монитора в списке `Monitors` глобального объекта `Screen`
`property Primary: Boolean;`	Состояние `true` сигнализирует о том, что монитор первичный

Таблица 30.3 *(окончание)*

Свойство	Описание
`property BoundsRect: TRect;`	Прямоугольная область монитора. Координаты левого верхнего угла соответствуют координатам (0, 0) первичного монитора
`property WorkareaRect: TRect;`	Размеры рабочей области монитора. В нее не входят панель задач и другие дополнительные панели
`property Left: Integer;`	Логическая позиция левой границы монитора
`property Top: Integer;`	Логическая позиция верхней границы монитора
`property Width: Integer;`	Ширина монитора в пикселах
`property Height: Integer;`	Высота монитора в пикселах

ГЛАВА 31

Холст *TCanvas*

В основу идеи графического механизма в ОС Microsoft Windows положена концепция аппаратно-независимого видеовывода. Это означает, что разрабатывающий прикладное программное обеспечение программист может не задумываться над тем, каким образом его приложение отправит данные о своем графическом интерфейсе видеоадаптеру, а тот, в свою очередь, "зажжет" необходимые пикселы на мониторе. Задача программиста заключается лишь в грамотном использовании в коде программы процедур и функций прорисовки графических примитивов (точек, линий, кривых, простейших геометрических фигур), заливки областей и вывода текста.

Для того чтобы приобрести возможность нарисовать линию или написать строку текста, программисту достаточно получить доступ к контексту нужного графического устройства и воспользоваться услугами стандартных функций двумерной графики Windows (Graphics Device Interface, GDI). Прямое обращение к функциям GDI для начинающего программиста достаточно затруднительно, поэтому разработчики Delphi создали уникальный класс `TCanvas`, который обеспечивает простой доступ к заданному контексту графического устройства и инкапсулирует наиболее востребованные функции GDI. Более подробно возможности класса `TCanvas` мы обсудим в следующей главе, но уже сейчас продемонстрируем его функциональность. Листинг 31.1 показывает, как мы получили возможность графического вывода на поверхности рабочего стола Windows.

Листинг 31.1. Доступ к контексту рабочего стола Windows

```
var Canvas: TCanvas;
begin
  Canvas:=TCanvas.Create;
  with Canvas do
  begin
    Canvas.Handle:=CreateDC(PChar('DISPLAY'),nil, nil, nil);
    Font.Size:=20;
    Font.Color:=clRed;
    TextOut(10,10,'Hello, world!');
  end;
  Canvas.Free;
end;
```

В коде примера создается экземпляр класса `TCanvas` и с помощью функции `CreateDC()` связывается с графическим контекстом экрана вашего компьютера. Факт успешного завершения эксперимента подтвердит надпись "Hello, world!" в левом верхнем углу рабочего стола.

Контекст графического устройства представляет собой весьма сложную структуру в памяти компьютера (в открытом пользовании нет ни одного официального документа Microsoft, описывающего структуру контекста Windows) и набор функций прикладного программного интерфейса, прямо или косвенно влияющих на процесс графического вывода. Ко всему прочему, контекст хранит сведения о своих текущих настройках — атрибутах контекста. В этой главе мы познакомимся с ключевыми атрибутами: кистью, пером и шрифтом. В объектно-ориентированном языке Delphi перечисленные атрибуты представлены в виде самостоятельных объектов — экземпляров класса `TBrush`, `TPen` и `TFont` соответственно. Но сначала мы поговорим о самом важном участнике всех графических операций — цвете.

Представление цвета

Любой из цветов радуги, который вы увидите на экране монитора, получается в результате сложения (в той или иной пропорции) всего-навсего трех базовых цветовых составляющих: красного, синего и зеленого цветов. Такую цветовую модель называют RGB, по первым буквам английских названий цветов (`Red`, `Green` и `Blue`).

Почему выбраны именно эти цвета? Все объясняется физиологическими особенностями строения человеческого глаза, он воспринимает цвета в диапазоне длин волн примерно 400—700 нм. Излучения с длинами волн 380—470 нм имеют фиолетовый и синий цвета, 480—500 нм — сине-зеленый, 510—560 нм — зеленый, 570—590 нм — желто-оранжевый, 600—760 нм — красный. Восприятие цвета глазом человека осуществляется за счет трех типов цветочувствительных фоторецепторов (колбочек). Существуют всего три типа этих фоторецепторов — колбочки, реагирующие на красный, зеленый или синий цвета. В зависимости от интенсивности той или иной цветовой составляющей соответствующий фоторецептор отправляет сигнал в наш головной мозг, где и формируется результирующая картинка.

Современное программное обеспечение ориентировано на работу с 24- или 32-разрядным представлением цвета (режим TrueColor). Для описания каждой цветовой составляющей отводится по 8 бит памяти. Если цвет 32-разрядный, то оставшийся четвертый байт хранит значение прозрачности (альфа-канал). Умножьте 4 байта на разрешение вашего рабочего стола, и вы получите весьма большое число. Например, для отображения одной картинки рабочего стола Windows на 19-дюймовом дисплее потребуется $1280 \times 1024 \times 4$ байта = = 5 242 880 байт (5 Мбайт) памяти.

В VCL на обслуживании цвета специализируется тип данных

```
type TColor = -$7FFFFFFF-1..$7FFFFFFF;
```

Это не что иное, как тип данных `Cardinal`, позволяющий описать 32-битное неотрицательное значение.

При назначении цвета можно воспользоваться системными константами, определяющими цвет в соответствии со стилем оформления Windows, назначить цвет вручную или обратиться к услугам макросов GDI. У каждого из перечисленных способов есть свои достоинства и недостатки.

Все системные константы объявлены в модуле `Graphics`, их имена начинаются с приставки `cl`, например, константа черного цвета называется `clBlack`, белого — `clWhite`. Есть и кон-

станты, хранящие настройки выбранной пользователем цветовой схемы. Например, константа clBtnFace помнит значение цвета поверхности кнопки, а константа clCaptionText — цвет, которым выводятся надписи в заголовках окна. По объективным причинам число системных констант весьма ограничено, поэтому они не в состоянии перекрыть весь цветовой спектр.

Для определения цвета из кода программы достаточно вспомнить, что каждая из цветовых составляющих должна находиться в пределах от 0 до 255 (от $00 до $FF в шестнадцатеричном представлении). Все остальное — дело техники. Например, для создания кисти яркого зеленого цвета достаточно воспользоваться значением $00FF00, красного — $0000FF, белого — $FFFFFF и т. п. Как видите, это очень быстрый способ, который позволит определить любой цвет, однако он не вполне удобен для человека. Попробуйте определить, какому цвету соответствует значение $F0CAA6? Получилось? Правильный ответ — небесно-голубой...

Третий способ определения значения цвета опирается на четверку макросов (табл. 31.1). Их основное достоинство заключается в простоте использования и наглядности. Кроме того, макросы незаменимы в том случае, когда мы намерены управлять цветами в динамике.

Таблица 31.1. *Макросы для работы с цветом*

Макрос	Назначение
Function RGB(Red,Green,Blue : Byte): TColor;	Макрос формирует результирующий цвет на основе сложения трех составляющих
Function GetRValue(Color : TColor) : Byte;	Группа макросов, решающих обратную задачу, они выделяют из смешанного цвета отдельную цветовую составляющую
Function GetGValue(Color : TColor) : Byte;	
Function GetBValue(Color : TColor) : Byte;	

Разместите на форме три компонента TTrackBar, передайте в свойство Max каждого из элементов управления значение 255. Наиболее прозорливый читатель уже наверняка догадался, что, изменяя положение ползунка компонента TTrackBar, мы станем управлять одной из цветовых составляющих. Для того чтобы это реализовать в коде, выберите событие OnChange() любого из элементов управления и напишите в нем строки кода, предложенные в листинге 31.2.

Листинг 31.2. Пример работы с макросом RGB()

```
procedure TForm1.TrackBar1Change(Sender: TObject);
var R,G,B:Byte;
begin
  R:= TrackBar1.Position;
  G:= TrackBar2.Position;
  B:= TrackBar3.Position;
  Form1.Color:=RGB(R,G,B);
end;
```

Не забудьте сделать событие OnChange() общим для всех компонентов TTrackBar и запускайте проект на выполнение. Перемещение любого из ползунков приведет к изменению цвета поверхности формы.

Кисть *TBrush*

Класс `TBrush` предназначен для описания кисти, с помощью которой осуществляется закраска указанной программистом области. Перед проведением "малярных работ" следует настроить важнейшие характеристики кисти. Цвет кисти устанавливается в свойстве

```
property Color: TColor;
```

Стиль кисти

```
property Style : TBrushStyle;
```

отвечает за узор, которым покроется закрашиваемая поверхность. Предусмотрены 8 вариантов стандартных узоров кисти (табл. 31.2).

Таблица 31.2. *Константы штриховки кисти* `TBrushStyle`

Константа	Описание
bsSolid	Сплошная кисть
bsClear	Пустая кисть
bsHorizontal	Горизонтальная штриховка
bsVertical	Вертикальная штриховка
bsFDiagonal	Штриховка с наклоном 45 градусов, линии направлены из левого верхнего угла в правый нижний угол закрашиваемой области
bsBDiagonal	Штриховка с углом 45 градусов, линии направлены из правого верхнего угла в левый нижний угол
bsCross	Штриховка из комбинации горизонтальных и вертикальных линий
bsDiagCross	Диагональная штриховка из наклонных линий

На внешний вид кисти оказывает влияние цвет заполнения фона. В представленном в листинге 31.3 примере мы закрашиваем два ряда прямоугольных областей, изменяя не только цвет и стиль кисти, но и цвет заполнения с помощью функции Windows API `SetBkColor()`. Верхний ряд выводится белой кистью с черным фоном, нижний — наоборот, черной кистью с белым фоном.

Листинг 31.3. Управление стилем кисти

```
procedure TForm1.FormPaint(Sender: TObject);
var Rct: TRect;
    BS : TBrushStyle;
begin
  Form1.Color:=clWhite; //клиентская часть формы окрашена в белый цвет
  with Form1.Canvas do
  begin
    Rct:=RECT(5,15,55,65);
    for BS:=Low(TBrushStyle) to High(TBrushStyle) do {верхний ряд}
    begin
      Brush.Style:=BS;                          {изменяем стиль кисти}
      Brush.Color:=clWhite;                     {цвет кисти белый}
      SetBkColor(Form1.Canvas.Handle,clBlack);{устанавливаем черный фон}
```

```
      FillRect(Rct);                          {закрашиваем область}
      OffsetRect(Rct,55,0);                   {сдвигаем область вправо}
    end;
    Rct:=RECT(5,70,55,120);
    for BS:=Low(TBrushStyle) to High(TBrushStyle) do {нижний ряд}
    begin
      Brush.Style:=BS;                        {изменяем стиль кисти}
      Brush.Color:=clBlack;                   {цвет кисти черный}
      SetBkColor(Form1.Canvas.Handle,Form1.Color);{устанавливаем
                                                  белый фон}
      FillRect(Rct);                          {закрашиваем область}
      OffsetRect(Rct,55,0);                   {сдвигаем область вправо}
    end;
  end;
end;
```

Результат работы программы представлен на рис. 31.1.

Рис. 31.1. Варианты закраски областей кистью различного стиля

Программист может создавать кисти с узором, созданным на базе битового образа. Для этого необходимо загрузить файл с картинкой в свойство

```
property Bitmap: TBitmap;
```

Имейте в виду, что по умолчанию Delphi не инициализирует свойство `Bitmap` кисти, поэтому перед загрузкой растрового рисунка необходимо создать экземпляр объекта `TBitmap` и присоединить его к кисти так, как представлено в листинге 31.4.

Листинг 31.4. Создание кисти из растровой картинки

```
with Form1.Canvas do
begin
  Brush.Bitmap:=TBitmap.Create;           //инициализация свойства
  Brush.Bitmap.LoadFromFile('mybmp.bmp'); //загрузка шаблона кисти
  FillRect(Form1.ClientRect);             //закраска поверхности
  Brush.Bitmap.Free;                      //очистка ресурса
end;
```

Завершая рассказ, посвященный кисти, упомянем свойство, в котором находится дескриптор кисти:

```
property Handle: HBRUSH;
```

Перо *TPen*

Перо предназначено для определения характеристик линий, рисуемых на поверхности графического устройства. Язык Delphi позволяет создавать перья двух базовых разновидностей.

- *Косметическое перо*, описываемое тремя параметрами: цветом, стилем линии и толщиной. Это перо как нельзя лучше подходит для черчения простых тонких линий и кривых, и его услуг вполне достаточно в 9 из 10 случаев.
- *Геометрическое перо* развивает возможности своего "коллеги" — оно дополнено еще четырьмя атрибутами: узором, типом наконечника, типом соединения (стыком) и штриховкой. Одна из существенных особенностей геометрического пера в том, что проведенные им линии чувствительны к двумерным аффинным преобразованиям, поэтому при работе приложения в мировых координатах геометрические перья превосходят своих косметических "коллег".

> ***Замечание***
>
> По умолчанию на базе класса `TPen` формируется простое косметическое перо, для создания более совершенного геометрического пера необходимо обратиться за помощью к функции Windows API `ExtCreatePen()`.

После вызова конструктора по умолчанию создается сплошное черное перо толщиной в 1 пиксел. Если атрибуты пера по умолчанию не соответствуют пожеланиям разработчика, то к его услугам предоставлены три свойства: цвет, толщина и стиль. Цвет пера определяется в свойстве

```
property Color: TColor;
```

Толщина линии в логических единицах:

```
property Width: Integer
```

Стиль пера выбирается из перечня стилей

```
property Style: TPenStyle;
```

Предусмотрены 9 стилей пера (табл. 31.3).

Таблица 31.3. *Стили пера* `TPenStyle`

Константа	Описание
`psSolid`	Сплошная линия
`psDash`	Линия, состоящая из отрезков
`psDot`	Линия точек
`psDashDot`	Линия из отрезков и точек
`psDashDotDot`	Отрезок + две точки
`psClear`	Невидимая линия
`psInsideFrame`	Сплошная линия
`psUserStyle`	Пользовательский стиль
`psAlternate`	Стиль используется только косметическим пером, он определяется набором пикселов

Первые семь стилей представляют собой базовый набор для обычного косметического пера. Использование перечисленных стилей (за исключением psSolid и psInsideFrame) имеет смысл только при толщине пера не более 1 пиксела. В противном случае вы получите не штриховую, а обычную сплошную линию. Особых комментариев достойны стили psClear и psInsideFrame. Стиль psClear соответствует пустому (невидимому) перу — рисование отключается. Стиль psInsideFrame обычно применяется при выводе простейших геометрических фигур пером шириной более 1 пиксела. Дело в том, что при использовании утолщенного пера (Style<>psInsideFrame) границы фигуры расширяются относительно заданных координат. Если вы нарисуете прямоугольник с координатами левого верхнего угла (10, 10) и правого нижнего (100, 100) пером psSolid толщиной 10 пикселов, действительный верхний угол окажется в точке (1, 1), а нижний — в точке (109, 109). Границы фигуры расширятся ровно на ширину пера. Установка текущего пера в состояние psInsideFrame запретит расширяться границам фигуры — линия будет утолщена вовнутрь. Листинг 31.5 демонстрирует порядок управления стилями пера при рисовании обычного отрезка.

Листинг 31.5. Управление стилем пера при рисовании отрезка

```
var y :integer;
    ps:TPenStyle;
begin
  with Form1.Canvas do
  begin
    y:=20;
    for PS:=Low(TPenStyle) to High(TPenStyle) do
    begin
      TextOut(10,y,IntToStr(Integer(PS)));
      Pen.Style:=PS;
      MoveTo(20,y);
      LineTo(200,y);
      Inc(y,20);
    end;
  end;
end;
```

Результаты выполнения кода отражены на рис. 31.5. Обратите внимание на то, что линии, нарисованные пером с двумя последними стилями (psUserStyle=7 и psAlternate=8), представляют собой обычную сплошную линию. Дело в том, что перечисленные стили предна-

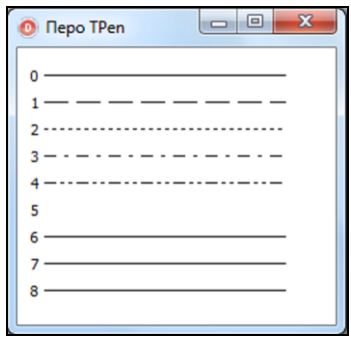


Рис. 31.2. Линии, которые рисует косметическое перо

значены для работы совместно с функцией Windows API `ExtCreatePen()` и в обычном режиме работы никакого смысла не имеют.

Для создания геометрического пера следует обратиться к функции Win API

```
function ExtCreatePen(PenStyle : Cardinal; Width : Cardinal;
                      const LogBrush : TLogBrush; StyleCount : Cardinal;
                      Style : pointer) : HPEN;
```

Вид пера определяется в первом параметре функции — геометрическому перу соответствует константа `PS_GEOMETRIC`. Ко всему прочему, в него передается константа, которая описывает наконечник пера, определяющего способ прорисовки окончания линий, и константа, назначающая способ соединения (стык) между линиями. Ширину пера определяет аргумент `Width`. Более подробные сведения о функции вы сможете подчерпнуть в документации MS Windows SDK, а сейчас рассмотрим пример (листинг 31.6) создания геометрического пера.

Листинг 31.6. Создание геометрического пера

```
var LogBrush:TLogBrush;
begin
  //1 вариант пера
  LogBrush.lbColor:=clRed;    {цвет кисти — красный}
  LogBrush.lbStyle:=BS_SOLID; {стиль кисти — сплошная кисть}
  form1.Canvas.Pen.Handle:=
      ExtCreatePen(PS_GEOMETRIC OR PS_ENDCAP_ROUND OR PS_SOLID,
                   19, LogBrush, 0, nil);
  form1.Canvas.MoveTo(10,20); form1.Canvas.LineTo(200,20);

  //2 вариант пера
  LogBrush.lbColor:=clBlue;       {цвет кисти — синий}
  LogBrush.lbStyle:=BS_HATCHED;   {стиль — узор}
  LogBrush.lbHatch:=HS_DIAGCROSS; {вид узора — диагональные линии}
  form1.Canvas.Pen.Handle:=ExtCreatePen(PS_GEOMETRIC OR PS_ENDCAP_SQUARE,
                                        19, LogBrush, 0, nil);
  form1.Canvas.MoveTo(10,50); form1.Canvas.LineTo(200,50);
end;
```

В листинге 31.6 последовательно создаются два пера. Но теперь, для того чтобы передать их в распоряжение холста Delphi, нам потребуется не только помощь функции `ExtCreatePen()`, но и свойство, в которое мы направим дескриптор нового пера:

```
property Handle: HPen;
```

Результаты выполнения листинга отображены на рис. 31.3. Обратите внимание на ряд уникальных особенностей отрезков, нарисованных геометрическими перьями. Во-первых, концы верхнего отрезка закруглены, этого результата мы добились благодаря константе `PS_ENDCAP_ROUND`, переданной в первый параметр функции `ExtCreatePen()`. Во-вторых, благодаря изменению стиля кисти `LogBrush` нижний отрезок заполнен диагональным узором.

Создавая перо, мы имеем возможность задать режим прорисовки линии на холсте

```
property Mode: TPenMode;

type TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy,
    pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge,
    pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor);
```

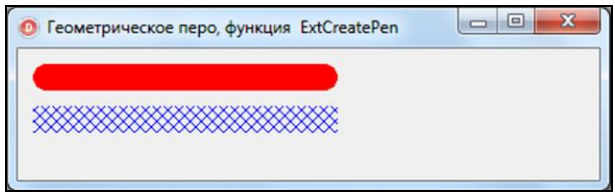


Рис. 31.3. Линии, проведенные геометрическими перьями

Представление цветной линии в таком случае определяется не только цветом пера, но и цветом области холста (по которой проходит линия) и характером сложения этих цветов. Эта операция называется *бинарной растровой операцией*, о ней мы подробнее поговорим в *главе 33*.

Шрифт *TFont*

Все элементы управления из палитры компонентов Delphi, способные выводить текст и базовый графический класс `TCanvas`, обладают свойством `Font`. Это свойство предоставляет доступ к очередному важному атрибуту контекста графического устройства — объекту `TFont` (шрифту).

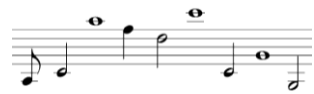
Самое главное свойство шрифта содержится в свойстве

```
property Name: TFontName;
```

Здесь определяется имя шрифта, а точнее, имя семейства шрифтов. Семейство — это группа объединенных одним названием (гарнитурой) шрифтов со сходными характеристиками. Так в семейство одного из самых распространенных шрифтов гарнитуры Arial входят: Arial, **Arial Bold**, ***Arial Bold Italic*** и *Arial Italic*. Заметьте, что имена физических шрифтов формируются из названия гарнитуры и начертания шрифта.

Наличие разных семейств шрифтов предполагает существование классификации шрифтов по принадлежности к семейству. Такая классификация предложена в табл. 31.4.

Таблица 31.4. *Классификация семейств шрифтов*

Семейство	Описание	Пример
Decorative	Оформительский шрифт	Old English
Dontcare	Шрифт с произвольными характеристиками	
Modern	Шрифт с одинаковой шириной символов	Courier New
Roman	Пропорциональный шрифт, обладающий переменной шириной символов и засечками	Times New Roman
Script	Векторный шрифт, имитирующий рукописный текст	*Script*
Swiss	Пропорциональный шрифт без засечек, с переменной шириной символов	Arial

ЗАМЕЧАНИЕ

Начиная с Delphi 2005, шрифт по умолчанию принадлежит семейству Tahoma, до этого (в Delphi 1.0, ..., Delphi 7.0) в качестве шрифта по умолчанию назначался MS Sans Serif.

Описание свойств класса шрифтов предложено в табл. 31.5

Таблица 31.5. *Свойства шрифта* `TFont`

Свойство	Описание
`property Charset: TFontCharset;`	Номер набора символов шрифта, по умолчанию соответствует `DEFAULT_CHARSET`, русская кириллица — `RUSSIAN_CHARSET`
`property Color: TColor;`	Цвет шрифта
`property FontAdapter: IChangeNotifier;`	Автоматически информирует объекты ActiveX о шрифте
`property Handle: HFont;`	Дескриптор шрифта для работы с функциями Win API
`property Height: Integer;`	Высота шрифта в пикселах
`property Size: Integer;`	Высота шрифта в пунктах Windows
`property Pitch: TFontPitch;` `type TFontPitch = (fpDefault, fpVariable, fpFixed);`	Способ установки ширины символов: • `fpDefault` — по умолчанию; • `fpFixed` — у всех ширина одинакова; • `fpVariable` — переменная ширина
`property PixelsPerInch: Integer;`	Число точек на дюйм. Используется Windows для установления соответствия между изображениями шрифта на экране и принтере
`property Orientation: Integer;`	Угол поворота текстовой строки в градусах относительно оси *x*
`property Style: TFontStyles;` `type TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);`	Способ начертания шрифта: жирный, курсив, подчеркнутый и перечеркнутый
`property Quality: TFontQuality;` `type TFontQuality = (fqDefault, fqDraft, fqProof, fqNonAntialiased, fqAntialiased, fqClearType, fqClearTypeNatural);`	Определяет требования к качеству вывода на экран текстовых надписей

ЗАМЕЧАНИЕ

Размер шрифта можно изменять при помощи свойств `Height` или `Size`. Свойства связаны формулой: `Font.Size = -Font.Height × 72 / Font.PixelsPerInch`.

Приведенный в листинг 31.7 код демонстрирует способы сбора информации об установленных в системе шрифтах в список `ListBox1`.

Листинг 31.7. Сбор сведений об установленных шрифтах

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Assign(Screen.Fonts);
  ListBox1.style:=lbOwnerDrawFixed;
end;
```

Для того чтобы предоставить пользователю возможность просмотреть, как будет выглядеть текстовая строка, выведенная шрифтом той или иной гарнитуры, воспользуемся листингом 31.8.

Листинг 31.8. Просмотр особенностей начертания шрифта

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var R :TRect;
    OldFontName:TFontName;
const S ='Abcdef...';
begin
  with ListBox1.Canvas do
  begin
    OldFontName:=Font.Name;
    FillRect(Rect);

    Font.Name:=ListBox1.Items.Strings[index];
    Font.Color:=clNavy;
    R:=Rect; R.Right:=TextWidth(S);
    TextRect(R, R.Left, R.Top, s);

    Font.Name:=OldFontName;
    Font.Color:=clWindowText;
    Rect.Left:=90;
    TextRect(Rect, Rect.Left+10, Rect.top, ListBox1.Items.Strings[index]);
  end;
end;
```

В листинге 31.8 описана реакция компонента на событие `OnDrawItem()`, в котором осуществляется перерисовка каждого из элементов списка. Обратите внимание на то, что кроме гарнитуры шрифта в элементе списка выводится демонстрационная строка, дающая представление об особенностях начертания глифов (рис. 31.4).

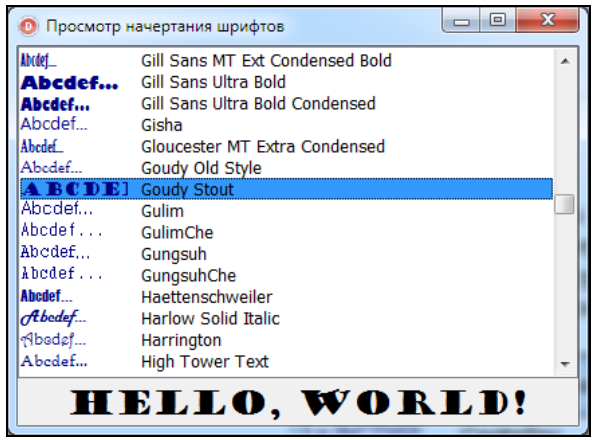


Рис. 31.4. Сбор сведений об установленных шрифтах

Холст *TCanvas* в VCL

С целью максимального упрощения работы с графикой и освобождения начинающего программиста от необходимости работать со сложными графическими функциями Windows API в состав Delphi включен класс `TCanvas`. Холст (именно так переводится слово Canvas с английского) описан в модуле `Graphics` и представляет собой объектно-ориентированное воплощение контекста графического устройства Windows. Судите сами: с одной стороны, объект `TCanvas` является собственно контекстом графического устройства, а с другой, холст осуществляет все графические операции при посредничестве классических свойств и методов.

С точки зрения цепочки наследования класс `TCanvas` находится в самой верхушке иерархии VCL, он является прямым потомком класса `TPersistent`. Несмотря на такую близость к праотцам, `TCanvas` — уже не абстрактный класс и в состоянии создавать полноценные объекты. Столь полезная способность позволяет разработчикам компонентов внедрять холст в экземпляры собственных объектов, таким образом, наделяя их почти всем спектром возможностей GDI.

> ***Замечание***
>
> Холст присутствует в большинстве визуальных компонентах Delphi, доступ к нему осуществляется посредством опубликованного у них свойства `Canvas`. Кроме того, в VCL имеется специальный объект, готовый исполнять обязанности холста. Это компонент `TPaintBox`.

Экземпляр класса `TCanvas` никогда не забывает, что он в проектах Delphi представляет интересы реального контекста графического устройства. Дополнительное напоминание об этом можно обнаружить в свойстве, хранящем дескриптор контекста:

```
property Handle: HDC;
```

В данной главе это свойство нам не понадобится, но позднее (в *главе 33*, в которой мы рассмотрим сложные приемы работы с контекстом) дескриптор пригодится нам для обращения к тем функциям Windows API, которым в описании класса `TCanvas` по тем или иным причинам не нашлось места.

Доступ к основным атрибутам холста обеспечивают свойства:

```
property Brush: TBrush; //кисть
property Pen: TPen;     //перо
property Font: TFont;   //шрифт
```

Закраска области

Среди всех методов холста, способных закрасить текущей кистью область, наибольшей скоростью обладает процедура

```
procedure FillRect(const Rect: TRect);
```

У метода `FillRect()` всего один недостаток — он умеет работать только с прямоугольными областями, представленными в форме структуры `Rect`.

Более значительными возможностями обладает метод

```
procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);
```

Это метод с зачатками интеллекта, он способен залить кистью область произвольной формы. Заливка начинается из точки (`X`, `Y`). Режим заливки определяется значением параметра

FillStyle. Если параметр принимает значение fsSurface, то заливка осуществляется до тех пор, пока вокруг точки (X, Y) не останется точек с цветом Color. В альтернативном режиме fsBorder заливка прекратится при выходе на границу с цветом Color.

Пример работы обеих функций закраски областей предложен в листинге 31.9. В рамках события перерисовки формы OnPaint() мы осуществляем заливку всей клиентской области формы сплошной кистью белого цвета. Затем в событии OnMouseDown(), генерируемом в момент нажатия кнопки мыши, над поверхностью формы инициируем вызов интеллектуального метода FloodFill(), осуществляющего заливку области, которой принадлежит точка с координатами (X, Y).

Листинг 31.9. Закраска области методами FillRect() и FloodFill()

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Form1.Canvas do
  begin
    Brush.Style:=bsSolid;
    Brush.Color:=clWhite;
    FillRect(Form1.ClientRect); //закраска формы
    //делим форму на части эллипсом и двумя диагональными линиями
    Pen.Color:=clBlack; //черный цвет пера
    Ellipse(Form1.ClientRect);
    MoveTo(0,0); LineTo(Form1.ClientWidth,Form1.ClientHeight);
    MoveTo(Form1.ClientWidth,0); LineTo(0,Form1.ClientHeight);
  end;
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Randomize;
  with Form1.Canvas do
  begin
    Brush.Style:=TBrushStyle(Random(8));
    Brush.Color:=RGB(Random(255),Random(255),Random(255));
    FloodFill(X, Y, clBlack, fsBorder);
  end;
end;
```

Для большей наглядности перед вызовом метода FloodFill() мы присваиваем кисти случайное значение цвета и заливки.

Градиентная заливка

Начиная с Delphi 2005, в составе этой среды разработки появилась процедура, позволяющая осуществлять градиентную заливку прямоугольной области:

```
procedure GradientFillCanvas(const ACanvas: TCanvas;
                             const AStartColor, AEndColor: TColor;
                             const ARect: TRect;
                             const Direction: TGradientDirection);
```

Сразу заметим, что это не метод класса TCanvas, а самостоятельная процедура, объявление которой вы обнаружите в модуле GraphUtil. Она требует указать холст ACanvas, на котором будет проводиться операция, стартовое AStartColor и конечное AEndColor значения цвета градиентной заливки, координаты обслуживаемой области ARect. Последний параметр функции Direction определяет направление заливки gdHorizontal (горизонтальное) либо gdVertical (вертикальное).

Листинг 31.10. Пример градиентной заливки клиентской области формы

```
uses GraphUtil;
...
GradientFillCanvas(Application.MainForm.Canvas,
                   clRed,clWhite,
                   Application.MainForm.ClientRect,gdVertical);
```

Предложенная в листинге 31.10 строка кода демонстрирует способ градиентной заливки клиентской области главной формы приложения.

Графические примитивы

Класс TCanvas обладает существенным набором методов для рисования разнообразных графических примитивов. Но самый простейший способ рисования обеспечивает не метод, а свойство

```
property Pixels[X, Y: Integer]: TColor;
```

Благодаря Pixels мы получим доступ к любой точке холста по ее координатам X и Y, как к элементу двумерного массива. Свойство позволяет выяснить, какой цвет установлен в заданной точке холста, и назначить требуемой точке необходимый цвет.

Прямо скажем, что закраска точек — достаточно трудоемкий и медленный процесс, и имеются другие, более расторопные способы графического вывода. Однако в некоторых случаях свойство Pixels может оказаться незаменимым в ситуации, когда требуется построить график сложной функции. Один из примеров подобных функций отражает листинг 31.11, он демонстрирует порядок поточечного вывода на поверхности формы геометрической фигуры гипоциклоиды. Не станем вдаваться в математические тонкости прорисовки кривой со столь мудреным названием, отметим лишь то, что внешний вид гипоциклоиды зависит от значения, возвращаемого свойством Position компонента TrackBar1.

Листинг 31.11. Применение свойства Pixels для построения гипоциклоиды

```
procedure TForm1.FormPaint(Sender: TObject);
var X,Y,K,R1,R2,T:real;
    offsetX,offsetY:Integer;
begin
  //находим центр клиентской области формы
  offsetX:=Form1.ClientWidth div 2;
  offsetY:=Form1.ClientHeight div 2;
  //подбираем радиус ограничивающей окружности
  if Form1.ClientWidth>Form1.ClientHeight then
    R1:=Form1.ClientHeight div 2 - 40
  else R1:=Form1.ClientWidth div 2 - 40;
```

```
  R2:=R1/TrackBar1.Position; // TrackBar1.Min=3
  K:=R1/R2;
  T:=0;
  while T<=2*pi do //цикл выполняется, пока T меньше и равно 2П
  begin
    X:=R2*(K-1)*(cos(t)+cos((K-1)*t)/(K-1)); //координата X
    Y:=R2*(K-1)*(sin(t)-sin((K-1)*t)/(K-1)); //координата Y
    Form1.Canvas.Pixels[Round(X)+offsetX, Round(Y)+offsetY]:=clBlack;
    T:=T+0.005;
  end;
end;
```

Результаты наших трудов представлены на рис. 31.5. Изменив положение ползунка, вы придадите гипоциклоиде новую форму.

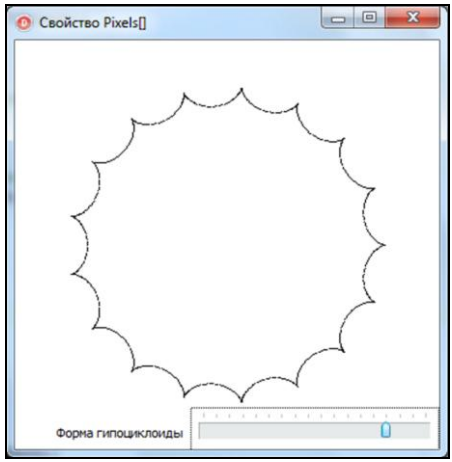


Рис. 31.5. Внешний вид гипоциклоиды

Линии

Хотя свойство холста `Pixels[]` наиболее универсально и позволяет нарисовать изображение любой степени сложности, но с другой стороны это чрезмерно трудоемкий и медленный процесс, и им не стоит злоупотреблять. Тем более в классе `TCanvas` собран обширный набор методов черчения линий и простейших геометрических фигур.

Для того чтобы вывести простейшую прямую, понадобятся два метода холста:

```
procedure MoveTo(X, Y: Integer);
procedure LineTo(X, Y: Integer);
```

Процедура `MoveTo()` позиционирует перо на начальную точку отрезка. Метод `LineTo()` проводит отрезок из текущей точки к конечной (причем конечная точка не закрашивается) и оставляет перо в этой точке. Если вы вновь вызовите метод `LineTo()`, то вывод линии начнется из последней точки.

Если требуется выяснить текущую позицию пера, то следует познакомиться со свойством

```
property PenPos : TPoint;
```

Свойство доступно не только для чтения, но и для записи, так что это своего рода замена метода `MoveTo()` для перемещения пера по холсту.

Если логика программы предполагает построение нескольких соединенных под разным углом друг к другу отрезков, то стоит обратиться к методу

```
procedure Polyline(Points: array of TPoint);
```

Для изображения ломаной линии необходимо заполнить массив координат точек, через которые пройдет линия. Начальной и конечной точкам ломаной линии соответствуют первый и последний элементы массива `Points`.

Ближайший родственник ломаной линии — многоугольник:

```
procedure Polygon(Points: array of TPoint);
```

Отличие рассмотренного метода от процедуры `Polyline()` заключается в том, что последняя объявленная в массиве точка соединяется линией с первой и область внутри многоугольника закрасится текущей кистью.

> ***ЗАМЕЧАНИЕ***
>
> Если вы планируете вывести на экран не весь массив точек, а только его часть, то воспользуйтесь функцией `Slice()`. Второй параметр функции ограничивает верхнюю границу массива: `Form1.Canvas.Polygon(Slice(Arr, 10));`.

Простейшие геометрические фигуры

В перечень простейших геометрических фигур входят: прямоугольник, эллипс и прямоугольник со скругленными углами.

Для рисования прямоугольника проще всего воспользоваться методом

```
procedure Rectangle(Left, Top ,Right ,Bottom: Integer); overload;
procedure Rectangle(const Rect: TRect); overload;
```

Процедура является перегружаемой и в качестве параметров принимает координаты левого верхнего (`Left`, `Top`) и правого нижнего (`Right`, `Bottom`) углов прямоугольника или координаты, заданные в формате прямоугольной структуры `TRect`.

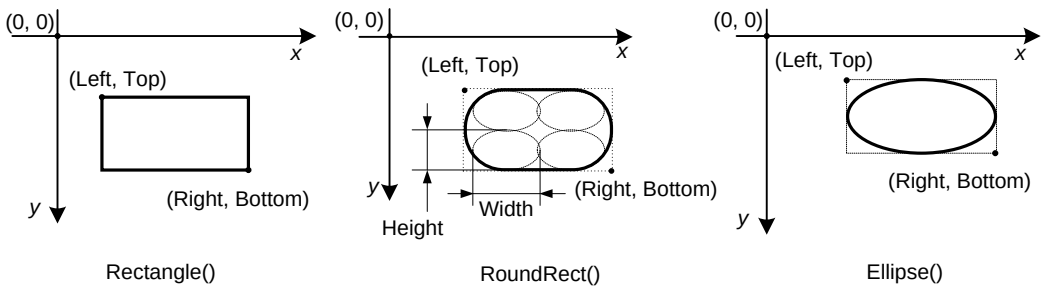


Рис. 31.6. Порядок вывода простейших геометрических фигур

Надо сказать, что большинство методов, рисующих простейшие фигуры, тесно связано с прямоугольником (рис. 31.6). Например: эллипс вписывается в рамки прямоугольника с координатами, аналогичными процедуре `Rectangle()`.

```
procedure Ellipse(Left, Top ,Right ,Bottom: Integer); overload;
procedure Ellipse(const Rect: TRect); overload;
```

Прямоугольник с более эстетичным контуром получится при использовании метода для рисования прямоугольников со скругленными углами:

```
procedure RoundRect(Left, Top ,Right ,Bottom, Width, Height: Integer);
```

Первые четыре параметра соответствуют координатам вершин функции `Rectangle()`. Значения `Width` и `Height` определяют степень сглаживания углов, задавая ширину и высоту скругляющего эллипса.

> ***Замечание***
>
> Методы `Rectangle()`, `Ellipse()`, `RoundRect()` и рассмотренные немного ранее `Chord()` и `Pie()` предназначены не только для рисования простейших геометрических фигур, но и для закрашивания ограниченной этими фигурами области текущей кистью.

На этом методы для работы с прямоугольной областью не заканчиваются. Контур толщиной в 1 пиксел для области `Rect` нарисует процедура

```
procedure FrameRect(const Rect: TRect);
```

Оконтуривание осуществляется не пером, а текущей кистью.

Эффект наличия фокуса ввода на прямоугольнике `Rect` обеспечивает метод

```
procedure DrawFocusRect(const Rect: TRect);
```

При повторном вызове процедуры фокус ввода снимается — изображение приводится к первоначальному виду.

Дуги

Концепция построения всех дуг основана на идее вывода части эллипса.

```
procedure Arc  (Left, Top, Right, Bottom, X3, Y3, X4, Y4 : Integer);
procedure Chord(Left, Top, Right, Bottom, X3, Y3, X4, Y4 : Integer);
procedure Pie  (Left, Top, Right, Bottom, X3, Y3, X4, Y4 : Longint);
```

Процедура `Arc()` чертит сегмент, `Chord()` — хорду, `Pie()` — сектор эллипса. Суть происходящего отражает рис. 31.7. Все изображения вписываются в эллипс (а точнее, в прямоугольник, содержащий эллипс) с координатами вершин (`Left`, `Top`) и (`Right`, `Bottom`). Логические координаты (`X3`, `Y3`) и (`X4`, `Y4`) описывают воображаемые лучи, исходящие из центра эллипса. Места пересечения лучей с контуром эллипса соединяются дугой. Начальной точкой дуги считается пересечение эллипса с лучом (`X3`, `Y3`), конечной — (`X4`, `Y4`).

Сплайн Безье

Программы инженерной графики, редакторы векторной графики, системы автоматизированного проектирования, издательские пакеты, приложения мультимедиа, игры, одним словом — современное программное обеспечение не в состоянии решить свои задачи за счет ограниченного перечня функций, рисующих прямые линии и дуги. Попробуйте начертить коленчатый вал или контуры истребителя средствами пусть даже очень хорошей функции построения линий! Поэтому в составе методов класса `TCanvas` имеется пара методов, способных нарисовать кривую линию практически любой степени сложности.

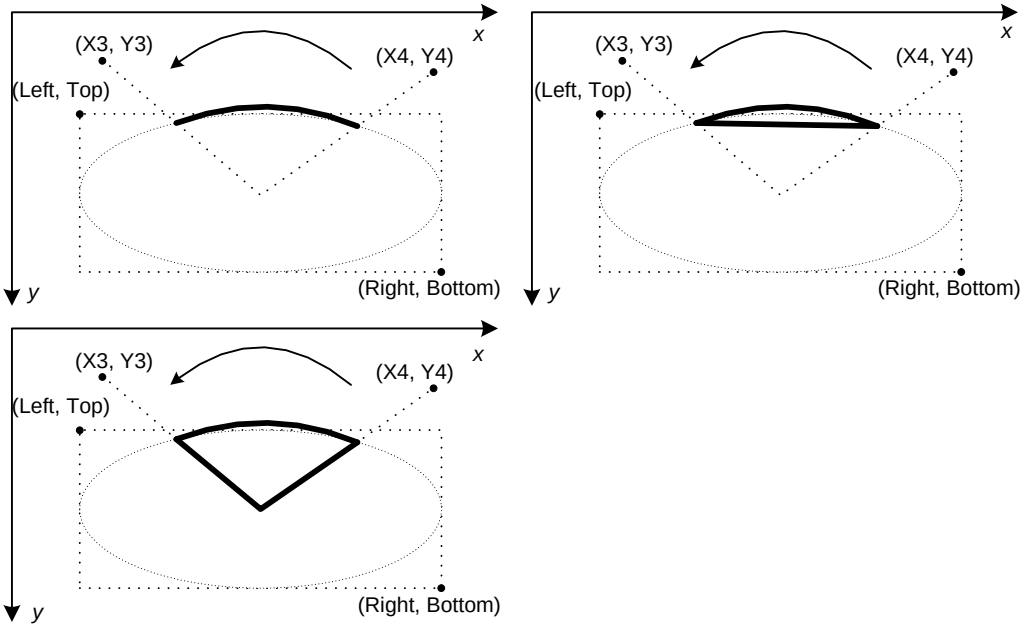


Рис. 31.7. Порядок вывода дуги, хорды и сектора

Примечание

В основу идеи рисования кривой был положен математический аппарат, разработанный на стыке 1950—1960-х годов П. Безье (P. Bezier) и П. де Кастело (P. De Casteljau). Эти математики (независимо друг от друга) создали эффективный и одновременно простой способ описания кривых линий.

Технически для описания кривой Безье достаточно определить всего четыре точки: две точки концов кривой и две точки управления, они отклоняют кривую от центральной оси. От расположения отклоняющих точек зависит, где и каким образом кривая сменит направление. Для построения кривой Безье (кубического сплайна) воспользуйтесь методом

```
procedure PolyBezier(const Points: array of TPoint);
```

Для задания первого сплайна необходимы 4 точки. Каждому последующему сплайну достаточно 3 новых точек, поскольку отправная точка новой кривой есть конечная точка предыдущей. Координатами конечных точек первой кривой служат первый и четвертый элементы массива `Points`. В качестве точек смещения выступают второй и третий элементы массива. Поведение сплайна, состоящего из 4-х точек, демонстрирует рис. 31.8.

Рисунок 31.8 отражает две отличительные особенности сплайна Безье:

- кривая всегда находится внутри фигуры с вершинами в определяющих точках кривой;
- проведенная из конечной в ближайшую отклоняющую точку прямая всегда представляет собой касательную к сплайну в его конечной точке.

В арсенале холста предусмотрен еще один метод, специализирующийся на построении сплайнов:

```
procedure PolyBezierTo(const Points: array of TPoint);
```

Единственное отличие от процедуры `PolyBezier()` заключается в том, что в качестве первой точки кривой Безье используется текущая позиция пера.

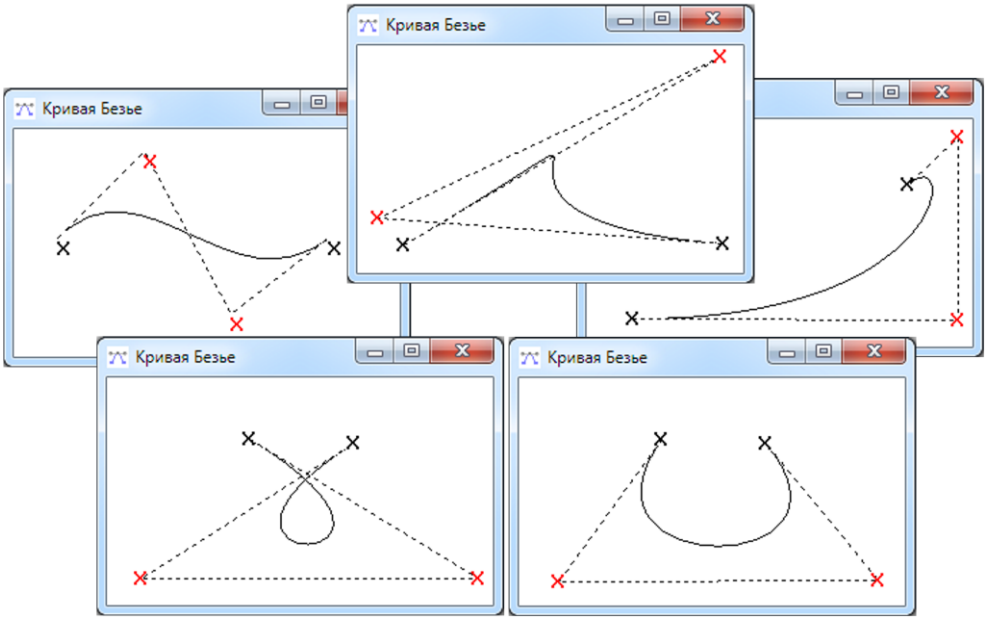


Рис. 31.8. Для определения сплайна Безье достаточно 4-х точек

Копирование части холста

Иногда вместо создания собственного графического шедевра хочется воспользоваться чужими трудами. Обычное копирование изображения, находящегося в пределах прямоугольника `Source` из холста `Canvas`, произведет процедура

```
procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

Результат копирования помещается в прямоугольник `Dest` холста, вызвавшего процедуру. Изменяя размер прямоугольника-получателя, можно получить эффект увеличения или уменьшения изображения. Например, для того чтобы "стащить" изображение рабочего стола и разместить его на своем холсте, нам потребуется всего 10 строк кода (листинг 31.12).

Листинг 31.12. Копирование изображения рабочего стола

```
procedure TForm1.Button1Click(Sender: TObject);
var DesctopCanvas:TCanvas;
begin
  DesctopCanvas:=TCanvas.Create;
  DesctopCanvas.Handle:=CreateDC(PChar('DISPLAY'),nil, nil, nil);
  Application.MainForm.Canvas.CopyRect(
    Application.MainForm.ClientRect, DesctopCanvas, Screen.DesktopRect);
  DesctopCanvas.Free;
end;
```

Щелчок по кнопке `Button1` приводит к тому, что на поверхности формы появляется текущая картина экрана.

Копирование допустимо осуществлять и из растрового изображения, например, скопировать прямоугольник `Source` из битовой карты `Bitmap` в прямоугольник `Dest` поможет процедура

```
procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap;
                    const Source: TRect; Color: TColor);
```

Первая особенность метода заключается в том, что при копировании цвет `Color` замещается текущим цветом кисти. Знание этого нюанса позволяет нарисовать прозрачную картинку. Для этого достаточно заменить фоновый цвет копируемого рисунка фоновым цветом холста. Вторая особенность метода — возможность копировать только часть исходного изображения. Для этого следует определиться с прямоугольными координатами `Source`.

Способ копирования изображения на поверхность холста Delphi определяется свойством

```
property CopyMode: TCopyMode;
```

По умолчанию оно установлено в режим копирования "один в один" — `cmSrcCopy`. В этом режиме пикселы изображения переносятся на холст-получатель без каких-либо изменений. Вместе с этим существует множество альтернативных вариантов вывода картинки на холсте (инвертирование пикселов, сложение пикселов и т. д.). Более подробно о растровых операциях мы поговорим в *главе 33*.

ГЛАВА 32

Растровая и векторная графика

На сегодняшний день существует огромный перечень программных продуктов, представляющих данные в визуальной форме, — это многочисленные графические редакторы, программы промышленного черчения, системы автоматизированного проектирования, геоинформационные пакеты. Все перечисленные приложения для хранения данных (фотографий, рисунков, схем, чертежей) используют разнообразные графические форматы, которые разделяются на два фундаментальных направления: растровую и векторную графику. Суть различия заключается в особенностях представления изображения в памяти. В первом случае это битовый массив, в ячейках которого содержатся данные о цветовых характеристиках каждого пиксела изображения. Во втором случае рисунок представляется в виде коллекции функций, способных рисовать графические примитивы. При последовательном вызове этих функций строится целостная картинка.

Каждый из графических форматов имеет положительные и отрицательные черты. Растровые изображения весьма ресурсоемки, чем больше глубина цвета и размеры рисунка, тем больше задействуется памяти. Векторные рисунки, напротив, характеризуются малыми размерами, ведь вместо раскрашенных пикселов они хранят математическое описание своего содержимого. Если сравнивать векторную и растровую графику по скорости вывода, то векторные изображения оказываются в роли отстающих, ведь прежде чем они превратятся в осязаемый образ, программе необходимо изучить их содержимое и пошагово вызвать все описывающие рисунок функции. Растровые изображения выводятся на экран молниеносно, ведь в большинстве случаев достаточно просто скопировать содержимое памяти на экран. Достоинства растровой графики меркнут, когда на первый план выступают вопросы масштабирования и точного измерения расстояний. Растровый рисунок весьма неудобен для редактирования. Инженер-конструктор нуждается в таком способе описания изображения, воспользовавшись которым он сможет создавать чертеж из набора шаблонов с возможностью легкой реконфигурации изображения (переноса, поворота объектов, изменения их размеров, настройки единиц измерения и т. п.). А это как раз то, чем не в состоянии похвастаться массив раскрашенных пикселов. Поэтому там, где не справляется растровый рисунок, Windows старается воспользоваться услугами векторной графики.

> ***Замечание***
>
> Положительные и отрицательные стороны графических форматов определяет область их применения. Так, основное назначение растровой графики — хранение фотографических и отсканированных изображений, а векторная графика незаменима при построении чертежей и схем.

В соответствии с традициями объектно-ориентированного программирования разработчики Delphi создали ряд классов, существенно упрощающих работу с графикой. Наиболее важные, специализирующиеся на работе с растровой и векторной графикой классы VCL представлены на рис. 32.1.

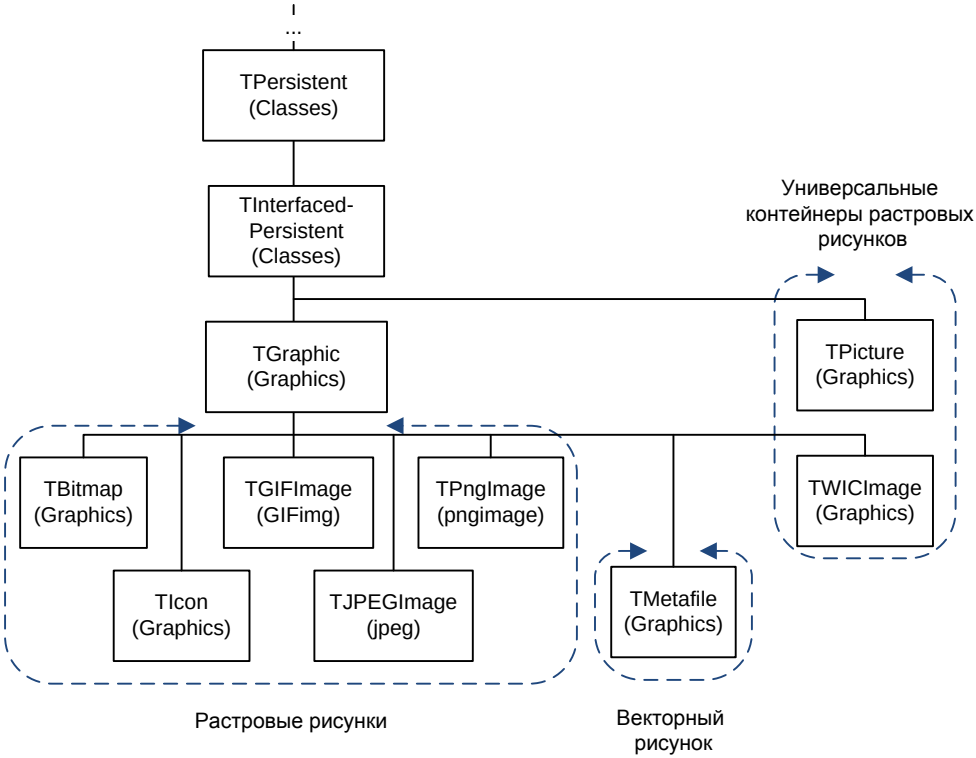


Рис. 32.1. Иерархия классов VCL, способных служить контейнерами для рисунков

Абстрактный базовый класс *TGraphic*

Отправной точкой для всех классов, способных служить контейнером для растровых и векторных изображений, выступает класс `TGraphic`, от него наследуются свойства и методы, обеспечивающие загрузку и сохранение изображений.

Класс `TGraphic` (а как следствие, и все его потомки) поддерживает операции по загрузке/сохранению графических образов из потока

```
procedure LoadFromStream(Stream: TStream); //работа с потоком
procedure SaveToStream(Stream: TStream);
```

файла

```
procedure LoadFromFile(const FileName: string); //работа с файлом
procedure SaveToFile(const FileName: string);
```

и буфера обмена Windows

```
procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle;
                                  APalette: HPALETTE);//буфер обмена
procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle;
                                var APalette: HPALETTE);
```

В качестве параметров при работе с буфером обмена выступают: `AFormat` — зарегистрированный графический формат буфера обмена `Windows`; дескрипторы: `AData` — данных; `APalette` — цветовой палитры.

> ***Замечание***
>
> Для обмена изображениями между большинством потомков класса целесообразно применять метод передачи ссылки на область памяти `Assign()`.

Свойство

```
property Empty: Boolean;
```

принимает значение `true`, если графический объект не содержит данных.

Свойство, показывающее, изменялся ли данный графический объект:

```
property Modified: Boolean;
```

Если для прорисовки изображения необходима палитра цветов, то доступ к дескриптору палитры предоставит свойство:

```
property Palette: HPALETTE;
```

Если изображение не использует палитру, то значение свойства равно 0.

При изменении цветовой палитры изменяется состояние свойства:

```
property PaletteModified: Boolean;
```

За прозрачность графического объекта отвечает свойство:

```
property Transparent: Boolean;
```

Размеры (в пикселах) графического объекта по вертикали и горизонтали:

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

Загрузка графических объектов больших размеров занимает много времени. В классе `TGraphic` предусмотрена возможность информирования пользователя о процессе загрузки данных. Для этого реализовано событие:

```
property OnProgress: TProgressEvent;
type TProgressEvent = procedure(Sender: TObject; Stage: TProgressStage;
                                PercentDone: Byte; RedrawNow: Boolean;
                                const Rct: TRect;
                                const Msg: string) of object;
```

Здесь: `Sender` — элемент-источник сообщения; `Stage` сигнализирует о стадии процесса (`psStarting` — старт, `psRunning` — процесс в работе, `psEnding` — завершение); `PercentDone` — приблизительный процент выполненной работы; `RedrawNow` — состояние прорисовки изображения; `Rct` — прямоугольник, содержащий координаты области изображения, нуждающегося в прорисовке; `Msg` может содержать текстовое сообщение, описывающее текущую операцию.

Замечание

Событие `OnProgress()` доступно только для экземпляра класса `TJPEGImage`.

В листинге 32.1 приведен пример использования события `OnProgress()` совместно с компонентом-шкалой `TProgressBar`, размещенным на форме `frmProgress`. Состоянием параметра `Stage` определяет поведение формы `frmProgress` (показ/закрытие).

Листинг 32.1. Событие загрузки рисунка `OnProgress()`

```
procedure TForm1.Image1Progress(Sender: TObject; Stage: TProgressStage;
          PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
          const Msg: String);
begin
  case Stage of
     psStarting: frmProgress.Show;
     psRunning : frmProgress.ProgressBar1.Position:=PercentDone;
     psEnding  : frmProgress.Close;
  end;
end;
```

Значок *TIcon*

Значок реализует один из самых "древних" способов представления рисунков в Windows. Для обслуживания значков (начиная с самой первой версии Delphi) создан класс `TIcon`. Объект-значок загружается из файла с расширением ico или другого ресурса. Класс `TIcon`, как наследник `TGraphic`, поддерживает операции по работе с потоком и файлом, но не воспринимает буфер обмена Windows.

Набор свойств и методов класса невелик и представлен в табл. 32.1.

***Таблица 32.1.** Свойства и методы значка `TIcon`*

Свойство/метод	Описание
`property Handle: HIcon;`	Дескриптор значка
`property Height: Integer;`	Высота и ширина изображения
`property Width: Integer;`	
`property Transparent: Boolean;`	Прозрачность значка
`function ReleaseHandle: HIcon;`	Уничтожает связанный со значком объект GDI и устанавливает ссылку на него в `nil`

Воспользовавшись Windows API, мы сможем извлечь ассоциированный с файлом значок. Для этого стоит обратить внимание на функцию

```
function ExtractAssociatedIcon(HINSTANCE:hInst; Path : PChar;
                               lpiIcon : Word):HIcon;
```

Она ищет значок в файле, указанном в параметре `Path`. Если в теле файла значок не обнаружен, то поиск продолжается в приложении, обслуживающем данный тип файла. В случае успеха функция возвращает дескриптор значка. В листинге 32.2 представлен пример ис-

пользования функции `ExtractAssociatedIcon()`. Если вы хотите использовать пример на практике, то не забудьте подключить к проекту модуль `ShellAPI`.

Листинг 32.2. Получение иконки из файла

```
uses ShellAPI;

function TForm1.GetIcon(FileName: String): TIcon;
var i:word;
begin
  if FileAge(FileName)>-1 then
  begin
    Result:=TIcon.Create;
    Result.Handle:=ExtractAssociatedIcon(Application.handle,
                                         PChar(FileName), i);
  end
  else Result:=Nil;
end;
```

Формат BMP, класс *TBitmap*

Класс `TBitmap` нацелен на обслуживание растровой графики. На практике битовые образы используются для хранения высококачественных изображений. Еще одно преимущество растровых изображений — в высокой скорости вывода рисунка на экран. За хорошее качество картинки приходится платить существенным объемом используемой памяти, потребности в памяти возрастают в прямой пропорции с геометрическими размерами изображения и глубиной цвета.

Еще со времен первых версий Windows в системе поддерживаются два формата хранения растровых изображений — аппаратно-зависимый битовый образ (Device-Dependent Bitmap, DDB) и аппаратно-независимый битовый образ (Device-Independent Bitmap, DIB). Индикатором формата образа служит свойство

```
property HandleType: TBitmapHandleType;
type TBitmapHandleType = (bmDIB, bmDDB);
```

Растровые картинки в формате DDB в основном применяются для осуществления внутренних операций, например для обеспечения механизма двойной буферизации. Это тот случай, когда для устранения мерцания изображение перед выводом на экран формируется в памяти компьютера. Недостаток аппаратно-зависимой битовой карты заключается в сложности вывода изображения на графических устройствах с иной цветовой организацией. Формат DIB сохраняет свою жизнеспособность на любом графическом устройстве, т. к. он содержит дополнительную служебную информацию о своих цветовых характеристиках. Чаще всего файлы в формате DIB имеют расширение имени bmp, реже — dib и rle.

Черно-белый битовый образ для хранения информации об одном пикселе требует всего один бит. Для выяснения, не является ли наш объект монохромным, обратитесь к свойству

```
property Monochrome: Boolean;
```

Хранение цветного изображения требует дополнительных битов памяти. Число цветов, представляемых в битовом образе, соответствует 2 в степени "число бит на пиксел": 16 цве-

тов — 4 бита на пиксел, 256 — 8, High Color — 15 или 16, True Color — 24 или 32 бита на пиксел.

Глубина цвета битового образа определяется состоянием свойства

```
property PixelFormat: TPixelFormat;
type TPixelFormat = (pfDevice, pf1bit, pf4bit, pf8bit, pf15bit,
                     pf16bit, pf24bit, pf32bit, pfCustom);
```

Формат `pfCustom` предназначен для реализации программистом собственных вариантов хранения данных о цвете.

Растровый рисунок вполне можно сделать прозрачным. Для этого нужно указать цвет фона битовой карты

```
property TransparentColor: TColor;
```

и установить в состояние `tmFixed` свойство

```
property TransparentMode: TTransparentMode;
type TTransparentMode = (tmAuto, tmFixed);
```

Если свойство `Transparent` установлено в автоматический режим (`tmAuto`), то за фоновый цвет битовой карты будет приниматься цвет ее левого верхнего пиксела.

Для использования имеющегося растрового образа в качестве маски для других битовых образов предусмотрено свойство:

```
property MaskHandle: HBitmap;
```

В нем вы обнаружите дескриптор маски.

Для создания маски из обычного образа применяют метод:

```
procedure Mask(TransparentColor: TColor);
```

Метод конвертирует битовую карту в монохромную картинку, заменив `TransparentColor` белым цветом, а все остальные цвета — черным.

Отключение образа-маски производится методом:

```
function ReleaseMaskHandle(): HBITMAP;
```

Класс `TBitmap` инкапсулирует не только битовый образ, но и цветовую палитру. Поэтому объект `TBitmap` обладает двумя дескрипторами:

```
property Handle: HBitmap;   //дескриптор растровой картинки
property Palette: HPalette; //дескриптор палитры
```

Дескриптор битового образа используется для доступа к функциям GDI. Дескриптор палитры предназначен для чтения или изменения цветовой палитры.

Два очень похожих метода отключают от `TBitmap` объекты GDI:

```
function ReleaseHandle: HBitmap;   //отключение от растрового образа
function ReleasePalette: HPalette; //отключение от палитры
```

Функции используются для передачи дескриптора какому-то другому объекту. После этого экземпляр `TBitmap` обнуляет дескриптор. Возможность разделения дескриптора между объектами объясняется наличием механизма кэширования. Но, как только вы вновь обратитесь к дескриптору битовой карты, разделение одной картинки между несколькими объектами заканчивается.

Умение работать с дескриптором битового образа может пригодиться при работе с функциями Win API. Растровые образы активно эксплуатируются в системе в качестве пояснительных рисунков и пиктограмм к элементам управления и пунктам меню. Наличие вспомогательного изображения делает приложение более привлекательным, а интерфейс управления программой интуитивно понятным. Предусмотрено несколько способов подключения к проекту битовых образов. Чаще всего они интегрируются непосредственно в исполняемый файл приложения или во внешний файл ресурса, в роли последнего обычно выступают динамические библиотеки. Несмотря на то, что в ресурсе растры хранятся в формате DIB, доступ к ним предоставляется в формате DDB-растра. Для этого предназначена функция

```
function LoadBitmap(Instance: HINST; BitmapName: PAnsiChar): HBITMAP;
```

Здесь `Instance` — экземпляр библиотеки, загруженной с помощью функции `LoadLibrary()`. Второй параметр, `BitmapName` — название ресурса. Если название неизвестно, то вместо него допускается передать числовой идентификатор, но в этом случае нам потребуются услуги макроса `MAKEINTRESOURCE()`. В случае успешного выполнения функция возвратит дескриптор аппаратно-зависимого битового образа.

> ***Замечание***
>
> Имеется соглашение об именовании модулей, содержащих в себе ресурсы пользовательского интерфейса. Обычно названия таких файлов заканчиваются буквами "ui" (от англ. *user interface*).

Один из вариантов совместной работы класса `TBitmap` и функции `LoadBitmap()` предлагает листинг 32.3. Предполагается, что на форме расположена сетка `StringGrid1:TStringGrid`, содержащая два столбца. На вход функции `FileName()` поступает имя библиотеки, которая загружается в память с помощью функции `LoadLibrary()`. После успешной загрузки мы последовательно опрашиваем библиотеку, проверяя наличие у нее ресурса с идентификатором `ResNum`. Если такой действительно существует, мы передаем дескриптор ресурса объекту `BMP:TBitmap` и связываем этот объект с ячейкой сетки `StringGrid1`.

Листинг 32.3. Загрузка растровых образов из библиотеки ресурсов

```
procedure TForm1.FindInLib(const FileName: string);
var LibInst: longint;
    aRow, ResNum: word;
    BMP: TBitmap;
begin
  LibInst:=LoadLibrary(pChar(FileName));
  if LibInst>0 then
  begin
    aRow:=1;
    for ResNum:=0 to High(ResNum)-1 do
    begin
      BMP:=TBitmap.Create;
      BMP.Handle:=LoadBitmap(LibInst,MAKEINTRESOURCE(ResNum));
      if BMP.Empty=false then
      begin
        if StringGrid1.RowCount<=aRow then
           StringGrid1.RowCount:=StringGrid1.RowCount+1;
```

```
        StringGrid1.Cells[0,aRow]:=IntToStr(ResNum);
        StringGrid1.Objects[1,aRow]:=TObject(BMP);
        INC(aRow);
      end else BMP.Free;
    end;
    FreeLibrary(LibInst);
  end;
end;
```

Для того чтобы пользователь смог увидеть ресурс в `StringGrid1`, следует воспользоваться событием перерисовки ячеек сетки `OnDrawCell` (листинг 32.4).

Листинг 32.4. Вывод битового образа в ячейке сетки

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
              ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
  if StringGrid1.Objects[ACol, ARow]<>nil then
    with StringGrid1.Canvas do
      Draw(Rect.Left,Rect.Top,TBitMap(StringGrid1.Objects[ACol, ARow]));
end;
```

Так как числовые идентификаторы растровых ресурсов нам не были заранее известны, пришлось проанализировать множество номеров `ResNum`, пока мы не добрались до намеченной цели. Например, в недрах библиотеки cryptui.dll из поставки Windows 7 пряталось несколько битовых образов (идентификаторы 305—316). В этом вы можете убедиться, взглянув на рис. 32.2 с экранным снимком написанного нами приложения.

Для загрузки данных из ресурса можно воспользоваться и встроенными методами:

```
procedure LoadFromResourceID(Instance: THandle; ResID: Integer);
procedure LoadFromResourceName(Instance: THandle; const ResName: string);
```

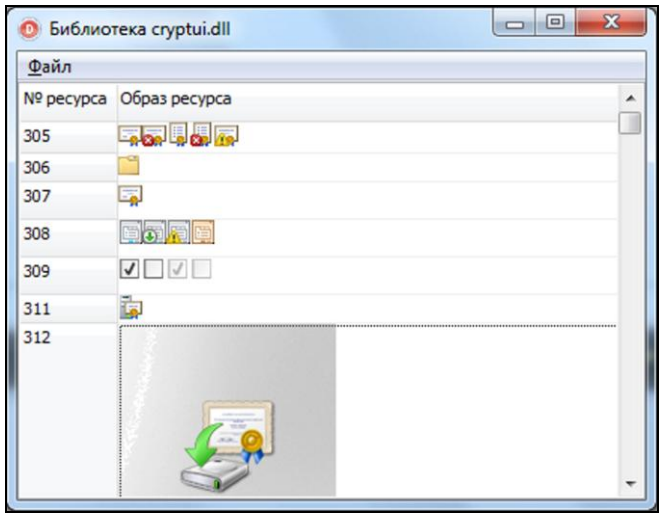


Рис. 32.2. Просмотр ресурсов библиотеки cryptui.dll

В параметр `Instance` передается ссылка на экземпляр ресурса. Выбор битового образа из ресурса осуществляется или при помощи параметра `ResID` (идентификатор ресурса) или `ResName` (имя элемента ресурса). Однако для рассмотренного в листинге 33.3 примера методы не вполне пригодны. Дело в том, что при отсутствии ресурса с заданным идентификатором `LoadFromResourceID()` и `LoadFromResourceName()` генерируют исключительную ситуацию.

Процедура

```
procedure Dormant;
```

создает в памяти растровый рисунок в формате DIB, уничтожая разделяемые дескрипторы предыдущей битовой карты.

Процедура

```
procedure FreeImage;
```

освобождает занятую битовым образом память.

На страницах этой книги мы уже обсуждали сложность решения задач, связанных с качественным масштабированием растровых изображений. Одно из готовых решений предлагает Delphi — это процедура

```
procedure ScaleImage(const SourceBitmap, ResizedBitmap: TBitmap;
                     const ScaleAmount: Double);
```

В процедуру направляются три параметра: исходное изображение `SourceBitmap`, коэффициент масштабирования `ScaleAmount`, результат операции помещается в `ResizedBitmap`.

Формат JPEG, класс *TJPEGImage*

Класс `TJPEGImage` инкапсулирует возможности одного из наиболее распространенных форматов сжатой растровой картинки — JPEG (Joint Photographic Expert Group). Этот формат был специально разработан для хранения 24-битных изображений. Исключительное достоинство JPEG заключается в том, что ему практически нет равных в искусстве сжатия фотографических изображений.

> ***Замечание***
>
> Для поддержки проектом Delphi файлов в формате JPEG подключите к нему одноименный модуль `jpeg`.

К особенностям класса `TJPEGImage` следует отнести:

- отсутствие у него холста (хотя объект `TJPEGImage` может нарисовать себя на холсте других объектов);
- класс не обеспечивает доступа к отдельному пикселу своего изображения;
- благодаря объекту `TJPEGData` допускает совместное использование своего дескриптора.

Объект `TJPEGImage` обязан не только сжимать растровые изображения, но и отображать распакованные данные в аппаратно-независимом формате DIB. Самый простой вариант взаимодействия между сжатой и обычной растровой картинкой осуществляется при помощи метода `Assign()`. В таком случае производится автоматическое конвертирование форматов: из DIB в JPEG или наоборот.

Для преобразования рисунка из формата JPEG в классический несжатый растр DIB предназначен метод:

```
procedure DIBNeeded;
```

Обычно процедура используется перед вызовом метода `Draw()` холста. Такой подход ускоряет прорисовку картинки.

Обратное преобразование (из DIB в JPEG) осуществляется методом:

```
procedure JPEGNeeded;
```

В результате несжатый растр конвертируется в JPEG-данные.

Для сжатия обычного растрового изображения в формат JPEG прежде всего надо установить степень упаковки. Для этого в классе `TJPEGImage` опубликовано свойство

```
property CompressionQuality: TJPEGQualityRange;//по умолчанию 90
type TJPEGQualityRange = 1..100;
```

Чем больше значение `CompressionQuality`, тем лучше сохранится качество изображения, но больше размер результирующего файла. С уменьшением значения `CompressionQuality` степень сжатия возрастает, но и ухудшается картинка.

Метод, осуществляющий сжатие растровой картинки, называется

```
procedure Compress;
```

Эта процедура автоматически вызывается при сохранении объекта JPEG в файл и поток.

> ***Внимание!***
>
> Применяемые в формате JPEG методы сжатия относятся к алгоритмам сжатия с потерями. Поэтому формат JPEG не стоит использовать в качестве промежуточного способа хранения изображения, в особенности когда приходится многократно редактировать и сохранять рисунок.

Два взаимосвязанных свойства определяют способ вывода изображения на экран — по мере распаковки (такой метод называют *прогрессивной разверткой*) или только по окончанию распаковки. Вывод по мере распаковки реализуется за счет того, что изображение JPEG разбивается на множество сканов. Начальные сканы являются грубым представлением картинки, но каждый последующий скан улучшает качество изображения.

```
property ProgressiveEncoding: Boolean;
property ProgressiveDisplay: Boolean;
```

Прогрессивная развертка картинки (`ProgressiveDisplay=true`) возможна только при условии, что файл сохранен в режиме прогрессивного сжатия (`ProgressiveEncoding=true`).

В том случае если вы используете прогрессивную развертку, допустимо сглаживание изображения:

```
property Smoothing: Boolean;
```

Качество изображения зависит не только от степени сжатия растровой картинки. Ряд свойств класса определяет работу `TJPEGImage` при декомпрессии и показе изображения. Критерий "скорость распаковки/качество изображения" задается в свойстве

```
property Performance: TJPEGPerformance;
type TJPEGPerformance = (jpBestQuality, jpBestSpeed);
```

Выбрав значение `jpBestQuality`, мы затратим несколько больше времени на распаковку файла, но получим более качественную картинку. Выбрав значение `jpBestSpeed`, получим обратный результат. На качество сохраняемого изображения `Performance` не влияет.

Критерий "скорость вывода изображения/глубина цветов" определяется свойством

```
property PixelFormat: TJPEGPixelFormat;
type TJPEGPixelFormat = (jf24Bit, jf8Bit);
```

При выборе значения `jf24Bit` вы получите полноценное 24-битное изображение, но на его вывод затратите значительно больше времени, чем на 8-битное.

Класс `TJPEGImage` поддерживает быстрый способ масштабирования:

```
property Scale: TJPEGScale;
type TJPEGScale = (jsFullSize, jsHalf, jsQuarter, jsEighth);
```

В соответствии со значениями `TJPEGScale` вы получите изображение в соотношении 1:1, 1:2, 1:4 и 1:8 к размерам исходной картинки.

Формат GIF, класс *TGifImage*

Формат графического обмена (Graphic Interchange Format, GIF) был анонсирован в 1987 году компанией CompuServe. Как следует из названия, GIF в первую очередь предназначался для сетевого обмена, поэтому разработчики формата приложили максимум усилий для сокращения размера файла. Именно поэтому максимальная поддерживаемая в формате глубина цвета составляет всего 8 бит. Кроме того, изображение в GIF может быть сжато за счет метода LZW. Кроме легковесности у GIF-формата имеется еще одна положительная черта — поддержка анимации. Такой результат достигается за счет хранения в файле многостраничных изображений. Последовательная смена изображений создает эффект мультипликации — рисунок оживает.

> ***Замечание***
>
> Для поддержки проектом Delphi файлов в формате GIF подключите к нему одноименный модуль `GIFimg`.

Для того чтобы приложения смогли качественно отобразить рисунок, у последнего имеется ряд информационных свойств, рассказывающих об особенностях изображения. О версии изображения сообщит свойство

```
property Version: TGIFVersion; //только для чтения
type TGIFVersion =(gvUnknown, gv87a, gv89a);
```

Существуют два варианта ответа: GIF87a и GIF89a. Формат GIF87a способен хранить только рисунки, формат GIF89a содержит дополнительный блок расширения, в котором могут храниться дополнительные сведения (отображаемый текст, текст комментария и даже программные команды управления графикой).

Сведения о глубине цвета хранятся в свойстве

```
property BitsPerPixel: integer; //только для чтения
```

Максимально допустимая глубина цвета (анализируются все картинки из состава GIF):

```
property ColorResolution: integer; //только для чтения
```

Признак наличия сведений о прозрачности:

```
property IsTransparent: boolean; //только для чтения
```

> ***Замечание***
>
> Загрузка рисунка осуществляется классическими для всех потомков `TGraphic` методами `LoadFromFile()` и `LoadFromStream()`.

Для включения анимации в многостраничных изображениях следует передать `true` в свойство

```
property Animate: boolean;
```

Особенности анимационного цикла задает свойство

```
property AnimateLoop: TGIFAnimationLoop;
type  TGIFAnimationLoop = (
    glDisabled,     //цикл отключен
    glEnabled,      //цикл по правилам, определенным в рисунке GIF
    glContinously);//бесконечный цикл независимо от установок рисунка
```

При желании допустимо изменить скорость смены картинок. Для этого предназначено свойство

```
property AnimationSpeed: integer;
```

Для вывода анимированного изображения на поверхности формы проще всего воспользоваться услугами компонента-рисунка `TImage`. В предложенном в листинге 32.5 примере мы создаем экземпляр класса `TGifImage`, загружаем в него файл с изображением и передаем рисунок в распоряжение компонента `Image1:TImage`.

Листинг 32.5. Загрузка и отображение рисунка GIF в компоненте `TImage`

```
uses GIFimg;
//...
var GIFImage:TGifImage;
begin
  if OpenPictureDialog1.Execute then
  begin
    GIFImage:= TGifImage.Create;
    GIFImage.LoadFromFile(OpenPictureDialog1.FileName);
    GIFImage.BackgroundColor:=clWhite; //белый фон
    GIFImage.AnimateLoop:=glEnabled;   //замкнутый цикл анимации
    GIFImage.Animate:=True;            //активируем анимацию
    Image1.Picture.Assign(GIFImage);
    GIFImage.Free;
  end;
end;
```

Класс `TGifImage` позволяет программисту легко управлять фоном рисунка. Для этого надо передать индекс подходящего цвета в палитре или явное значение фонового цвета в свойства:

```
property BackgroundColorIndex: BYTE;
property BackgroundColor: TColor;
```

Управление фреймами рисунка GIF

Наиболее яркая отличительная черта `TGifImage` — инкапсуляция хранилища многостраничных изображений. Доступ хранилищу предоставляет свойство

```
property Images: TGIFImageList; //только для чтения
```

Хранилище реализовано в форме списка `TGIFImageList`. Список позволяет управлять фреймами (экземплярами класса `TGifFrame`), входящими в состав рисунка. Основные свойства и методы хранилища `TGIFImageList` представлены в табл. 33.2, а фрейм рассмотрен в табл. 32.3.

***Таблица 32.2.** Основные характеристики списка изображений `TGIFImageList`*

Свойство	Описание
`property Count: Integer;`	Число элементов списка
`function First: TGIFItem;`	Доступ к первому элементу списка
`function Last: TGIFItem;`	Доступ к последнему элементу списка
`property Frames[Index: Integer]: TGIFFrame;`	Доступ к заданному фрейму по индексу
`property Items[Index: Integer]: TGIFItem;`	Доступ к любому элементу списка по индексу
`function IndexOf(Item: TGIFItem): Integer;`	Возвращает индекс фрейма `Item`
`function Add(Item: TGIFItem): Integer;`	Добавляет в конец списка новый элемент `Item`
`procedure Insert(Index: Integer; Item: TGIFItem);`	Вставляет элемент `Item` в указанную позицию списка
`procedure Clear;`	Полная очистка списка
`procedure Delete(Index: Integer);`	Удаляет фрейм с индексом `Index` из списка
`function Remove(Item: TGIFItem): Integer;`	Удаляет фрейм `Item`
`procedure Exchange(Index1, Index2: Integer);`	Заменяет фреймы местами
`procedure Move(CurIndex, NewIndex: Integer);`	Перемещает фрейм из позиции `CurIndex` в `NewIndex`

***Таблица 32.3.** Основные характеристики фрейма `TGifFrame`*

Свойство	Описание
`procedure Clear;`	Очистка содержимого фрейма
`property Empty: boolean;`	Проверка на наличие данных
`property HasBitmap: boolean;`	Проверка на наличие битового образа
`property Bitmap: TBitmap;`	Представляет фрейм в виде битового образа. Свойство доступно как для чтения, так и для записи
`property Left: WORD;`	Местоположение фрейма относительно опорного рисунка и его размеры
`property Top: WORD;`	
`property Width: WORD;`	
`property Height: WORD;`	

Таблица 32.3 *(окончание)*

Свойство	Описание
`procedure Draw(ACanvas: TCanvas; const Rect: TRect; DoTransparent, DoTile: boolean);`	Методы позволяют осуществлять индивидуальную прорисовку фрейма
`procedure StretchDraw(ACanvas: TCanvas; const Rect: TRect; DoTransparent, DoTile: boolean);`	
`procedure SaveToStream(Stream: TStream);`	Методы, позволяющие сохранять/загружать фрейм
`procedure LoadFromStream(Stream: TStream);`	

На рис. 32.3 представлен экранный снимок приложения, демонстрирующего порядок работы с фреймами многостраничного GIF.

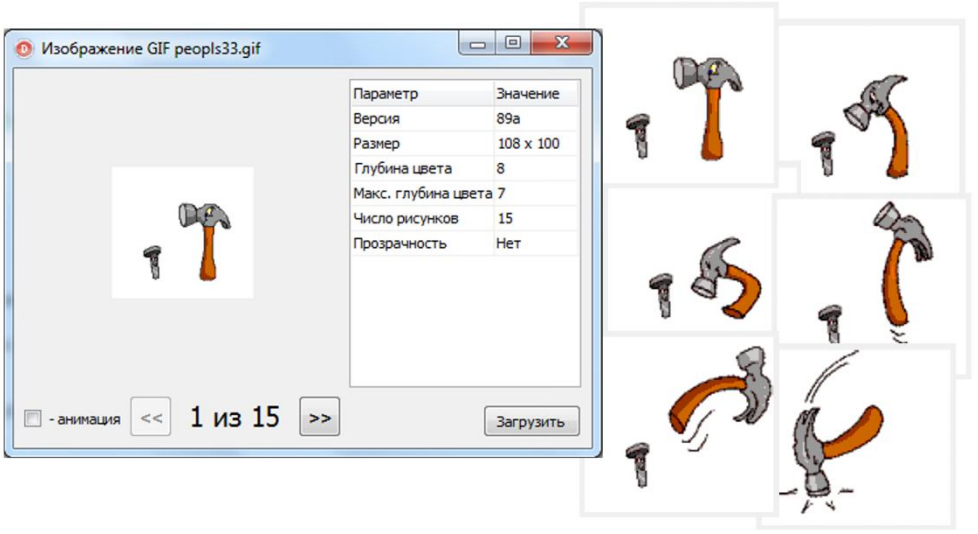


Рис. 32.3. Приложение просмотра изображений GIF с поддержкой анимации

Фрейм к набору картинок GIF можно добавить и напрямую, не обращаясь к списку `TGIFImageList`, а воспользовавшись услугами метода

```
function Add(Source: TPersistent): TGifFrame;
```

В качестве источника фрейма может быть любой совместимый по своим параметрам рисунок, ссылка на него передается в единственный параметр метода.

Оптимизация рисунка GIF

Во времена разработки формата GIF предполагалось, что 256 цветов вполне достаточно для качественного представления графики. Уверенность в этой идее подкреплял еще тот факт, что отказ от полноцветных изображений существенно снижает итоговый размер файла GIF, а это весьма важно для быстрой передачи картинок по сети. Поэтому вместо реальных значений цветов в картинке GIF содержатся ссылки на палитру цветов. Подробные сведения об используемой в картинках GIF палитре цветов предоставит свойство

```
property GlobalColorMap: TGIFColorMap; //только для чтения
```

Для оптимизации состава цветов в палитре GIF проще всего обратиться к свойству

```
property ColorReduction: TColorReduction;
type TColorReduction = (rmNone, rmWindows20, rmWindows256, rmWindowsGray,
                        rmMonochrome, rmGrayScale, rmNetscape, rmQuantize,
                        rmQuantizeWindows, rmPalette);
```

Выбирая то или иное состояние свойства, вы определяете состав палитры. Например, режим `rmWindows20` устанавливает палитру из 20 цветов, `rmWindowsGray` — палитру оттенков серого. Наиболее универсально состояние — `rmQuantize`, в этом случае глубина цвета настраивается с помощью свойства

```
property ReductionBits: integer;
```

Как всегда, у медали есть и обратная сторона. Внешний вид рисунков на основе палитры проигрывает их полноцветным собратьям. Из-за этого для вывода графики GIF часто применяют дополнительные цифровые алгоритмы, улучшающие результирующее изображение. Для улучшения качества вывода рисунка класс `TGifImage` может воспользоваться цветовыми фильтрами, перечень которых хранится в свойстве

```
property DitherMode: TDitherMode;
type TDitherMode = (dmNearest,dmFloydSteinberg,dmStucki, dmSierra,
                    dmJaJuNI,dmSteveArche, dmBurkes);
```

Для активизации фильтра следует обратиться к свойству

```
property Dithering: TGIFDithering;
type TGIFDithering = (gdDisabled, //отключено
                      gdEnabled,  //включено
                      gdAuto);{включено, если дисплей или браузер работает
                          не в полноцветном (256 цветов и ниже) режиме}
```

Простую оптимизацию палитры цветов осуществляет процедура

```
procedure OptimizeColorMap;
```

Метод удалит повторяющиеся цвета и освободит незадействованные ячейки палитры.

Если вам лень оптимизировать рисунок, последовательно обращаясь к тому или иному свойству, то стоит воспользоваться одним-единственным методом оптимизации:

```
procedure Optimize(Options: TGIFOptimizeOptions;
                   ColorReduction: TColorReduction = rmNone;
                   DitherMode: TDitherMode = dmNearest;
                   ReductionBits: integer = 8);
```

Параметры метода вряд ли нуждаются в комментариях, ведь их названия повторяют имена рассмотренных ранее свойств. Единственное исключение составляет параметр, отвечающий за опции оптимизации `Options`:

```
type  TGIFOptimizeOption = (
  ooCrop,     //обрезает анимированные фреймы
  ooMerge,    //объединяет пикселы одинаковых цветов
  ooCleanup,  //удаляет комментарии и расширения приложения
  ooColorMap);//упорядочивает палитру и удаляет неиспользуемые элементы
```

Если свойство

```
property ShouldDither: boolean; //только для чтения
```

возвращает значение `true`, то это является признаком использования алгоритмов оптимизации.

Обработка событий

Класс `TGifImage` отличается достаточно серьезным набором обработчиков событий (табл. 32.4).

***Таблица 32.4.** Обработка событий в классе `TGifImage`*

Событие	Описание
`property OnWarning: TGIFWarning;`	Сигнализирует о возникновении исключительной операции, связанной с выводом или обработкой изображения
`property OnStartPaint: TNotifyEvent;`	Генерируется перед началом вывода изображения
`property OnPaint: TNotifyEvent;`	Вызывается в момент прорисовки изображения
`property OnEndPaint : TNotifyEvent;`	Завершает вывод изображения
`property OnAfterPaint: TNotifyEvent;`	Вывод изображения завершен
`property OnLoop: TNotifyEvent;`	Цикл вывода анимированного изображения

Формат PNG, класс *TPngImage*

Класс `TPngImage` предназначен для обслуживания новейшего из всех перечисленных в главе форматов растровых изображений PNG (Portable Network Graphics). Формат появился на свет осенью 1996 года. Если следовать дословному переводу PNG — это портативная сетевая графика. Чем хорош PNG? Во-первых, он использует алгоритм сжатия без потерь, что в выгодную сторону отличает его от JPEG. Во-вторых, формат поддерживает 48-битную глубину цвета. Благодаря этому PNG способен без постороннего вмешательства управлять прозрачностью. В-третьих, PNG характеризуется наилучшим цветовым соответствием (он поддерживает пять различных цветовых моделей).

> ***Замечание***
>
> Для поддержки проектом Delphi файлов в формате PNG подключите к нему модуль `pngimage`.

Мы не станем глубоко вдаваться в подробности графического формата PNG, но обязательно упомянем важную особенность организации данных в этих изображениях. Рисунки в формате PNG представляют собой последовательность *блоков данных* (или как иногда их называют — *порций*, от англ. *chunk*), содержащих сведения об изображении. Создатели формата разрешают определять три разновидности блоков:

1. Обязательные блоки, без обработки которых мы не сможем достать из объекта изображение. Стандарт PNG определяет 4 критические порции: IHDR, PLTE, IDAT и IEND. Критические порции составляют минимально необходимый набор обязательных блоков, присутствующих в каждом изображении.
2. Открытые (public) блоки, содержащие специфичные данные от сторонних разработчиков. Независимые программисты вносят свои предложения о включении своих наработок в структуру объекта PNG в группу PNG Development Group, и если наработки кажутся полезными создателям PNG, то им присваивается статус открытых.
3. Частные (private) блоки, определяемые любыми программистами и используемые только в их программах.

Если возникает задача создать новый рисунок в формате PNG, то следует воспользоваться услугами конструктора

```
constructor CreateBlank(ColorType, Bitdepth: Cardinal; CX, CY: Integer);
```

позволяющего создавать пустой рисунок. Цветовую схему определяет первый параметр `ColorType` конструктора, он может принимать следующие значения:

- `COLOR_GRAYSCALE` — полутоновая шкала градаций серого;
- `COLOR_RGB` — полноцветное изображение в формате RGB;
- `COLOR_PALETTE` — изображение на основе палитры цветов;
- `COLOR_GRAYSCALEALPHA` — градации серого с поддержкой прозрачности;
- `COLOR_RGBALPHA` — полноцветное изображение RGB с поддержкой прозрачности.

Параметр `Bitdepth` конкретизирует точность дискретизации (битовую глубину), допустимые значения — 1, 2, 4, 8 и 16 бит. Геометрические размеры создаваемого рисунка определяются аргументами `CX` и `CY`.

Сразу после создания рисунок пуст, об этом можно судить по свойству

```
property Empty: Boolean;
```

Изображение обладает собственным холстом:

```
property Canvas: TCanvas;
```

Благодаря этой черте очень легко вносить редакторские правки в любой загруженный в созданный на основе `TPngImage` рисунок. Для этого достаточно просто воспользоваться услугами имеющихся в распоряжении класса `TCanvas` свойств и методов.

Еще один способ доступа к содержимому рисунка PNG обеспечивает свойство

```
property Pixels[const X, Y: Integer]: TColor;
```

Благодаря этому свойству программист сможет общаться с рисунком, как с двумерным массивом пикселов.

Все вышесказанное подытоживает код создания рисунка в формате PNG (листинг 32.6). В рамках листинга мы создаем рисунок в цветовой схеме RGB с 8-битовой точностью дискретизации размером 100 × 100 пикселов.

Листинг 32.6. Создание рисунка в формате PNG

```
var PNG: TPngImage;
    x,y,z:real;
begin
  PNG:=TPngImage.CreateBlank(COLOR_RGB,8,100,100);
  with PNG.Canvas do
  begin
    Brush.Color:=clWhite;
    FillRect(Rect(0,0,100,100));
    Pen.Color:=clGray;
    Rectangle(Rect(0,0,100,100));
    Pen.Color:=clRed;
    Brush.Color:=clYellow;
```

```
    Ellipse(Rect(1,1,99,99));
    Font.Color:=clNavy;
    TextOut(20,45,'Формат PNG');
  end;
  PNG.TransparentColor:=clWhite;
  PNG.Draw(Form1.Canvas,Rect(10,10,110,110));
  PNG.SaveToFile('c:\demo.png');
  PNG.Free;
end;
```

Листинг 32.6 раскрывает еще одну интересную особенность класса `TPngImage` — наличие собственных методов вывода изображения на поверхности холста:

```
procedure Draw(ACanvas: TCanvas; const Rect: TRect);
procedure DrawUsingPixelInformation(Canvas: TCanvas; Point: TPoint);
```

Создатели класса справедливо полагают, что такой сложный графический объект, как PNG, гораздо лучше холста знает, как следует осуществлять собственную прорисовку. Для этого ему достаточно получить ссылку на экземпляр холста и определить прямоугольную область `Rect`, в которой следует отобразить рисунок или координаты левой верхней точки начала вывода `Point`. Разница между методами заключается не только в особенностях определения места вывода рисунка PNG, но и в порядке интерпретации данных, описывающих изображение. Так, метод `Draw()` учитывает сведения только из обязательных блоков изображения PNG, а метод `DrawUsingPixelInformation()` позволяет выводить и сведения из частных секций изображения (блоков, разработанных программистом специально для конкретной программы).

> ***Замечание***
>
> Если вам потребуется получить абсолютный доступ к файлу PNG, то стоит вспомнить о наборе свойств и методов класса `TPngImage`, поддерживающих низкоуровневый доступ к блокам изображения (`Header`, `Chunks`, `PixelInformation`, `Scanline`, `ExtraScanline` и т. д.).

На размер сохраняемого изображения оказывает влияние уровень компрессии, задаваемый в свойстве

```
property CompressionLevel: TCompressionLevel;// TCompressionLevel = 0..9;
```

Нулевое значение соответствует несжатому изображению, а 9 назначает максимальную степень сжатия.

На особенности сжатия оказывает влияние свойство

```
property Filters: TFilters;
type TFilter = (pfNone, pfSub, pfUp, pfAverage, pfPaeth);
```

Свойство определяет метод фильтрации. Он должен быть назначен перед началом сжатия графического образа.

Формат PNG поддерживает режим чередования, называемый Adam 7. Идея метода принадлежит Адаму Кастелло. Для ускорения загрузки изображение делится на блоки 8×8 пикселов и обновляется за семь проходов. Для включения/отключения режима отвечает свойство

```
property InterlaceMethod: TInterlaceMethod;
type TInterlaceMethod = (imNone, imAdam7);
```

Осуществляя управление прозрачностью, в простейшем случае достаточно указать цвет, не подлежащий отображению на устройствах вывода. Для этого предназначено свойство

```
property TransparentColor: TColor;
```

Задача по информированию программиста о выбранном режиме прозрачности возложена на свойство

```
property TransparencyMode: TPNGTransparencyMode;//только для чтения
type TPNGTransparencyMode = (ptmNone,    //прозрачность не используется
                             ptmBit,     //объявлен прозрачный цвет
                             ptmPartial);//используется альфа-канал
```

Если уже существующее PNG-изображение нуждается в подключении альфа-канала, то обращаемся к методу

```
procedure CreateAlpha;
```

В этом случае структура изображения преобразуется, к каждому пикселу рисунка добавляются индивидуальные сведения о его прозрачности.

Для полного отказа от прозрачности достаточно вызвать процедуру

```
procedure RemoveTransparency;
```

Полезная черта класса — встроенный метод, позволяющий масштабировать изображение:

```
procedure Resize(const CX, CY: Integer);
```

Новые геометрические размеры по горизонтали и вертикали направляются в параметры `CX` и `CY`.

Кодер PNG допускает хранение в графическом объекте текстовой информации в формате ASCII, точнее говоря, комбинаций "ключевое слово — значение". Вставку текста осуществляют методы:

```
procedure AddtEXt(const Keyword, Text: AnsiString);
procedure AddzTXt(const Keyword, Text: AnsiString);
```

Метод `AddtEXt()` обычно используется для вставки заранее предопределенных ключевых слов, хранящих сведения об авторе (Author), комментариях (Comment), правах (Copyright), времени создания (Creation Time), описании (Description), заголовке (Title) и т. д. Метод `AddzTXt()` используется для вставки дополнительных текстовых сведений. Существует еще техническое отличие в хранении текстовых пар, добавленных в изображение PNG с помощью методов `AddtEXt()` и `AddzTXt()`. Текст, добавленный методом `AddzTXt()`, сжимается алгоритмом архивации, а текст, внесенный в картинку методом `AddtEXt()`, хранится в первозданном виде.

Векторная графика, метафайл *TMetaFile*

В Windows векторная графика зиждется на понятии метафайла, способного представлять рисунок в формате, независящем от устройства. В отличие от растрового рисунка метафайл хранит не массив окрашенных пикселов, а множество обращений к функциям GDI. В момент вывода изображения на поверхности устройства метафайл последовательно вызывает все описанные в нем функции, поэтому часто используют выражение "воспроизведение метафайла".

В Windows широко применяются два формата метафайлов: *метафайл* (Windows Metafile, WMF) и более современная версия — *метафайл с расширенными возможностями*

(Enhanced Metafile, EMF). Метафайл WMF появился на свет в 1985 году вместе с Windows 1.0 и к сегодняшнему дню устарел. Ключевое различие между форматами WMF и EMF в том, что новая версия метафайла не зависит от устройства и способна использовать современные команды GDI. По умолчанию класс `TMetaFile` нацелен на работу с расширенным метафайлом. Для обеспечения совместимости с устаревшим форматом WMF опубликовано свойство

```
property Enhanced: Boolean; //состояние true — EMF, false — WMF
```

Низкоуровневый доступ к функциям Windows API реализуется через дескрипторы метафайла и его палитры:

```
property Handle: HMetafile;
property Palette: HPalette;
```

Целый набор свойств связан с геометрическими размерами метафайла. Экранные размеры в пикселах:

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

Размеры в сотых долях миллиметра (0,01 мм):

```
property MMHeight: Integer;
property MMWidth: Integer;
```

Число точек на дюйм в координатной системе метафайла:

```
property Inch: Word;
```

Информацию о создателе и краткое описание метафайла можно получить из свойств:

```
property CreatedBy: string;  //только для чтения
property Description: string; //только для чтения
```

Объект `TMetafile` предназначен для вывода изображения на графическом устройстве. А для того чтобы нарисовать метафайл, необходимо использовать возможности класса `TMetafileCanvas`.

Холст метафайла *TMetafileCanvas*

Холст метафайла `TMetafileCanvas` — это потомок уже известного нам класса `TCanvas`, предназначенный для создания векторных рисунков в программах, написанных на Delphi. Перечень свойств и методов практически на 100% соответствует возможностям родительского класса `TCanvas`, но за некоторым исключением.

У холста метафайла `TMetafileCanvas` переопределен конструктор

```
constructor Create(AMetafile: TMetafile; ReferenceDevice: HDC);
```

Конструктор создает экземпляр холста метафайла и выделяет ему дескриптор. Параметр `AMetafile` соответствует объекту `TMetafile`, `ReferenceDevice` — дескриптор графического устройства.

Существует и более совершенная версия конструктора, она позволяет не только создавать экземпляр метафайла, но и добавлять к нему текстовую информацию:

```
constructor CreateWithComment(AMetafile: TMetafile; ReferenceDevice: HDC;
                              const CreatedBy, Description: String);
```

Для этих целей в состав параметров конструктора вошли еще два параметра: создатель `CreatedBy` и описание рисунка `Description`.

Представленный в листинге 32.7 код демонстрирует порядок создания векторного рисунка из трех объектов — красного прямоугольника, зеленого эллипса, синего треугольника — и текстовой надписи.

Листинг 32.7. Рисуем метафайл

```
var MetaFile :TMetaFile;
begin
  MetaFile:=TMetaFile.Create;
  MetaFile.Height:=200;
  MetaFile.Width:=200;
  with TMetaFileCanvas.Create(MetaFile,0) do
  begin
    Brush.Color := clRed;
    Rectangle(5,5,85,85);
    Brush.Color := clGreen;
    Ellipse(40,40,120,120);
    Brush.Color := clBlue;
    Polygon([Point(110,80),Point(60,140),Point(160,140)]);
    Brush.Color:=clWhite;
    TextOut(15,150,'Метафайл EMF');
    Destroy;
  end;
  Form1.Canvas.Draw(0,0,MetaFile);
  MetaFile.SaveToFile('C:\Metafile.emf');
  MetaFile.free;
end;
```

Универсальный контейнер *TPicture*

Класс `TPicture` служит классическим примером инкапсуляции и полиморфизма. Экземпляр класса представляет собой универсальное хранилище изображения, способное содержать любой из рассмотренных в этой главе графических объектов. Более того, `TPicture` превосходно справляется с решением вопросов по вызову методов, характерных для конкретного потомка `TGraphic`, тем самым упрощая работу программиста.

Обращение к графическому ресурсу реализуется при помощи свойства:

```
property Graphic: TGraphic;
```

В дополнение к этому (в зависимости от типа объекта) доступ осуществляется через свойства:

```
property Icon: TIcon;
property Bitmap: TBitmap;
property Metafile: TMetafile;
```

Методы, связанные с загрузкой/сохранением графического объекта, перед выполнением анализируют тип графического объекта, а затем вызывают соответствующие этим объектам методы `LoadFromFile()`, `SaveToFile()`, `LoadFromClipboardFormat()` и `SaveToClipboardFormat()`.

Замечание

Для того чтобы контейнер `TPicture` приобрел возможность обработки дополнительных графических форматов (JPEG, GIF, TIF и PNG), необходимо подключение модулей: `jpeg`, `GIFimg` и `pngimage`.

Универсальный контейнер *TWICImage*

Сравнительно недавно в состав VCL вошел еще один класс — `TWICImage`, специализирующийся на работе с изображениями. В отличие от рассмотренного ранее контейнера `TPicture` наш новый знакомый демонстрирует альтернативный подход к обслуживанию растровых изображений. Отличие класса в том, что на этот раз `TWICImage` инкапсулирует не фирменные классы Embarcadero, а идеи корпорации Microsoft, точнее, набор структур и низкоуровневых функций Windows Imaging Component (WIC).

Внимание!

Для работы функций Windows Imaging Component приложение должно выполняться под управлением операционной системы Windows XP SP3/Vista/7 с установленным пакетом .NET 3.0, а также необходимо наличие DirectX Runtime.

Ключевое достоинство `TWICImage` заключается в универсальности, ведь WIC поддерживает внушительный перечень форматов растровых рисунков: BMP, GIF, ICO, JPEG, PNG, TIFF и Windows Media Photo. К сожалению, недостатков (а точнее, у инкапсулированного в `TWICImage` механизма WIC) также хватает. Во-первых, повышенная ресурсоемкость, свойственная всем приложениям для .NET. Во-вторых, относительная нерасторопность по сравнению со специализированными классами VCL. В-третьих, малые возможности по управлению рисунками — функционал ограничивается загрузкой, сохранением и графическим выводом.

Класс `TWICImage` поддерживает загрузку/сохранение рисунка из файла (`LoadFromFile` и `SaveToFile`), потока (`LoadFromStream` и `SaveToStream`) и буфера обмена (`LoadFromClipboardFormat` и `SaveToClipboardFormat`). Ко всему прочему `TWICImage` позволяет осуществлять полиморфное представление с помощью метода `Assign()`.

Ряд свойств поддерживает доступ к дескрипторам и интерфейсам WIC (табл. 32.5).

***Таблица 32.5.** Свойства и методы `TWICImage`*

Свойство/метод	Описание
`property Handle: IWicBitmap;`	Обеспечивает доступ к интерфейсу `IWICBitmap`, позволяющему обращаться к низкоуровневым функциям WIC
`property ImageFormat: TWICImageFormat;` `type TWICImageFormat = (wifBmp, wifPng, wifJpeg, wifGif, wifTiff, wifWMPhoto, wifOther);`	Управление форматом. Позволяет идентифицировать формат загруженного рисунка и изменить формат при сохранении рисунка
`property EncoderContainerFormat: TGUID;`	Позволяет управлять форматом рисунка с помощью глобального уникального идентификатора формата GUID
`class property ImagingFactory: IWICImagingFactory;`	Получение интерфейса фабрики класса

Коллекция изображений *TImageList*

В составе компонентов Delphi имеется контейнер, специализирующийся на одновременном хранении списка графических образов, — это компонент `TImageList`. Класс описан в модулях `Controls` и `ImgList`, компонент проживает на странице **Win32** палитры компонентов Delphi. На роль рисунков могут претендовать только одинаковые по габаритам растровые файлы bmp, gif, tiff, jpg, png, ico и метафайлы emf и wmf. Контейнер изображений обладает удобным интерфейсом, позволяющим предоставлять доступ к загруженным в него рисункам другим элементам управления.

Контейнер предъявляет ряд требований к обслуживаемым изображениям. В первую очередь это предельные геометрические размеры рисунков. По умолчанию это картинка 16 × 16 пикселов. Размеры устанавливаются парой свойств:

```
property Height: Integer; //высота
property Width: Integer;  //ширина
```

Если же размеры рисунка превосходят значения, указанные в свойствах `Height` и `Width`, компонент предложит на выбор два варианта развития событий: разбиение изображения на несколько частей или сжатие его до приемлемых границ. Строгость в подходе к габаритам пиктограмм объясняется особенностями обслуживания переданных в `TImageList` изображений. В памяти элемента управления рисунки хранятся не по отдельности, а в виде единого блока (рис. 32.4). Другими словами, все заносимые в контейнер картинки склеиваются в единый растровый образ. Когда программист обращается к пиктограмме по ее индексу, компонент (зная, что все ресурсы обладают одинаковыми размерами) быстро находит в матрице область требуемого рисунка и возвращает соответствующие ему пикселы.

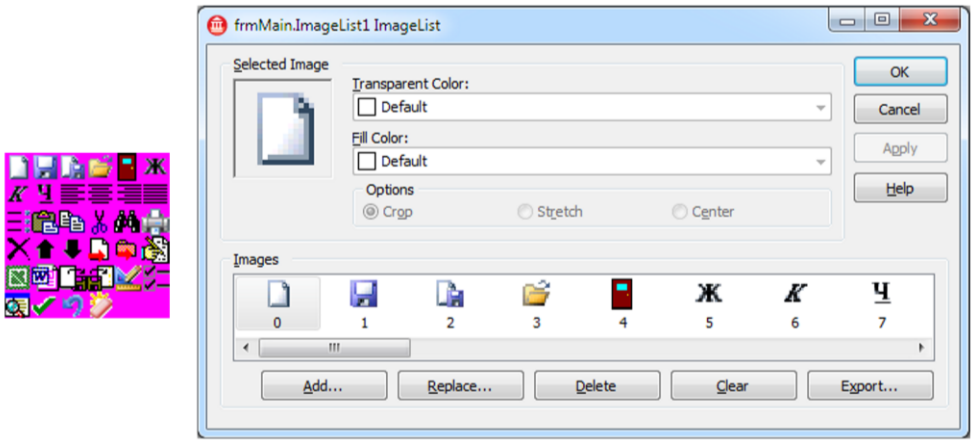


Рис. 32.4. Редактор коллекции изображений компонента `TImageList`

Кроме собственно цветных изображений, контейнер может содержать и аналогичное число масок (монохромных битовых образов). Благодаря маске мы получаем возможность вывода на экран значка с эффектом прозрачности — маска складывается с основным рисунком, причем черные пикселы маски заменяются соответствующим цветом поверхности.

Для определения типа списка стоит обратиться к свойству:

```
property ImageType: TImageType; //по умолчанию itImage
type TImageType = (itImage, itMask);
```

Значение `itImage` — список изображений, который не содержит маски; `itMask` — список изображений, который хранит маски.

Загрузка образов в контейнер

Предусмотрены два способа загрузки рисунков в контейнер `TImageList`. Во время визуального проектирования нам на помощь приходит интегрированный редактор компонента. Для вызова редактора достаточно обратиться к контекстному меню компонента либо дважды щелкнуть по нему кнопкой мыши (см. рис. 32.4). Интерфейс редактора интуитивно понятен и не требует дополнительных комментариев.

Если же возникла необходимость управления содержимым компонента во время выполнения программы, то нам не обойтись без поддержки методов класса `TImageList`.

Добавление к списку значка осуществляет функция

```
function AddIcon(Image: TIcon): Integer;
```

Для одновременного добавления в список очередной пиктограммы `Image` и ее маски `Mask` подойдет метод

```
function Add(Image, Mask: TBitmap): Integer;
```

Как уже отмечалось ранее, маска позволяет определять, какие из пикселов основного изображения станут прозрачными. Если маска не требуется, то передайте во второй параметр неопределенный указатель `nil`. Вместо контуров маски допускается определить цвет прозрачности. Для этого понадобится другая функция:

```
function AddMasked(Image: TBitmap; MaskColor: TColor): Integer;
```

Все три функции добавляют изображение в конец списка и в качестве результата возвращают индекс пиктограммы в списке.

Если логика программы требует вставки пиктограммы в определенное место списка, то целесообразно воспользоваться одной из процедур, представленных в табл. 32.6.

***Таблица 32.6.** Методы добавления картинок в коллекцию*

Метод	Описание
`procedure InsertIcon(Index: Integer; Image: TIcon);`	Вставляет значок `Image` в позицию `Index`
`procedure Insert(Index: Integer; Image, Mask: TBitmap);`	Вставляет изображение `Image` и маску `Mask` в позицию `Index`
`procedure InsertMasked(Index: Integer; Image: TBitmap; MaskColor: TColor);`	Заменяет изображение `Image` в позиции `Index` и генерирует для него маску на основе цвета `MaskColor`

Замечание

Ответственность за включение или отключение маски несет свойство `Masked`. Если это свойство установлено в `True`, результирующее изображение будет представлять комбинацию рисунка и маски.

Если позиция пиктограммы в списке вас не устраивает, то для ее перемещения на более достойное место потребуется процедура

```
procedure Move(CurIndex, NewIndex : Integer);
```

Метод передвинет образ из позиции `CurIndex` в позицию `NewIndex`.

Для замены одного рисунка другим используются процедуры, представленные в табл. 32.7.

Таблица 32.7. *Методы замены картинок в коллекции*

Метод	Описание
procedure ReplaceIcon(Index: Integer; Image: TIcon);	Заменяет значок с индексом Index новой картинкой Image
procedure Replace(Index: Integer; Image, Mask: TBitmap);	Заменяет пиктограмму с индексом Index, изображением Image и маской Mask
procedure ReplaceMasked(Index: Integer; NewImage: TBitmap; MaskColor: TColor);	Заменяет изображение с номером Index новым растровым образом NewImage и генерирует для него маску на основе цвета MaskColor

Коллекция изображений TImageList умеет загружать данные из ресурса. Типом данных TResType предусмотрено существование трех вариантов таких данных: растровый рисунок (rtBitmap), курсор (rtCursor) и значок (rtIcon). К наиболее универсальным методам обслуживания ресурса относится пара перезагружаемых функций

```
function GetInstRes(Instance: THandle; ResType: TResType;
         const Name: string; Width: Integer; LoadFlags: TLoadResources;
         MaskColor: TColor): Boolean; overload;
function GetInstRes(Instance: THandle; ResType: TResType;
            ResID: DWORD; Width: Integer; LoadFlags: TLoadResources;
            MaskColor: TColor): Boolean; overload;
```

Здесь: Instance — дескриптор контейнера; ResType — тип ресурса; MaskColor — цвет маски. Разница между методами — в способе указания ресурса. В первом случае нам потребуется его имя (параметр Name), во втором — числовой идентификатор (параметр ResID). С помощью комбинации флагов LoadFlags определяются особенности загрузки ресурса. Например, флаг lrMonoChrome указывает на то, что нам требуется черно-белое изображение, флаг lrFromFile говорит, что данные загружаются из внешнего файла.

Практически все остальные методы доступа к внешнему ресурсу выступают в роли надстроек над функцией GetInstRes(). Если ресурс интегрирован в тот же файл, которому принадлежит компонент ImageList, или к нашему процессу статически подключена внешняя библиотека с интересующим нас ресурсом, то для загрузки данных следует применять функции:

```
function GetResource(ResType: TResType; const Name: string;
                     Width: Integer;
                     LoadFlags: TLoadResources;
                     MaskColor: TColor): Boolean;
function ResourceLoad(ResType: TResType; const Name: string;
                      MaskColor: TColor): Boolean;
```

Если мы планируем поместить в список изображений данные из файла с именем Name, то вместо GetInstRes() проще вызвать ее упрощенный аналог:

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

При наличии дескриптора внешнего контейнера, в котором хранится требуемый ресурс (например, динамической библиотеки или пакета), с ролью загрузчика превосходно справится функция:

```
function ResInstLoad(Instance: THandle; ResType: TResType;
                     const Name: string; MaskColor: TColor): Boolean;
```

В случае успешного завершения все рассмотренные функции возвратят `true`.

В момент любого изменения содержимого списка генерируется событие

```
property OnChange: TNotifyEvent;
```

Об общем количестве пиктограмм в контейнере нас проинформирует свойство

```
property Count: Integer; //только для чтения
```

Полная очистка коллекции изображений осуществляется процедурой

```
procedure Clear;
```

Для удаления одного рисунка из списка необходимо воспользоваться методом

```
procedure Delete(Index: Integer);
```

Единственный параметр процедуры — индекс удаляемого рисунка.

Особенности отображения значков

К вопросу отображения рисунка на поверхности элементов управления компонент `TImageList` подошел со всей ответственностью. Вне зависимости от выбранной пользователем цветовой схемы рабочего стола компонент способен прорисовать свою пиктограмму на любом элементе управления с обеспечением прозрачности и эффекта фокуса ввода. В зависимости от сложности задачи результат достигается либо за счет обычного копирования пиктограммы в нужную область, либо посредством проведения тернарной растровой операций между тремя объектами: поверхностью элемента управления, значком и кистью заливки (маской). Порядок вывода значка на поверхности элемента управления находится в прямой зависимости от состояния свойства:

```
property DrawingStyle: TDrawingStyle;
```

Предусмотрены 4 режима работы компонента (табл. 32.8). По умолчанию компонент настроен на нормальный вывод значков (стиль `dsNormal`).

Таблица 32.8. *Возможные значения* `TDrawingStyle`

Значение	Описание	Примечание
`dsFocus`	Эффект фокуса ввода. К растру рисунка подмешивается 25% цвета, указанного в свойстве `BlendColor`	Эффект проявится только для набора изображений, содержащего маску
`dsSelected`	Эффект выбранного объекта. К растру подмешивается 50% цвета, указанного в свойстве `BlendColor`	
`dsNormal`	Обычный режим. Вывод изображения осуществляется с учетом цвета `BkColor`. Если значение `BkColor` не установлено (`clNone`), то для обеспечения эффекта прозрачности значок опирается на маску	
`dsTransparent`	Эффект прозрачности. Достигается без учета указанного в `BkColor` цвета	

Над обеспечением видимости эффекта фокуса ввода на значке трудится еще одно свойство:

```
property BlendColor: TColor;
```

Это свойство учитывается только в тех случаях, когда свойство `DrawingStyle` установлено в `dsFocus` или `dsSelected`. Если элемент управления выбран пользователем, то при проведении растровой операции этот цвет подмешивается в растровую операцию.

Список изображений `TImageList` способен не только хранить растровые образы, но и выводить их на поверхности элементов управления, обладающих своим холстом. Для этого предназначены процедуры:

```
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer;
               Enabled: Boolean=True); overload;
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer;
               ADrawingStyle: TDrawingStyle;
               AImageType: TImageType; Enabled: Boolean=True); overload;
```

Здесь: `Canvas` — канва элемента управления; `X`, `Y` — координаты начала вывода рисунка; `Index` — номер рисунка; в состоянии `true` параметра `Enabled` процедура выводит цветное изображение, в состоянии `false` — его монохромную версию; от состояния `AImageType` зависит, будет ли использоваться маска или нет.

Еще одна пара перегружаемых процедур умеет выводить на поверхности другого элемента изображение, перекрытое оверлейным образом:

```
procedure DrawOverlay(Canvas: TCanvas; X, Y: Integer;
                      ImageIndex: Integer; Overlay: TOverlay;
                      Enabled: Boolean = True); overload;
procedure DrawOverlay(Canvas: TCanvas; X, Y: Integer;
                      ImageIndex: Integer; Overlay: TOverlay;
                      ADrawingStyle: TDrawingStyle;
            AImageType: TImageType; Enabled: Boolean = True); overload;
```

> ***Внимание!***
>
> Перед вызовом процедуры `DrawOverlay()` не забывайте назначить оверлейный образ. Это осуществляется за счет обращения к методу `Overlay()`.

По сравнению с предыдущими процедурами здесь добавился новый параметр `Overlay`. Это индекс оверлейной маски, он может принимать значение в диапазоне от 0 до 3.

Индекс пиктограммы, которая станет использоваться в роли оверлейной маски, определяется функцией:

```
function Overlay(ImageIndex: Integer; Overlay: TOverlay): Boolean;
```

Оверлейный образ может выводиться поверх основного с обеспечением прозрачности с помощью процедуры `DrawOverlay()`.

Прозрачность

Для того чтобы система определилась, какие именно пикселы пиктограммы надо сделать прозрачными, программист должен либо явным образом указать значение замещаемого цвета, либо назначить маску-шаблон, при бинарном сложении которой с основным рисунком достигается эффект прозрачности.

Если мы пойдем по первому пути, то прозрачный цвет определяется для всех находящихся в контейнере значков в свойстве

```
property BkColor: TColor;
```

Второй подход позволит определить прозрачные пикселы для каждой пиктограммы отдельно. В этом случае нужно задействовать любой из уже известных нам методов `AddMasked()`, `InsertMasked()`, `Add()` или `Insert()`. Второе решение обладает наибольшей гибкостью, т. к. позволяет делать прозрачными любые пикселы изображения, а не только пикселы одного определенного цвета.

Для включения режима прозрачности установите в `true` свойство

```
property Masked: Boolean;
```

> ***ЗАМЕЧАНИЕ***
>
> Вывод на экран прозрачного рисунка осуществляется в два этапа. На первом этапе производится логическое умножение (операция `AND`) пикселов рисунка и ее маски. Затем полученное изображение переносится на поверхность элемента управления, причем пикселы изображения объединяются с пикселами поверхности на основе исключающего ИЛИ (операция `XOR`).

Экспорт значков из контейнера

Наиболее близкая по духу к функциям GDI пара методов позволяет получить дескрипторы хранящихся в контейнере изображений:

```
function GetImageBitmap: HBITMAP;
function GetMaskBitmap: HBITMAP;
```

Первый метод возвращает дескриптор блока основных изображений, второй — дескриптор маски.

При необходимости извлечь из контейнера отдельную пиктограмму в виде родных для программиста Delphi растрового образа `TBitmap` или значка `TIcon` обратимся к одной из следующих процедур:

```
procedure GetBitmap(Index: Integer; Image: TBitmap);
procedure GetIcon(Index: Integer; Image: TIcon); overload;
procedure GetIcon(Index: Integer; Image: TIcon;
        ADrawingStyle: TDrawingStyle, AImageType: TImageType); overload;
```

В параметре `Index` указываем индекс пиктограммы, параметр `Image` возвратит изображение в формате `TBitmap` (процедура `GetBitmap()`) или в формате `TIcon` (процедура `GetIcon()`). Более полная версия перегружаемого метода `GetIcon()` потребует двух дополнительных параметров, оказывающих влияние на внешний вид возвращаемого рисунка. В первую очередь это параметр `ADrawingStyle`, определяющий стиль вывода значка на экран (`dsFocus`, `dsSelected`, `dsNormal` или `dsTransparent`), и параметр `AimageType`, конкретизирующий, что именно нас интересует — картинка `itImage` или ее маска `itMask`.

ГЛАВА 33

Сложные графические задачи

В основе графических решений визуальной библиотеки компонентов Delphi лежит объектно-ориентированное воплощение контекста графического устройства — холст `TCanvas`. Класс инкапсулирует ключевой набор функций Windows API, обеспечивающих работу с деловой графикой и весьма упрощает работу программиста. Однако класс `TCanvas` далеко не всесилен, в его составе по тем или иным причинам не нашлось места для достаточно существенного перечня функций GDI, в их числе функции, определяющие режимы отображения Windows, страничные и мировые преобразования, сокращены возможности по работе с регионами отсечения и т. д.

В этой главе мы поговорим о ряде программных решений, позволяющих существенно расширить возможности графического вывода в приложениях Delphi. В основе всех этих решений лежит использование функций прикладного интерфейса программирования Windows API в союзе с классом `TCanvas`.

Растровые операции

Представьте себе все доступные пикселы экрана в виде двумерного массива D размером $x \times y$. Процесс прорисовки пиксела заключается в указании его координат и передаче по этому адресу нового значения цвета P, скажем: $D\ x, y\ = P$. Все очень просто и как нельзя лучше подойдет для вывода где-то на экране точки с заданным цветом. Но в реальной жизни все обстоит гораздо сложнее. Например, когда краска с кисти художника ложится на холст, она взаимодействует с поверхностью холста или с другими красками, изменяя результирующий цвет. Почему бы не реализовать такую интересную возможность и в программах Delphi?

Раз перед присвоением пикселу нового цвета мы намерены учитывать его предыдущее состояние, то предложенная ранее упрощенная формула $D\ x, y\ = P$ преобразуется в более серьезную $D\ x, y\ = f\ \ D\ x, y\ , P\ \ $. Здесь f — функция, описывающая взаимодействие старого $D\ x, y\ $ и нового P цветов пиксела. Подобная функция двух параметров называется *бинарной растровой операцией* (binary Raster Operation, ROP2).

> ***Замечание***
>
> Существуют и более сложные растровые операции, именуемые *тернарными* и *кватернарными*.

Управляя процессом вывода растровой картинки на экран, GDI производит поразрядные булевы операции между вовлеченными в процесс объектами. В случае ROP2 в роли участников растровой операции выступают два действующих лица:

- пиксел-получатель *D* (destination);
- пиксел пера *P* (pen).

С только что анонсированными действующими лицами осуществляются следующие логические операции:

- конъюнкция AND (логическое "И");
- дизъюнкция OR (логическое "ИЛИ");
- инверсия NOT (логическое "НЕТ");
- XOR (логическое исключающее "ИЛИ").

Обсуждая ключевой для графической подсистемы Delphi класс `TCanvas`, мы уже упоминали о существовании свойств холста, определяющих порядок проведения растровых операций. В первую очередь это свойство `CopyMode`, управляющее процессом переноса на холст исходного графического образа. По умолчанию свойство установлено в состояние `cmSrcCopy`, что означает прямое копирование пикселов из образа на экран. Кроме того, у пера (класс `TPen`) предусмотрено свойство `Mode`, определяющее порядок сложения пикселов пера и пикселов холста.

Существует и другой способ управления растровыми операциями, он основан на обращении к паре функций Win API:

```
function GetROP2(DC : HDC):integer;
function SetROP2(DC : HDC; DrawMode : integer):integer;
```

Первая из функций информирует программиста о том, какой именно режим ROP2 в текущий момент установлен у графического устройства с дескриптором контекста `DC`. По умолчанию это режим `R2_COPYPEN`, соответствующий обычному переносу цвета пера в пиксел-приемник. Для смены растровой операции предназначена функция `SetROP2()`. Для этого она вооружена параметром `DrawMode`, в который и направляется подходящий код ROP2 (табл. 33.1). В случае успешного выполнения функция возвращает код предыдущего состояния ROP2.

Таблица 33.1. *Коды основных растровых операций*

Код ROP2	Операция	Описание
R2_BLACK	D=0	Пиксел-получатель D устанавливается в черный цвет
R2_COPYPEN	D=P	Пиксел D принимает цвет пера P
R2_MASKNOTPEN	D=D AND (NOT P)	Конъюнкция цветов пиксела D и инвертированного пера P
R2_MASKPEN	D=D AND P	Конъюнкция пера P и приемника D
R2_MASKPENNOT	D=(NOT D) AND P	Конъюнкция цветов пера P и инвертированного пиксела D
R2_MERGENOTPEN	D=D OR (NOT P)	Дизъюнкция пиксела D и инвертированного пера P
R2_MERGEPEN	D=D OR P	Пиксел D представляет собой дизъюнкцию цветов пера P и пиксела D

***Таблица 33.1** (окончание)*

Код ROP2	Операция	Описание
R2_MERGEPENNOT	D=(NOT D) OR P	Дизъюнкция цветов инвертированного пиксела D и пера P
R2_NOP	D=D	Цвет пиксела D не изменяется
R2_NOT	D=NOT D	Инверсия цвета пиксела D
R2_NOTCOPYPEN	D=NOT P	Цвет пиксела D инвертируется по отношению к цвету пера P
R2_NOTMASKPEN	D=NOT (D AND P)	Инверсия операции R2_MASKPEN
R2_NOTMERGEPEN	D=NOT (D OR P)	Инверсия операции R2_MERGEPEN
R2_NOTXORPEN	D=NOT (D XOR P)	Инверсия операции R2_XORPEN
R2_WHITE	D=1	Пиксел D устанавливается в белый цвет
R2_XORPEN	D=D XOR P	Пиксел D и перо P объединяются по правилу исключения "ИЛИ"

Опираясь на возможности операций ROP2, можно получить весьма нетривиальные результаты. Надеюсь, вы не забыли пример из листинга 31.2, в котором мы управляли цветом формы, изменяя положение бегунков трех компонентов TTrackBar (каждый из компонентов отвечал за свою цветовую составляющую RGB). Этот пример можно существенно улучшить за счет бинарной растровой операции.

Итак, в нашем распоряжении три компонента TTrackBar с предельным значением свойства Max, равным 255. Компонент tbRed отвечает за значение красного цвета, tbGreen — зеленого и tbBlue — синего. Обращаемся к событию OnPaint() формы проекта и в его рамках рисуем три эллипса (соответственно красного, зеленого и синего цветов) так, чтобы они частично перекрывали друг друга (листинг 33.1).

Листинг 33.1. Пример управления сложением цветов с помощью ROP2

```
procedure TForm1.Form1Paint(Sender: TObject);
var ROP2:Integer;
begin
  with Form1.Canvas do
  begin
    ROP2:=GetRop2(PaintBox1.Canvas.Handle); //запоминаем исходное
                                            //состояние
    Brush.Color:=clBlack;
    FillRect(ClientRect);

    SetRop2(PaintBox1.Canvas.Handle,R2_MERGEPEN); //новый режим ROP2
    //рисуем красный эллипс
    Brush.Color:=RGB(tbRed.Position,0,0);
    Ellipse(1,1,Form1.ClientWidth*2 div 3, Form1.ClientHeight*2 div 3);

    //рисуем зеленый эллипс
    Brush.Color:=RGB(0,tbGreen.Position,0);
```

```
    Ellipse(Form1.ClientWidth div 3,1, Form1.ClientWidth,
            Form1.ClientHeight*2 div 3);

    //рисуем синий эллипс
    Brush.Color:=RGB(0,0,tbBlue.Position);
    Ellipse(Form1.ClientWidth div 6, Form1.ClientHeight div 3,
      Form1.ClientWidth-Form1.ClientWidth div 6, Form1.ClientHeight);

    SetRop2(PaintBox1.Canvas.Handle,ROP2); //восстанавливаем состояние
  end;
end;
```

Самое важное в примере то, что перед началом прорисовки эллипсов мы устанавливаем растровую операцию холста в состояние `R2_MERGEPEN`, что заставит систему осуществлять логическое сложение пиксела холста и пиксела эллипса. В результате дизъюнкции пикселов красного и зеленого цветов мы получаем желтый цвет, сложение зеленого и синего цветов приводит к появлению голубого цвета, и т. д. К сожалению, черно-белая печать книги не позволяет передать все оттенки цветов, поэтому на рис. 33.1 пришлось оставить пояснительные подписи...

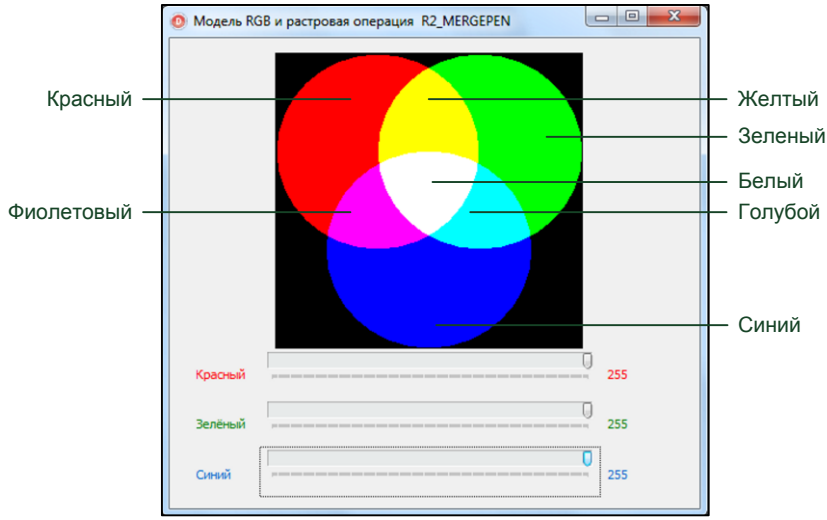


Рис. 33.1. Растровая операция `R2_MERGEPEN` с пикселами красного, зеленого и синего цветов

Управление прозрачностью

Рассуждая о способе описания 32-битного цвета, мы отметили факт существования специального байта, предназначенного для хранения коэффициента прозрачности. В этом случае, кроме обработки красной, зеленой и синей составляющих, система обязана анализировать состояние альфа-канала и осуществлять растровую операцию с учетом его настоятельных пожеланий. Абсолютно непрозрачному пикселу соответствует значение `$FF` (255 в десятичном представлении), а нулевое значение указывает на полную прозрачность.

Замечание

По сути, альфа-наложение представляет собой очередную разновидность растровой операции, но теперь при формировании результирующего пиксела учету подлежит дополнительный альфа-коэффициент.

Очень гибкими возможностями по управлению прозрачностью обладает функция

```
function AlphaBlend(dcDest: HDC;
          XOriginDest, YOriginDest, WidthDest, HeightDest: Integer;
          dcSrc: HDC;
          XOriginSrc, YOriginSrc, WidthSrc, HeightSrc: Integer;
          BlendFunction: TBlendFunction): BOOL;
```

Функция осуществляет перенос пикселов из контекста источника (описываемого дескриптором `dcSrc`) в контекст получателя (дескриптор `dcDest`) с заданной степенью прозрачности. Параметры `XOriginDest`, `YOriginDest`, `WidthDest`, `HeightDest` описывают геометрию приемной области, параметры `XOriginSrc`, `YOriginSrc`, `WidthSrc`, `HeightSrc` характеризуют область-источник. Изюминка функции в последнем параметре, в него направляется структура `TBlendFunction` (листинг 33.2), при посредничестве которой программист определяет порядок взаимодействия между пикселами контекста-источника и контекста-получателя.

Листинг 33.2. Объявление структуры `TBlendFunction`

```
type TBlendFunction = packed record
    BlendOp: BYTE;             {всегда AC_SRC_OVER}
    BlendFlags: BYTE;          {зарезервировано — всегда 0}
    SourceConstantAlpha: BYTE; {альфа-коэффициент от 0 до 255}
    AlphaFormat: BYTE; {формат наложения:
           0 — постоянный альфа-коэффициент для всех пикселов;
           AC_SRC_ALPHA — альфа-коэффициент отдельных пикселов}
end;
```

Если формат альфа-наложения (параметр `AlphaFormat`) равен нулю, то `SourceConstantAlpha` определяет степень прозрачности для всех пикселов растра. Диапазон ограничен значениями 0—255, где 0 соответствует абсолютной прозрачности, а 255 — полной непрозрачности. Если поле `AlphaFormat` структуры `TBlendFunction` принимает значение `AC_SCR_ALPHA`, то функция приобретает возможность оценивать альфа-коэффициент для каждого пиксела в отдельности.

Допустим, что в нашем распоряжении есть объект `BMP:TBitmap`, в который загружен растровый рисунок. Этот рисунок нам предстоит отобразить на поверхности формы, постепенно изменяя коэффициент прозрачности. В этом случае стоит воспользоваться исходным кодом из листинга 33.3.

Листинг 33.3. Управление прозрачностью вывода растрового изображения

```
var BMP:TBitmap;

procedure TForm1.FormPaint(Sender: TObject);
var BlendFunction:TBlendFunction;
    Alpha,Y,X : word;
    DC, memDC:HDC; //дескрипторы контекста
```

```
const AlphaStep=5; //шаг приращения коэффициента прозрачности
begin
  DC:=Form1.Canvas.Handle;  //дескриптор холста

  with BlendFunction do  //настройка параметров прозрачности
  begin
    BlendOp:=AC_SRC_OVER;
    BlendFlags:=0;
    SourceConstantAlpha:=0;
    AlphaFormat:=0;
  end;

  memDC:=CreateCompatibleDC(DC);  //создадим совместимый контекст в памяти
  SelectObject(memDC,BMP.Handle); //заносим в память картинку BMP:TBitmap

  X:=0; Y:=0; Alpha:=0;
  while Alpha<=255 do  //изменяем коэффициент прозрачности в цикле
  begin
    Windows.AlphaBlend(DC, X, Y, BMP.Width, BMP.Height,
              memDC, 0, 0, BMP.Width, BMP.Height, BlendFunction);

    INC(X,BMP.Width);
    if X>Form1.ClientWidth-BMP.Width then //вычисляем место вывода
    begin
      INC(Y,BMP.Height);
      X:=0;
    end;
    INC(Alpha,AlphaStep);
    BlendFunction.SourceConstantAlpha:=Alpha;
  end;
  DeleteDC(memDC); //освобождаем совместимый контекст
end;
```

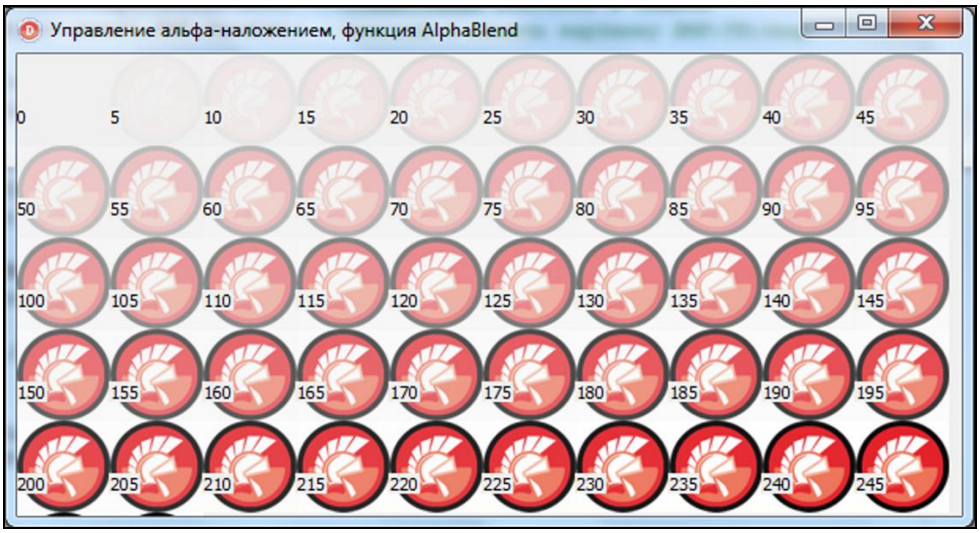


Рис. 33.2. Вывод битового образа с разными коэффициентами прозрачности

Основная особенность представленного в листинге 33.3 кода в том, что во время перерисовки формы мы заносим растровый образ BMP в искусственно созданный с помощью функции Win API `CreateCompatibleDC()` контекст графического устройства в памяти. Этот контекст обладает характеристиками, абсолютно идентичными характеристикам холста формы, поэтому мы его называем *совместимым*. Далее в цикле `while..do` мы выводим битовый образ на поверхность формы, постепенно уменьшая коэффициент прозрачности, поэтому наш рисунок мало-помалу "проявляется" на белом фоне окна (рис. 33.2).

Системы координат и режимы отображения

До сих пор, осуществляя графический вывод (например, на поверхности формы), мы с вами исходили из предположения, что начало координат располагается в левом верхнем углу формы и координатная ось *x* направлена вправо, а координатная ось *y* — вниз. При этом мы были уверены, что единственно возможной единицей измерения должен выступать пиксел. Таким образом, обратившись к методу `TextOut(5, 5, 'Hello, word!')`, мы рассчитываем увидеть надпись "Hello, world!" в левом верхнем углу формы, причем левый верхний пиксел текста должен находиться в точке с координатами (5, 5). Но настал тот час, когда мы поймем, что это весьма упрощенное представление о системе координат Windows.

Современные операционные системы Microsoft Windows XP/Vista/7 для определения областей на поверхности вывода используют четыре системы координат: мировые координаты, страничные координаты, координаты устройства и физические координаты.

Мировые координаты предназначены для осуществления *двумерных аффинных преобразований*. Для начинающего программиста Delphi этот термин наверняка нов, поэтому пока достаточно знать, что аффинные преобразования используются для осуществления разнообразных визуальных эффектов (операций вращения растра, сдвига, скручивания и т. п.).

Страничные координаты соответствуют координатам любой области, обладающей контекстом устройства. Например, в проектах Delphi, построенных на фундаменте VCL, контекстом описывается поверхность формы (класс `TForm`) и поверхность области для рисования (класс `TPaintBox`). Это система координат, устанавливаемая по умолчанию для всех контекстов во всех версиях Windows, так что, даже не подозревая этого, все свои графические задачи мы до сих пор решали именно в этой системе.

Координаты устройства определяются техническими характеристиками аппаратного устройства, с которым работает Windows. Здесь все устроено привычным для нас образом: в качестве единицы измерения всегда применяется пиксел, начало координат находится в левой верхней точке, ось *x* направлена слева направо, а ось *y* проведена сверху вниз.

Физические координаты напрямую используются драйвером графического устройства и явным образом недоступны из обычных программ.

> ***ЗАМЕЧАНИЕ***
>
> Во время программирования с помощью методов холста `TCanvas` наши проекты по умолчанию ориентируются на страничные координаты. Для перехода к мировым координатам потребуется помощь функций Win API.

Все четыре координатные системы взаимоувязаны. И, если мы что-то нарисуем в мировых координатах, прежде чем изображение появится на графическом устройстве, Windows последовательно преобразует рисунок из мировых координат в страничные, затем из страничных в координаты устройства и в самом конце (опираясь на поддержку драйвера графического устройства) трансформирует координаты устройства в физические.

Методы класса TCanvas в качестве параметров требуют передачи логических координат. Направляя Delphi строку кода TextOut(5, 5, 'Hello, word!'), мы информируем систему вывода о том, что рассчитываем увидеть надпись приветствия в точке с координатами (5, 5). Но в этом случае совсем не обязательно речь ведется о пикселах, как мы привыкли до сих пор. Вполне вероятно, что контекст графического устройства в качестве единицы измерения применяет дюймы, миллиметры или другую единицу. Более того, может оказаться, что точка начала отсчета координат появится не в привычном для нас левом верхнем углу рабочей области окна, а где-то в другом месте.

Говоря о графическом интерфейсе пользователя, мы не раз применяли термин *"окно"*. С точки зрения графического вывода окно — это некоторый прямоугольный регион в страничной системе координат. На практике это может быть клиентская область формы TForm стандартного проекта Delphi или поверхность любого другого элемента управления. При работе со страничными координатами надо усвоить еще один термин — *область вывода*. В отличие от окна, область вывода — это прямоугольник, но уже в системе координат устройства, например вашего видеоадаптера. Именно в эту систему координат будет спроецирован образ из страничной системы. Управляя начальными координатами и размерами окна, мы получаем возможность определять отображаемую часть изображения. В свою очередь, настраивая область просмотра, мы назначаем место вывода отображаемой части изображения на поверхности устройства.

Перенос начала координат

Операционная система позволяет изменять местоположение точки начала отсчета координат как области вывода, так и окна. Для этого предусмотрены две функции Windows API: SetViewportOrgEx() и SetWindowOrgEx(). Для установки начала координат области вывода, описываемого контекстом устройства DC, предназначен метод:

```
function SetViewportOrgEx(DC : HDC; X, Y : Integer;
                          Point : PPoint) : Boolean;
```

Здесь X, Y — координаты нового начала координат области вывода в единицах измерения устройства. Другими словами, это физические координаты, в качестве единицы измерения которых выступает пиксел. Point — указатель на необязательную структуру TPoint, в нее сохранится предыдущее значение начала координат.

Попробуем научиться переносить точку начала координат с помощью функции SetViewportOrgEx(). Для этого создайте новый проект и в секции private объявите переменную P:TPoint. После этого нам понадобится поддержка двух событий формы. Событие OnMouseDown(), возникающее по щелчку кнопкой мыши по поверхности формы, позволит нам запомнить координаты указателя мыши в переменной P в момент щелчка. Последствия переноса отразятся во время перерисовки клиентской области формы, для этого задействуется событие OnPaint(). Код обработчиков событий представлен в листинге 33.4. После переноса начала координат в новое место мы рисуем две линии, точно соответствующие новым координатным осям X и Y. Затем, чуть ниже точки начала координат, выводим подпись "(0, 0)".

Листинг 33.4. Перенос начала координат

```
var P:TPoint;
...
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  P:=Point(X,Y); //сохранили координаты
  Form1.Repaint; //дали команду перерисовки формы
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  with Form1.Canvas do
  begin
    SetViewportOrgEx(Handle,P.X,P.Y,nil); //перенос центра координат
    MoveTo(-200, 0); LineTo(200,0); //рисуем ось x из точки -200 в 200
    MoveTo(0,-200);  LineTo(0,200); //рисуем ось y из точки -200 в 200
    TextOut(2,2,'(0, 0)');          //подпись начала координат
  end;
end;
```

При желании точно такой же результат может быть получен с помощью функции

```
function SetWindowOrgEx(DC: HDC; X,Y: Integer; Point : PPoint) : Boolean;
```

Однако на этот раз функция определяет начало координат вывода окна (а не проекции окна, как ее коллега по цеху `SetViewportOrgEx`). Еще одно отличие этой функции `SetWindowOrgEx()` от `SetViewportOrgEx()` в том, что здесь аргументы `X` и `Y` задаются в логических координатах. Таким образом, метод позволяет манипулировать не только пикселами, но и любыми другими удобными для пользователя единицами измерения, например миллиметрами и дюймами.

Управление страничными координатами

Логические координаты созданы с целью максимального упрощения нашей работы. А операционная система без вмешательства извне сумеет преобразовать логические координаты в физические координаты принтера или дисплея. Мы уже привыкли к тому, что по умолчанию система координат берет начало в левом верхнем углу рабочей поверхности — в точке с координатами (0, 0). Горизонтальная ось называется осью *x* и проведена из крайней верхней точки поверхности рисования в крайнюю правую точку. Ось *y* берет начало в той же точке (0, 0) и идет вертикально вниз. Теперь пора узнать, что этот самый распространенный режим отображения называется *текстовым* (в терминах Windows ему соответствует идентификатор `MM_TEXT`). Причина названия режима "текстовым" заключается в том, что оси *x* и *y* системы координат режима `MM_TEXT` проведены точно так, как мы читаем текст — слева направо, сверху вниз. Как видите, в данном случае физические единицы устройства — пикселы — полностью совпадают с нашими логическими координатами. Однако в Windows, помимо текстового режима, реализовано еще целых семь режимов отображения (табл. 33.2).

По сути, метрические режимы изображения — это заранее подготовленные шаблоны, устанавливающие фиксированные протяженности окна и области вывода. Для России наиболее подходят режимы `MM_LOMETRIC` и `MM_HIMETRIC`, которые нацелены на работу с миллиметрами. Логическая единица грубого режима `MM_LOMETRIC` соответствует 0,1 мм. Прецизионный режим `MM_HIMETRIC` различает величину 0,01 мм.

Таблица 33.2. *Режимы отображения Windows*

Классификация	Режим отображения	Логические единицы	Направления осей
Текстовый режим	MM_TEXT	Пиксел	*x* — вправо, *y* — вниз
Метрические режимы	MM_LOMETRIC	0,1 мм	*x* — вправо, *y* — вверх
	MM_HIMETRIC	0,01 мм	
	MM_LOENGLISH	0,01 дюйма (примерно 0,254 мм)	
	MM_HIENGLISH	0,001 дюйма (примерно 0,0254 мм)	
	MM_TWIPS	1/1440 дюйма (примерно 0,0176 мм)	
Пользовательские режимы	MM_ISOTROPIC	Настраиваемые	Настраивается
	MM_ANISOTROPIC	Настраиваемые	Настраивается

Ключевая особенность пользовательских режимов в том, что они способны изменять направления осей абсцисс и ординат. Еще одна положительная черта режимов MM_ISOTROPIC и MM_ANISOTROPIC в том, что в отличие от стандартных метрических режимов пользовательские режимы отображения предоставляют программисту право самостоятельно назначать логическую единицу измерения. Тем самым мы получаем дополнительную степень свободы, благодаря которой сможем создавать программное обеспечение, например нацеленное на промышленное черчение или на построение топографических систем с единицами измерения, на порядки отличающимися от фундаментальных дюймов и миллиметров. Различие между двумя пользовательскими режимами раскрывается в их названиях. Изотропный (одинаковый) режим отображения разрешает программисту самостоятельно назначать логическую единицу измерения, но только при условии, что логические единицы измерений осей *x* и *y* будут эквивалентны друг другу. Напротив, анизотропный (неодинаковый) режим допускает, чтобы логические единицы измерений горизонтальной и вертикальной осей были различными, например: в качестве единиц измерения для оси *x* программист может назначить миллиметры, а для *y* — километры.

За управление режимом отображения отвечает пара функций Win API:

```
function GetMapMode(DC : HDC) : Integer;
function SetMapMode(DC : HDC; MapMode : Integer) : Integer;
```

Первая функция информирует программиста о текущем режиме контекста, а вторая изменяет режим отображения. В случае успешного выполнения функция SetMapMode() возвращает значение предыдущего режима отображения, в противном случае 0.

Переключение режимов отображения с параллельным изменением логической единицы очень часто применяется в разнообразных текстовых и графических редакторах, системах автоматизированного проектирования, картографических приложениях. Одним словом, во всех программных продуктах, ориентированных на стандартные системы измерения. Листинг 33.5 демонстрирует способ начертания сантиметровой линейки, которая может пригодиться, например, для горизонтальной разметки текста в текстовом редакторе.

Листинг 33.5. Перенос начала координат

```
procedure TfrmMain.PaintBox1Paint(Sender: TObject);
var X : Integer;
begin
  with PaintBox1.Canvas do
  begin
    SetMapMode(Handle,MM_TEXT); //обычный (текстовый) режим отображения
    //перенос точки начала координат
    SetViewportOrgEx(Handle, 0, PaintBox1.Height div 2, nil);
    SetMapMode(Handle,MM_LOMETRIC);  //метрический режим отображения

    MoveTo(0,0);  LineTo(1000,0);    //горизонтальная линия линейки
    X:=0;
    repeat
      MoveTo(X,0); LineTo(X,20); //риски линейки
      SetBKMode(Handle,TRANSPARENT);
      Font.Size:=20;
      TextOut(X,10, IntToStr(X div 100); //подпись значения линейки
      INC(X,100);              //приращение X на 100 логических единиц
    until  X>=10000;
  end;
end;
```

Мировые координаты и аффинные преобразования

В первых версиях Windows 95/98/ME понятия мировых координат не существовало, поэтому для поддержки программ старого парка даже в Windows 7 контекст графического устройства по умолчанию устанавливается в страничную систему координат.

Появление мировых координат в современных версиях Windows предоставило программисту удобный механизм осуществления двумерных преобразований на плоскости. Двумерные преобразования основаны на идее отображения одного множества точек в другое, в перечень возможностей преобразований входит как элементарный перенос изображения с одного места на другое без внесения в него каких-либо искажений, так и сложные процедуры растяжения (сжатия), деформации и вращения. Все операции преобразований опираются на математический аппарат, именуемый двумерными *аффинными преобразованиями*.

> ***ЗАМЕЧАНИЕ***
>
> С практической точки зрения разработчика программного обеспечения аффинное преобразование определяет порядок перехода от прямоугольной декартовой системы координат к назначаемой программистом системе (при необходимости даже косоугольной).

Для выяснения, в какой из систем координат установлен контекст графического устройства, обращаемся к функции Windows API

```
function GetGraphicsMode(DC:HDC):integer;
```

информирующей нас о текущей системе координат контекста устройства с дескриптором `DC`. В случае успешного выполнения функция вернет одну из двух констант: `GM_COMPATIBLE` или `GM_ADVANCED`. Значение `GM_COMPATIBLE` свидетельствует о том, что контекст установлен

в страничные координаты, или, как часто говорят, в совместимый графический режим. Константа GM_ADVANCED сигнализирует о том, что система координат контекста настроена на работу в мировых координатах и готова к осуществлению аффинных преобразований.

Для смены системы координат предназначена функция

```
function SetGraphicsMode(DC:HDC; Mode : integer):integer;
```

В первый параметр передается дескриптор контекста графического устройства, во второй — одна из двух упомянутых ранее констант.

Собственно процессом преобразования мировых координат в страничные для контекста устройства DC управляет одна-единственная функция GDI:

```
function SetWorldTransform(DC : HDC; XForm: TXForm) : Boolean;
```

Результат работы функции определяется правилами двумерных аффинных преобразований, передаваемых программистом в матрицу аффинных преобразований XForm, описываемую структурой:

```
type TXForm = packed record
    eM11: Single;
    eM12: Single;
    eM21: Single;
    eM22: Single;
    eDx : Single;
    eDy : Single;
end;
```

Каждое поле структуры входит в математические выражения, определяющие алгоритм преобразования точек. В результате этих преобразований исходная точка (x, y) транслируется в точку с координатами (x', y'). Наиболее простое назначение у двух последних полей структуры (eDx и eDy) — сдвигать координаты на указанное число логических единиц вдоль осей абсцисс и ординат. Роль полей eM11, eM21, eM21, eM22 гораздо значительнее, для ее разъяснения предлагаю обратиться к табл. 33.3.

***Таблица 33.3.** Назначение полей eM11, eM12, eM21 и eM22 структуры TXForm*

Вид операции	Поля структуры			
	eM11	**eM12**	**eM21**	**eM22**
Тождественное отображение	1	0	0	1
Отражение	Отражение по горизонтальной оси –1	0	0	Отражение по вертикальной оси –1
Поворот на Θ градусов	Косинус угла Θ	Синус угла Θ	Отрицательный синус Θ	Косинус угла Θ
Масштабирование	Коэффициент масштабирования по горизонтали	0	0	Коэффициент масштабирования по вертикали
Поперечная деформация	1	Константа сдвига по вертикали	Константа сдвига по горизонтали	1

Таким образом, если, например, нам требуется изменить масштаб изображения, то следует передать соответствующие коэффициенты масштабирования в поля eM11 и eM22 записи TXForm, при этом не забыв записать в остальные поля нулевое значение. Для поворота изображения нам следует лишь пересчитать значения синуса и косинуса для заданного угла и направить значения в поля eM11, eM21, eM21, eM22. Если по какой-либо причине мы не можем получить требуемого результата за счет единственного аффинного преобразования, допускается создать их комбинацию. Специально для этого реализована функция:

```
function CombineTransform(var p1: TXForm; const p2 : TXForm;
                          const p3 : TXForm):Boolean;
```

Здесь: константы p2 и p3 — первая и вторая структуры TXForm; результат их комбинирования окажется в первом параметре — переменной p1.

Наличие мировых координат существенно упрощает решение ряда графических задач. Например, до того как в Windows появилась поддержка двумерных аффинных преобразований, программисту приходилось изрядно потрудиться над решением даже такой элементарной задачи, как поворот отрезка вокруг своей оси на заданное число градусов. А теперь, благодаря мировым координатам, сложность подобных задач уменьшилась на порядок. Для подтверждения этих слов напишем небольшую программу, рисующую обычные стрелочные часы.

Создайте новый проект Delphi и разместите на главной форме таймер TTimer. С каждым возникновением события OnTimer (листинг 33.6) декодируем системное время (функция DecodeTime), заполняя заранее подготовленные переменные Hour, Min, Sec и MSec (часы, минуты, секунды и миллисекунды соответственно).

Листинг 33.6. Пересчет текущего времени в углы поворота стрелок часов

```
private
    eHour,eMin,eSec:real; //углы поворотов стрелок

//...
procedure TForm1.Timer1Timer(Sender: TObject);
var Hour, Min, Sec, MSec: Word;
begin
  DecodeTime(Time, Hour, Min, Sec, MSec);//декодируем время
  eSec:=Sec*pi/30; //отклонение секундной стрелки
  eMin:=Min*pi/30; //отклонение стрелки минут
  eHour:=Hour*pi/6+Min/60*pi/6; //отклонение часовой стрелки
  Form1.Repaint;   //перерисовка формы
end;
```

На следующем этапе программа конвертирует полученные значения в углы поворота стрелок часов. Например, угол поворота минутной и секундной стрелок (переменные eMin и eSec) вычисляется следующим образом: текущее значение времени умножается на величину π/30. С углом поворота часовой стрелки немножко сложнее. При определении ее местоположения учитываются не только часы (значение Hour), но и минуты (значение Min). Переменная eHour определяет грубое местоположение стрелки. Например, если системное время — 12 часов ровно, то стрелка укажет на 12-часовую отметку циферблата. Переменная Min уточняет степень отклонения стрелки от часовой риски. Если системное время — 12 часов 30 минут, то за счет переменной Min к углу поворота стрелки добавится значение

π/12. Таким образом, часовая стрелка получит дополнительное смещение от вертикального положения, указывающее, что скоро наступит 13 часов дня.

На рис. 33.3 представлен подробный чертеж будущего циферблата нашего проекта. Обратите внимание на следующее:

- начала координат перенесены в центр окна;
- изменено поведение оси *y*, теперь она направлена вверх;
- установлен размер окна 100 × 100 логических единиц, это решение позволит безболезненно масштабировать часы при любом изменении размеров формы.

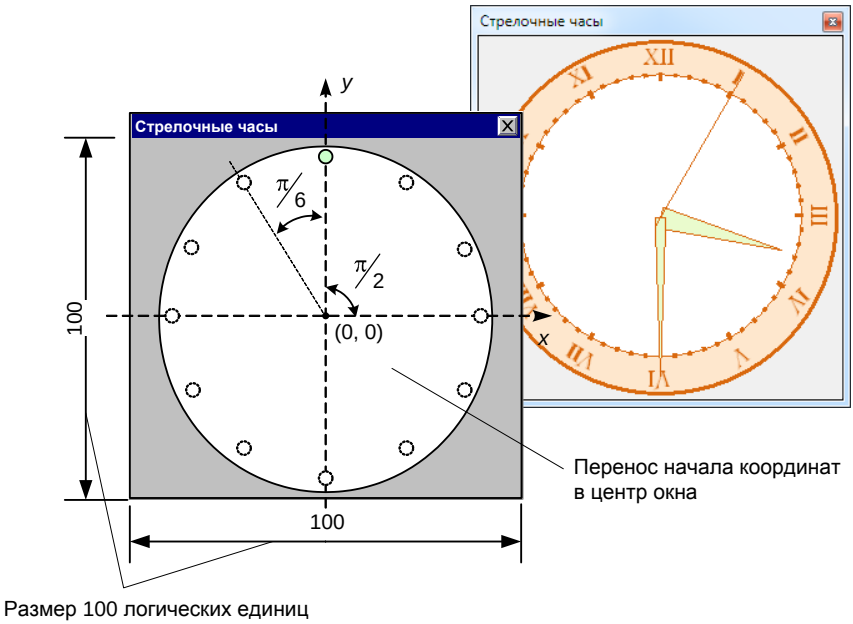


Рис. 33.3. Циферблат стрелочных часов

Листинг 33.7 раскрывает порядок графического вывода часов. Для этих целей мы воспользовались событием `OnPaint()` формы. В первых строках события переводим страничную систему координат в изотропный режим отображения и устанавливаем начало координатных осей в центр окна. В результате ось *y* направляется из центра окна на 12-часовую отметку, а ось *x* — из центра окна на 3-часовую отметку циферблата. Для вывода на экран минутной и часовой стрелок вызывается функция `Polygon()`, в которую передается массив `Ar`, состоящий из трех точек. С секундной стрелкой еще проще — это обычный отрезок, проведенный парой функций `MoveToEx()` и `LineTo()`. По умолчанию все стрелки "смотрят" на 12-часовую отметку часов. Для того чтобы они повернулись на требуемый угол, воспользуемся аффинными преобразованиями, благо требуемые углы стрелок уже посчитаны в листинге 33.6.

Листинг 33.7. Прорисовка часов

```
procedure TForm1.FormPaint(Sender: TObject);
var R:TRect;
    XForm:TXForm;
```

```
    Ar:Array[0..2] of TPoint;
    CS,SN:DOUBLE;
begin
  R:=Form1.ClientRect;

  with Form1.Canvas do
  begin
    SetMapMode(Handle, MM_ISOTROPIC);
    //переносим начала координат в центр окна
    SetViewportOrgEx(Handle, R.Right DIV 2, R.Bottom DIV 2, nil);
    //протяженность в логических единицах 100 x 100
    SetWindowExtEx(Handle, 99, 99, nil);
    SetViewportExtEx(Handle, R.Right, R.Bottom, nil);
    Ellipse(-45,-45,45,45);

    SetGraphicsMode(Handle,GM_ADVANCED);//мировые координаты

  //часовая стрелка
    CS:=COS(eHour); SN:=SIN(eHour);
    XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;
    SetWorldTransform(Handle,XForm);
    Ar[0].X:=-4; Ar[0].Y:=0; Ar[1].X:=4; Ar[1].Y:=0;
    Ar[2].X:=0;  Ar[2].Y:=-30;
    Brush.Color:=clSkyBlue;
    Pen.Style:=psSolid;
    Polygon(Ar);
  //минутная стрелка
    CS:=COS(eMin); SN:=SIN(eMin);
    XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;
    SetWorldTransform(Handle,XForm);
    Ar[0]:=Point(-3,0); Ar[1]:=Point(3,0); Ar[2]:=Point(0,-40);
    Polygon(Ar);
  //секундная стрелка
    CS:=COS(eSec); SN:=SIN(eSec);
    XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;
    SetWorldTransform(Handle,XForm);
    MoveTo(0, 5); LineTo(0,-42);
  end;
end;
```

Если код повторен безошибочно, то в вашем распоряжении окажутся обещанные стрелочные часы.

ГЛАВА 34

Управление печатью

Между нами говоря, огромная часть пользователей полагает, что компьютер — это дорогая печатающая машинка. А так как организация печати — очень непростая задача, то тема использования принтера в проектах Delphi весьма злободневна. Основная сложность в том, что это как раз тот случай, когда программист не сможет обойтись без знаний процедур и функций графической подсистемы Windows. Впрочем, в простейших случаях неоценимый вклад в проект может внести инкапсулирующий возможности принтера класс `TPrinter`.

Прежде чем потребовать от принтера, чтобы он нанес на лист бумаги первые строки текста или пикселы изображения, стоит разобраться, на какие подвиги наш принтер способен. Может быть, он не способен печатать на листах формата A3, а нам нужен именно такой отчет. Может быть, принтер обладает недостаточным разрешением? Может быть, принтер не поддерживает двухстороннюю печать? Одним словом, у современного печатающего устройства очень много технических характеристик, которые следует учитывать при создании отчета.

В *главе 30* уже упоминалось о существовании в составе Windows API универсальной функции `GetDeviceCaps()`, способной проинформировать нас о технических параметрах устройств вывода графической информации. Для этого в `GetDeviceCaps()` передается дескриптор исследуемого контекста графического устройства `DC`. Второй параметр `Index` представляет собой один из почти трех десятков идентификаторов (некоторые из них вы найдете в табл. 34.1). В зависимости от значения идентификатора функция возвратит ту или иную характеристику устройства.

Таблица 34.1. *Возможные значения* `Index` *для исследования печатающего устройства*

Значение `Index`	**Возвращаемая характеристика**
`TECHNOLOGY`	Технология драйвера. Может принимать одно из следующих значений: • DT_PLOTTER — векторный плоттер; • DT_RASDISPLAY — растровый дисплей; • DT_RASPRINTER — растровый принтер
`HORZSIZE`	Физическая ширина области отображения в миллиметрах
`VERTSIZE`	Физическая высота области отображения в миллиметрах
`HORZRES`	Ширина области отображения в пикселах
`VERTRES`	Высота области отображения в пикселах

Таблица 34.1 *(окончание)*

Значение `Index`	Возвращаемая характеристика
`LOGPIXELSX`	Число пикселов в логическом дюйме по ширине области отображения
`LOGPIXELSY`	Число пикселов в логическом дюйме по высоте
`BITSPIXEL`	Число битов цвета для представления одного пиксела
`ASPECTX`	Относительная ширина пиксела устройства по горизонтали
`ASPECTY`	Относительная ширина пиксела устройства по вертикали
`ASPECTXY`	Относительная ширина пиксела устройства по диагонали
`PHYSICALWIDTH`	Ширина физической страницы в пикселах
`PHYSICALHEIGHT`	Высота физической страницы в пикселах
`PHYSICALOFFSETX`	Расстояние от левого края физической страницы до левой границы области печати в единицах устройства
`PHYSICALOFFSETY`	Расстояние от верхнего края физической страницы до верхней границы области печати в единицах устройства
`SCALINGFACTORX`	Коэффициент масштабирования для оси *x* принтера
`SCALINGFACTORY`	Коэффициент масштабирования для оси *y* принтера

Описание принтера в Delphi, класс *TPrinter*

Язык Delphi предлагает один из наиболее простых и одновременно удобных способов работы с печатающим устройством. Для этого к нашему проекту необходимо подключить модуль `Printers`. В результате столь несложной операции программист получает право обращаться к принтеру. Для этого предназначена функция:

```
function Printer: TPrinter;
```

Метод возвратит объект класса `TPrinter`, инкапсулирующий в своем составе значительную часть интерфейса печати GDI.

Выбор принтера

В рамках операционной системы может быть установлено неограниченное число принтеров. Это локальный принтер, сетевые печатающие устройства и различного рода эмуляторы принтеров, например, весьма популярный Adobe Acrobat Distiller, "перегоняющий" любые документы в формат документа Adobe Acrobat, или Microsoft Office Document Image Writer Driver, позволяющий формировать снимки документов в формате растрового изображения TIFF либо MDI (Microsoft Document Imaging). Весь перечень доступных устройств собирается в списке

```
property Printers: TStrings;
```

Индекс текущего принтера окажется в свойстве

```
property PrinterIndex: Integer;
```

Меняя порядковый номер, мы назначаем принтер, на который отправится задание. Для обращения к используемому по умолчанию принтеру присвойте свойству `PrinterIndex` значе-

ние –1. В листинге 34.1 представлен пример инициализации перечня доступных принтеров во время создания главной формы приложения.

Листинг 34.1. Получение списка зарегистрированных в системе принтеров

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  ComboBox1.Items.Assign(Printer.Printers);
  ComboBox1.ItemIndex:=Printer.PrinterIndex;
end;
```

Управление страницей документа

Вспомним еще про один ингредиент, без которого печать невозможна, — это лист бумаги. Средствами визуальной библиотеки компонентов можно легко получить данные о ключевых характеристиках бумаги. В первую очередь это высота и ширина листа:

```
property PageHeight: Integer;
property PageWidth: Integer;
```

Размеры передаются в логических единицах.

Ориентация листа бумаги при печати определяется свойством

```
property Orientation: TPrinterOrientation;
type TPrinterOrientation = (poPortrait, poLandscape);
```

Предлагаются всего два варианта ориентации — портретная или альбомная.

Информация о некоторых возможностях принтера по управлению заданием содержится в свойстве:

```
property Capabilities: TPrinterCapabilities;
```

Описание `TPrinterCapabilities` предложено в табл. 34.2.

***Таблица 34.2.** Возможные значения `TPrinterCapabilities`*

Значение	Описание
`pcCopies`	Пользователю разрешено определять число распечатываемых копий документа
`pcOrientation`	Принтер позволяет изменять ориентацию бумаги
`pcCollation`	При печати нескольких экземпляров одного документа указывает, что экземпляры разбираются по копиям

Формирование и отправка задания на печать

Вся идея работы с объектом принтера заключается в обращении к поверхности бумаги как к холсту, на котором можно рисовать. Именно для этой цели принтер инкапсулирует хорошо знакомый и горячо любимый программистами Delphi фирменный объект VCL — холст:

```
property Canvas: TCanvas;
```

Далее остается проявить умение и сноровку и отправить на печать данные. В простейшем случае достаточно написать менее 10 строк исходного кода (листинг 34.2).

Листинг 34.2. Отправка на печать текстовой строки

```
uses Printers;

procedure TForm1.Button1Click(Sender: TObject);
begin
  with Printer do
  begin
    BeginDoc;     //старт документа
    Canvas.TextOut(10,10,'Тест печати'); //печать строки
    EndDoc;       //завершение документа
  end;
end;
```

Печать инициируется посредством обращения к процедуре

```
procedure BeginDoc;
```

Это команда спулеру печати Windows о начале нового задания. Если все прошло успешно, то не забудьте попрощаться, известив спулер о завершении задания

```
procedure EndDoc;
```

Индикатором того, что принтер приступил к выполнению задания, станет переход в `true` свойства

```
property Printing: Boolean; //только для чтения
```

Если существует необходимость разбиения документа на несколько страниц, вспоминаем о существовании процедуры:

```
procedure NewPage;
```

Обращение к этому методу вынуждает принтер завершить работу с текущим листом бумаги и "стащить" из лотка следующий. Номер распечатываемой в данный момент страницы хранится в свойстве

```
property PageNumber: Integer;
```

Для того чтобы сформированное нами задание на печать более профессионально смотрелось в окне менеджера печати перед обращением к процедуре `BeginDoc()`, определите заголовок заданию. Имя задания передается в свойство

```
property Title: string;
```

Отмена задания

В процессе печати документа спулер периодически проверяет состояние свойства

```
property Aborted: Boolean; //только для чтения
```

И если названное свойство возвратит значение `true`, то это станет командой на отмену текущего задания. Свойство доступно только для чтения, поэтому при необходимости отменить уже отправленное на принтер задание обратитесь к методу

```
procedure Abort;
```

Процедура самостоятельно позаботится о переводе Aborted в состояние true. В результате нераспечатанное задание сбрасывается, и принтер останавливается либо переходит к обслуживанию следующей задачи.

Печать многострочного текста

В девяти случаях из десяти принтер применяется для распечатки текста, в простейшем случае это может быть многострочный неформатированный текст. В подобной ситуации для вывода строк на бумагу стоит воспользоваться услугами холста, а точнее — его метода TextOut(). Напомню, что первые два параметра метода определяют место вывода строки: это, соответственно, горизонтальный и вертикальный отступы от левого верхнего угла страницы. Если с горизонтальным отступом все понятно, при печати неформатированного, выровненного по левому краю текста он каждый раз принимает одно и то же значение, то с вертикальным отступом немного сложнее. Программист обязан предусмотреть порядок назначения вертикального отступа печати для каждой строки. Поэтому в нашем примере перед отправкой задания мы с помощью метода TextHeight() выясняем высоту строки и сохраняем это значение в локальной переменной LineHeight. Позднее это значение будет использоваться как множитель при пересчете места печати очередной строки из свойства Lines компонента Memo1.

```
Y:=Y+LineHeight*2;
```

Выяснив величину отступа, в качестве второго параметра передаем ее в функцию TextOut(), а затем делаем все, как в листинге 34.3.

Листинг 34.3. Печать многострочного текста

```
var LineNum :Word;
    LineHeight, Y, X : integer;
begin
  Printer.Canvas.Font:=Memo1.Font;            //синхронизация шрифтов
  LineHeight:=Printer.Canvas.TextHeight('A'); //выясняем высоту строки
  Printer.Title:='Название задания';          //назначаем имя задания

  X:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETX); {отступ от левого края}
  Y:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETY); {отступ от верхнего
                                                     края}
  Printer.BeginDoc; //старт задания
  for LineNum:=0 to Memo1.Lines.Count-1 do {перебор всех строк Memo1}
  begin
    if Y>Printer.PageHeight-20 then
    begin
      Printer.NewPage; {если страница закончилась, то начинаем очередную}
      Y:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETY);
    end
    else Y:=Y+LineHeight*2;{вертикальный отступ для очередной строки}
    {печать строки}
    Printer.Canvas.TextOut(X,Y,Memo1.Lines.Strings[LineNum]);
  end;
  Printer.EndDoc; //завершаем задание
end;
```

Особенности печати изображений

Подготовка к печати растровых изображений должна начинаться с учета разрешающей способности принтера. Тому есть два серьезных основания.

Во-первых, разрешающая способность даже самого слабого с технической стороны принтера значительно превосходит возможности самого лучшего дисплея. Поэтому занимающая все пространство экрана картинка при выводе на лист бумаги оставит много свободного места. Допустим, что монитор способен выводить 96 точек в одном дюйме по вертикали. Это разрешение вполне подходит для визуального восприятия информации с экрана. Но при попытке вывода на принтер с разрешением 600 точек на дюйм картинки (занимающей на экране площадь 2×2 дюйма) мы получим результат, который едва ли разглядишь под увеличительным стеклом. Почему? Ответ на этот вопрос дадут элементарные расчеты.

Высота изображения = 2 дюйма × 96 точек/дюйм = 192 пиксела.

192 пиксела/ 600 точек/дюйм = 0,32 дюйма, что примерно равно 8 мм.

Таким образом, 2 "огромных" экранных дюйма на принтере с разрешением 600 dpi превратились в скромные 8 миллиметров.

Во-вторых, еще одним поводом для коррекции размеров отправляемого на печать задания может стать поддержка рядом принтеров "нестандартного" разрешения. Довольно часто встречаются модели с неидентичными разрешениями по осям *x* и *y* (360×180, 600×300, 1200×600 и т. п. точек на дюйм). При выводе рисунка на такой принтер без масштабирования мы вполне закономерно получим искаженное (вытянутое) изображение.

Пример печати изображений

Теперь предлагаю опробовать наши силы на реальном задании. Разработаем программу, способную стать базовой для многих приложений, работающих с графикой. Для того чтобы пример стал как можно более универсальным, поставим перед собой следующие задачи:

- перед печатью пользователь должен получить примерный внешний вид отчета, т. е. нам следует создать окно предварительного просмотра;
- пользователь должен обладать возможностью выбирать печатающее устройство и режим печати.

Приступим к реализации задуманного. Добавьте к проекту новую форму, переименуйте ее в `frmReport` и сохраните в папке с остальными формами, соответствующий форме модуль назовите report.pas. Исключите форму из списка автоматически создаваемых. В строку `uses` формы-отчета добавьте ссылку на модуль печати `printers`.

В секции публичных объявлений формы предварительного просмотра объявим переменную `PictureFileName` (в которую мы позднее направим путь и имя к файлу с изображением) и переменную `WICImage`, в которую будет загружен растровый рисунок из файла (листинг 34.4).

Листинг 34.4. Публичные переменные

```
public
    PictureFileName :string;     //путь и имя файла
    WICImage        :TWICImage;  //объект-рисунок
```

Для вызова формы предварительного просмотра разместите на главной форме новую кнопку (или пункт меню) и напишите представленные в листинге 34.5 строки.

Листинг 34.5. Вызов формы предварительного просмотра

```
frmReport:=TfrmReport.Create(Application.MainForm);
frmReport.PictureFileName:=//строка с именем файла с изображением
frmReport.ShowModal;       //вывод окна подготовки отчета на экран
frmReport.Free;            //освобождение ресурса
```

Здесь мы создаем новый экземпляр формы и выводим на экран. На этом подготовительные операции завершены.

Окно предварительного просмотра

Практически ни один принтер не способен печатать на всей поверхности листа. Наличие "мертвых" зон по краям страницы объясняется особенностями конструкции печатающего устройства и в первую очередь зависит от его механической части. В технические подробности кинематики принтеров мы вдаваться не станем, но сделаем вывод, что перед формированием задания на печать целесообразно выяснить геометрические размеры страницы и размеры недоступных для печати областей листа бумаги, чтобы часть распечатываемой информации не угодила в запретные зоны. Для решения этой задачи в качестве помощника привлечем функцию `GetDeviceCaps()`. Метод не только укажет ширину (`PHISICALWIDTH`) и высоту (`PHISICALHEIGHT`) страницы, но и уточнит физические возможности печатающего устройства по выводу данных. В первую очередь это отступы от левого (`PHISICALOFFSETX`) и верхнего (`PHISICALOFFSETY`) краев листа. Размеры правой и нижней "мертвых" зон легко вычисляются, если мы знаем физическую ширину (`HORZRES`) и высоту (`VERTRES`) доступной для печати области (рис. 34.1).

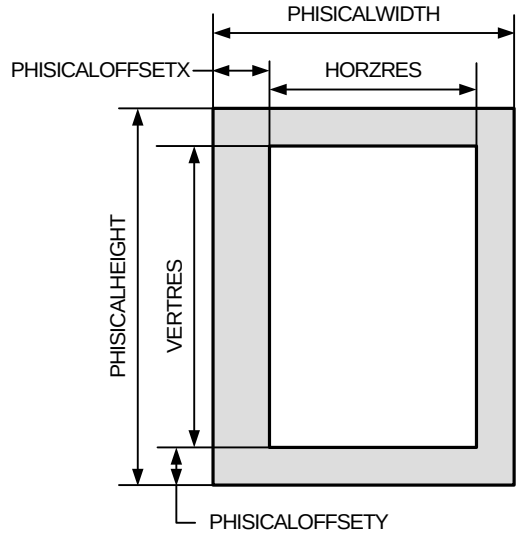


Рис. 34.1. Выяснение размеров листа бумаги с помощью функции `GetDeviceCaps()`

Для хранения всей служебной информации в секции частных объявлений формы-отчета объявим восемь переменных (листинг 34.6).

Листинг 34.6. Частные переменные

```
private
    {разрешение по горизонтали и вертикали в лог. единицах}
    prLOGPIXELSX, prLOGPIXELSY,
    prPHYSICALWIDTH, prPHYSICALHEIGHT, {размеры страницы в лог. единицах}
    prPHYSICALOFFSETX, prPHYSICALOFFSETY,   {поля отступа слева и сверху}
    prHORZRES, prVERTRES:integer;      {размеры рабочей области страницы}
```

Такой внушительный перечень переменных нужен для обеспечения хранения данных о физических возможностях принтера. Для заполнения большинства из представленных выше переменных создаем отдельную процедуру (листинг 34.7).

Листинг 34.7. Получение характеристик принтера

```
procedure TfrmReport.GetPrinterSettings;
begin
{узнаем разрешение принтера — число точек на дюйм}
  prLOGPIXELSX:=GetDeviceCaps(Printer.Handle, LOGPIXELSX);
  prLOGPIXELSY:=GetDeviceCaps(Printer.Handle, LOGPIXELSY);
{размеры выбранного листа бумаги в логических единицах принтера}
  prPHYSICALWIDTH:=GetDeviceCaps(Printer.Handle, PHYSICALWIDTH);
  prPHYSICALHEIGHT:=GetDeviceCaps(Printer.Handle, PHYSICALHEIGHT);
{отступы слева и сверху}
  prPHYSICALOFFSETX:=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETX);
  prPHYSICALOFFSETY:=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETY);
{ширина и высота рабочей области}
  prHORZRES:=GetDeviceCaps(Printer.Handle, HORZRES);
  prVERTRES:=GetDeviceCaps(Printer.Handle, VERTRES);
end;
```

Теперь займемся внешним видом формы предварительного просмотра (рис. 34.2). Нам потребуются следующие компоненты:

- панель группировки `GroupBox1`. Разместите панель в правой части формы, выбрав режим выравнивания `Align=alRight`;
- комбинированный список `ComboBox1` для хранения перечня доступных принтеров. Расположите компонент на панели группировки `GroupBox1`;
- диалог настройки принтера `PrinterSetupDialog1`;
- кнопка `btnPrinterSetup`, предназначенная для вызова диалога настройки принтера;
- в нижней части панели группировки расположите еще две кнопки. Кнопка `btnPrint` возьмет ответственность за отправку задания на печать. Кнопка `btnCancel` закроет форму;

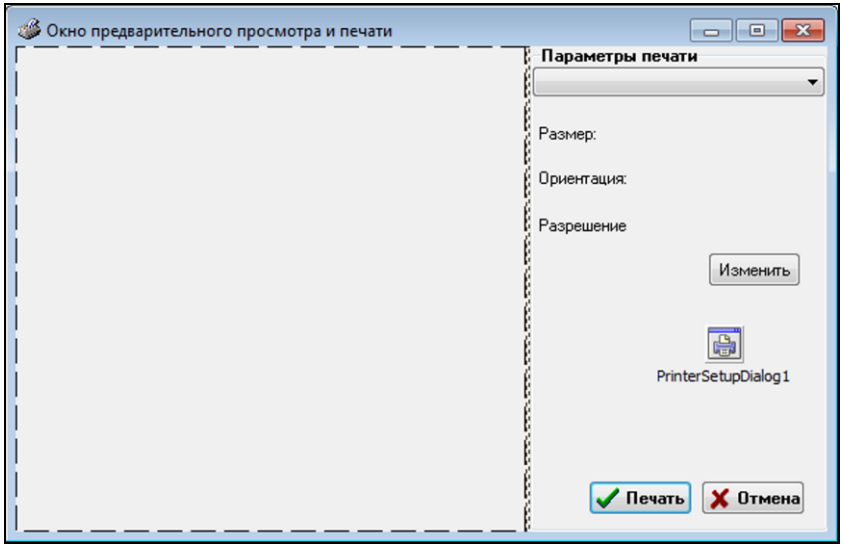


Рис. 34.2. Внешний вид формы предварительного просмотра задания на печать

- в левой части формы расположим область для графического вывода `PaintBox1:TPainBox`. На ее поверхности мы нарисуем окно предварительного просмотра. Установите свойство компонента `Align` в состояние `alClient`.

В момент вывода формы на экран генерируется событие `OnShow()`. Это подходящий момент для сбора сведений об установленных в системе печатающих устройствах и их параметрах (листинг 34.8).

Листинг 34.8. Вывод формы на экран

```
procedure TfrmReport.FormShow(Sender: TObject);
var ext:string;
begin
  ComboBox1.Items:=Printer.Printers;        //собираем принтеры
  ComboBox1.ItemIndex:=Printer.PrinterIndex;//устанавливаем принтер
  GetPrinterSettings;               //узнаем характеристики принтера
  WICImage:=TWICImage.Create;       //готовим контейнер для рисунка
  WICImage.LoadFromFile(PictureFileName);//загружаем растровый рисунок WIC
end;
```

Перечень всех доступных принтеров передается в свойство `Items` комбинированного списка, затем мы "вычисляем" текущий принтер и получаем все необходимые для нашей дальнейшей работы характеристики печатающего устройства. Кроме того, в листинге 34.8 создается экземпляр класса `TWICImage` (напомню, что это универсальный контейнер для растровых рисунков), и в него загружается подлежащее печати изображение.

При смене пользователем текущего принтера (смена производится в комбинированном списке `ComboBox1`) следует вновь собрать все его характеристики. Поэтому в момент закрытия комбинированного списка вновь вызовем процедуру `GetPrinterSettings()` так, как это предложено в листинге 34.9.

Листинг 34.9. Изменение текущего принтера

```
procedure TfrmReport.ComboBox1CloseUp(Sender: TObject);
begin
  Printer.PrinterIndex:=ComboBox1.ItemIndex;//выбор принтера
  GetPrinterSettings; //получение сведений о принтере
  PaintBox1.Repaint;  //перерисовка окна предварительного просмотра
end;
```

Кроме того, инициируем перерисовку области вывода `PaintBox1`. Эту операцию мы вызываем, немного забегая вперед, ведь пока в обработчике события `OnPaint()` компонента `PaintBox1` не написано ни одной строки. Но это один из самых сложных этапов программирования, и нам необходимо к нему как следует подготовиться. Впрочем, мы уже близко подобрались к цели, нам осталось описать событие щелчка по кнопке `btnPrinterSetup` (листинг 34.10).

Листинг 34.10. Вызов диалога настройки принтера

```
procedure TfrmReport.btnPrinterSetupClick(Sender: TObject);
begin
  PrinterSetupDialog1.Execute(); //старт диалога
  ComboBox1.ItemIndex:=Printer.PrinterIndex;
  ComboBox1CloseUp(Sender); //вызываем событие OnCloseUp списка
end;
```

Щелчок по кнопке пробуждает к жизни диалог изменения настроек принтера `PrinterSetupDialog1`. Обращение к методу `Execute()` компонента выводит на экран стандартное окно **Настройка печати**. Благодаря диалоговому окну мы получаем возможность не только изменить текущий принтер, но и выбрать размеры листа бумаги, его ориентацию и качество печати (в первую очередь определяемое разрешением принтера).

Теперь наступает самый интересный и вместе с тем наиболее сложный этап работы. Нам предстоит написать исходный код для окна предварительного просмотра, подлежащего печати текста. На исполнение роли этого окна нами уже был выдвинут компонент `PaintBox1`. Такой чести компонент удостоился благодаря наличию у него своего собственного холста для рисования — свойства `Canvas`. На нашу долю остается немного "поколдовать" с кодом прорисовки в рамках события `OnPaint()`(листинг 34.11).

Листинг 34.11. Вывод окна предварительного просмотра

```
procedure TfrmReport.PaintBox1Paint(Sender: TObject);
var PaperRect,WorkRect:TRect;
begin
  GetPrinterSettings;
  with PaintBox1.Canvas do
  begin
    Brush.Color:=clCream;
    FillRect(PaintBox1.ClientRect);
    SetMapMode(Handle, MM_ISOTROPIC);// изотропный режим отображения
```

```
    {определяем протяженность логического окна,
     равную сторонам листа плюс 100 единиц}
    SetWindowExtEx  (Handle, prPHYSICALWIDTH+100,
                     prPHYSICALHEIGHT+100, nil);
    SetViewportExtEx(Handle, PaintBox1.ClientWidth,
                     PaintBox1.ClientHeight, nil);
    {перенос начала координат в левый верхний угол листа бумаги}
    SetWindowOrgEx(Handle, -50, -50, nil);
    //прямоугольная область страницы
    PaperRect :=RECT(0, 0, prPHYSICALWIDTH,   prPHYSICALHEIGHT);
    //прямоугольная рабочая область
    WorkRect.Top   :=PaperRect.Top +prPHYSICALOFFSETY;
    WorkRect.Left  :=PaperRect.Left+prPHYSICALOFFSETX;
    WorkRect.Bottom:=WorkRect.Top  +prVERTRES;
    WorkRect.Right :=WorkRect.Left +prHORZRES;
    Brush.Color:=clWhite; //рисуем лист
    FillRect(PaperRect);
    //ИЗОБРАЖЕНИЕ ДЛЯ ПЕЧАТИ
    if WICImage<>nil then Draw(WorkRect.Left, WorkRect.Top, WICImage);
    Pen.Color:=clRed;   //рисуем границы рабочей области
    Pen.Style:=psDot;
    Brush.Style:=bsClear;
    Rectangle(WorkRect);
  end;
end;
```

При решении подобных задач мы не сможем обойтись без поддержки функций Windows API, отвечающих за настройку режима отображения. Сердце процедуры — код, переводящий холст в пользовательский (изотропный) режим отображения. Для этого вызывается функция `SetMapMode()` с ключом `MM_ISOTROPIC`. С этого момента мы приобретаем эксклюзивное право назначать собственную единицу измерения. Несколько позднее, благо-

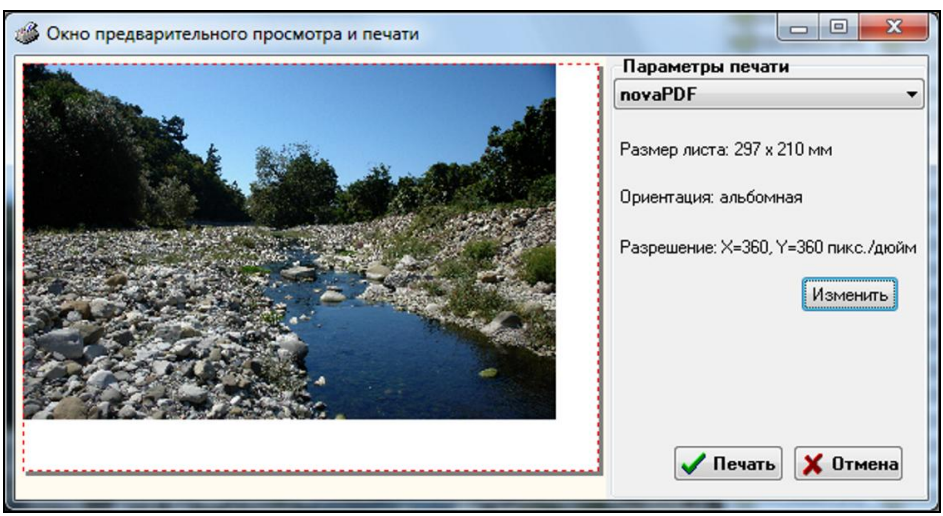


Рис. 34.3. Внешний вид окна предварительного просмотра

даря изотропному режиму отображения и функциям GDI `SetWindowExtEx()` и `SetViewportExtEx()`, изменяем значения протяженности холста. Мы без всякого зазрения совести обманули окно предварительного просмотра, с этого момента приложение начинает полагать, что крошечная поверхность компонента `PaintBox` вмещает в себя целый лист бумаги. Для еще большего удобства с помощью `SetWindowOrgEx()` мы переносим точку отсчета координат в левый верхний угол листа бумаги. Все остальное — дело техники: определяем координаты прямоугольной области, имитирующей лист бумаги (структура `PaperRect`), и рабочей области (структура `WorkRect`). Имитацию печати рисунка осуществляем самым простейшим способом — выводим рисунок в левом верхнем углу рабочей области с помощью метода `Draw()`.

На рис. 34.3 представлен вид разработанного нами окна предварительного просмотра. Дело осталось за малым — завершить проект. Для этого надо написать процедуру печати отчета на реальном принтере.

Отправка задания на печать

Благодаря тому, что работа со свойством `Canvas` компонента `PaintBox1` почти ничем не отличается от работы с аналогичным свойством принтера `TPrinter`, нам остается просто перенести код из процедуры предварительного просмотра в процедуру печати (листинг 34.12).

Листинг 34.12. Код печати

```
procedure TfrmReport.btnPrintClick(Sender: TObject);
begin
  Printer.Title:=ExtractFileName(PictureFileName);
  Printer.BeginDoc; {начало документа}
  if WICImage<>nil then  //ИЗОБРАЖЕНИЕ
      Printer.Canvas.Draw(prPHYSICALOFFSETX,prPHYSICALOFFSETY,WICImage);
  Printer.EndDoc; {конец документа}
end;
```

По сравнению с кодом предварительного просмотра процедура печати существенно упростилась. Причина в том, что здесь нам не надо идти на "обман" области вывода `PaintBox1`, внушая компоненту, что он обладает тем же разрешением, что и принтер.

ЧАСТЬ III

VCL и Windows API

Глава 35. Реестр Windows
Глава 36. Управление процессами
Глава 37. Многопоточные приложения
Глава 38. Взаимодействие процессов
Глава 39. Сетевое взаимодействие
Глава 40. Сервисы Windows
Глава 41. Динамически подключаемые библиотеки
Глава 42. Многокомпонентная модель COM
Глава 43. Автоматизация
Глава 44. Интерфейс *IShellFolder*
Глава 45. DataSnap
Глава 46. LiveBindings

ГЛАВА 35

Реестр Windows

Каждое серьезное приложение, в том числе и сама операционная система, нуждается в централизованном хранилище данных, способном содержать сведения о настройке аппаратных средств компьютера, программных продуктов, предпочтениях пользователей, работающих на компьютере, и другие данные. В роли такого хранилища в Windows выступает системный реестр.

Системный реестр — это иерархическая база данных, которая физически располагается в файлах USER.DAT и SYSTEM.DAT. Для просмотра и редактирования реестра в состав Windows включена утилита Редактор реестра (рис. 35.1). Для ее запуска необходимо щелкнуть по кнопке **Пуск** и в строке **Выполнить** набрать имя файла regedit.

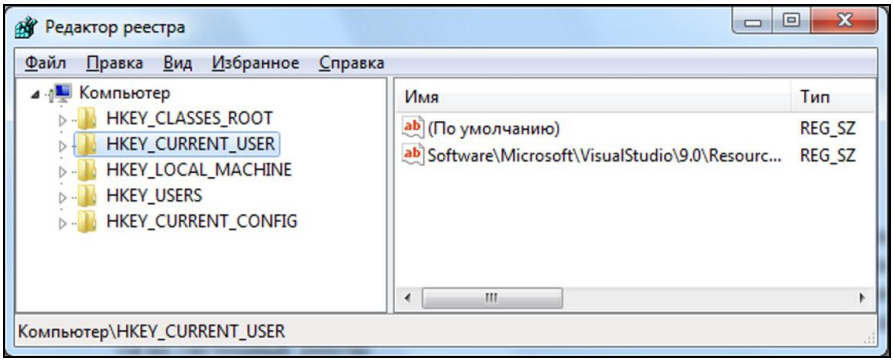


Рис. 35.1. Редактор реестра

Запустив Редактор реестра в Windows Vista/7, вы увидите, что реестр включает пять корневых разделов, имя каждого из которых начинается с префикса `HKEY_`, а если вы откроете утилиту в Windows Server 2003/2008, то получите доступ к 6 разделам. Каждый из разделов имеет свое назначение:

- раздел `HKEY_CLASSES_ROOT` содержит все файловые ассоциации (расширения файлов) и определения COM-объектов;
- в разделе `HKEY_LOCAL_MACHINE` представлена информация о типах аппаратных средств, программного обеспечения, других настроек;
- раздел `HKEY_CURRENT_CONFIG` содержит данные о текущей конфигурации компьютера (является подразделом `HKEY_LOCAL_MACHINE`);

- раздел HKEY_USERS хранит индивидуальные настройки всех пользователей компьютера, каждый пользователь представлен ключом SID;
- раздел HKEY_CURRENT_USER соответствует данным о пользователе, в текущий момент работающем на компьютере (является подразделом HKEY_USERS);
- раздел HKEY_PERFORMANCE_DATA имеется только у серверов Windows и содержит сведения о производительности.

Каждый раздел представляет собой контейнер, содержащий другие разделы, называемые *подразделами*, и их параметры. Параметр описывается именем, типом данных и значением.

В помощь программисту Delphi для работы с системным реестром разработаны три класса (рис. 35.2):

- класс TRegistryIniFile, предоставляющий перечень наиболее востребованных и, что очень важно, удобных методов для работы с реестром;
- полнофункциональный класс TRegistry, оказывающий весь спектр услуг по управлению системным реестром Windows;
- класс TRegIniFile, являющийся надстройкой над классом TRegistry и объединяющий в себе функциональную полноту родительского класса и простоту работы.

В этой главе мы подробно рассмотрим классы TRegistryIniFile и TRegistry. Полученных знаний окажется вполне достаточно, чтобы самостоятельно освоить работу с TRegIniFile.

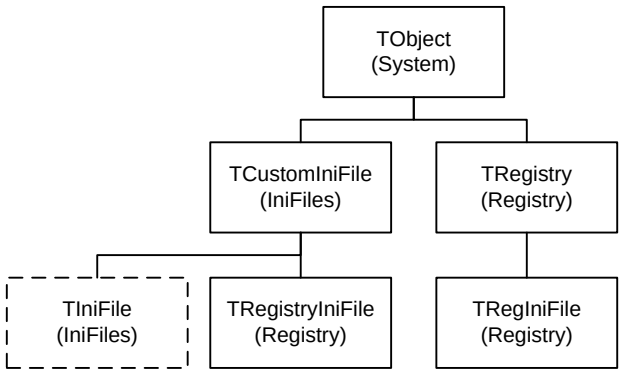


Рис. 35.2. Иерархия классов, работающих с реестром Windows

Класс *TRegistryIniFile*

До выхода в свет Windows 95 роль системного реестра отводилась многочисленным файлам инициализации (INI-файлам). Для работы с файлами инициализации еще со времен первой версии Delphi сохранился весьма удобный в эксплуатации класс TIniFile (см. рис. 35.2), который активно задействовался в программах для старого 16-разрядного парка Windows 3.*x*. Сегодня такой способ хранения данных морально устарел и за очень редким исключением не используется. Чтобы упростить программистам Delphi переход от работы с файлами инициализации к реестру, разработчики VCL создали очень простой и удобный класс TRegistryIniFile.

Класс TRegistryIniFile объявлен в модуле Registry, жизненный путь экземпляра класса традиционно начинается с вызова его конструктора:

```
constructor Create(const FileName: string); overload;
constructor Create(const FileName: string; AAccess: LongWord); overload;
```

Вновь созданный экземпляр получает доступ к разделу реестра текущего пользователя HKEY_CURRENT_USER, а имя подраздела уточняется в первом параметре конструктора. При необходимости, во время создания объекта можно определить уровень доступа к реестру, для этих целей предназначен параметр AAccess (табл. 35.1). Если же этого не сделать, то будет предоставлен абсолютный доступ (флаг KEY_ALL_ACCESS).

> ***Внимание!***
>
> Класс TRegistryIniFile предназначен для обслуживания раздела HKEY_CURRENT_USER. Если логика работы вашего приложения предполагает получение доступа к другим корневым разделам, следует воспользоваться услугами класса TRegistry или TRegIniFile.

Таблица 35.1. *Флаги доступа к реестру*

Флаг	Описание
KEY_ALL_ACCESS	Комбинация флагов KEY_READ, KEY_WRITE и KEY_CREATE_LINK
KEY_READ	Комбинация флагов KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS и KEY_NOTIFY
KEY_WRITE	Комбинация флагов KEY_SET_VALUE и KEY_CREATE_SUB_KEY
KEY_QUERY_VALUE	Разрешает получать данные из параметров подраздела
KEY_ENUMERATE_SUB_KEYS	Разрешает просматривать подразделы реестра
KEY_NOTIFY	Разрешает получать уведомление об изменении ключа
KEY_SET_VALUE	Разрешает устанавливать значения
KEY_CREATE_SUB_KEY	Разрешает создавать подразделы
KEY_CREATE_LINK	Допускает создание связей с другими подразделами
KEY_EXECUTE	Предоставляет право на чтение данных

Появившийся на свет экземпляр класса TRegistryIniFile прикрепляется к заданному в конструкторе подразделу и работает с ним вплоть до вызова деструктора. Единственное свойство реестра

```
property RegIniFile: TRegIniFile;
```

может напомнить забывчивому разработчику имя выбранного подраздела.

Чтение из реестра

В классе TRegistryIniFile реализован внушительный список методов, осуществляющих операции чтения значений параметров. Большинство названий методов чтения начинается со слова Read (табл. 35.2) и включает три параметра: параметр Section идентифицирует имя подраздела; Ident — имя параметра в подразделе; аргумент Default содержит значение по умолчанию, которое возвратит функция, если не обнаружит заданный подраздел или параметр.

Таблица 35.2. *Основные методы чтения параметров системного реестра*

Метод	Описание
`function ReadBool(const Section, Ident: string; Default: Boolean): Boolean ;`	Для логических типов
`function ReadFloat(const Section, Ident: string; Default: Double): Double;`	Вещественные числа
`function ReadInteger(const Section, Ident: string; Default: Longint): Longint;`	Целые числа
`function ReadString(const Section, Ident, Default: String): String;`	Текст
`function ReadDate(const Section, Ident: string; Default: TDateTime): TDateTime;`	Дата
`function ReadTime(const Section, Ident: string; Default: TDateTime): TDateTime;`	Время
`function ReadDateTime(const Section, Ident: string; Default: TDateTime): TDateTime;`	Комбинация даты и времени
`function ReadBinaryStream(const Section, Name: string; Value: TStream): Integer;`	Чтение бинарного потока в `Value`

Для проверки факта существования подраздела с именем `Section` используйте функцию

```
function SectionExists(const Section: string): Boolean;
```

Если подраздел существует, то метод вернет `true`.

Для проверки существования параметра `Ident` внутри подраздела `Section` используйте

```
function ValueExists(const Section, Ident: string): Boolean;
```

Еще один метод осуществляет посекционное чтение. Результат записывается в список строк

```
procedure ReadSectionValues(const Section: string; Strings: TStrings);
```

Процедура

```
procedure ReadSection(const Section: string; Strings: TStrings);
```

прочитает только имена подразделов внутри текущего раздела.

Используя процедуру

```
procedure ReadSections(Strings: TStrings);
```

вы получите возможность экспортировать содержимое выбранного раздела реестра в набор строк.

Запись в реестр

Имена методов, осуществляющих запись значений в файл инициализации, начинаются со слова `Write`. Процедура также включает три аргумента. Первые два определяют имя подраздела и параметра: `Section`, `Ident`. Последний аргумент должен содержать значение соответствующего типа. Все процедуры записи приведены в табл. 35.3.

Таблица 35.3. *Основные методы записи значений параметров системного реестра*

Метод	Описание
`procedure WriteBool(const Section, Ident: string; Value: Boolean);`	Для логических типов
`procedure WriteFloat(const Section, Ident: string; Value: Double);`	Вещественные числа
`procedure WriteInteger(const Section, Ident: string; Value: Longint);`	Целые числа
`procedure WriteString(const Section, Ident, Value: string);`	Текст
`procedure WriteDate(const Section, Ident: string; Value: TDateTime);`	Дата
`procedure WriteTime(const Section, Ident: string; Value: TDateTime);`	Время
`procedure WriteDateTime(const Section, Ident: string; Value: TDateTime);`	Комбинация даты и времени
`procedure WriteBinaryStream(const Section, Name: string; Value: TStream);`	Запись бинарного потока из `Value`

ЗАМЕЧАНИЕ

Если указанные в методах `Write...` раздел или имя параметра (`Section`, `Ident`) отсутствуют в реестре, то они будут созданы.

Удаление подраздела

Для удаления ненужного подраздела из реестра вам понадобится метод

```
procedure EraseSection(const Section: string);
```

В качестве параметра передается имя ненужного подраздела.

Для удаления одного-единственного ключа используйте метод

```
procedure DeleteKey(const Section, Ident: string);
```

В этом случае, кроме имени секции — `Section`, потребуется имя ключа — `Ident`.

Пример

Порядок работы с классом `TRegistryIniFile` раскрывает листинг 35.1. В нем предлагается исходный код программы, сохраняющий местоположение и размеры формы в реестре. Во время создания формы мы производим чтение из реестра значений, определяющих положение левого верхнего угла, высоту и ширину формы. При уничтожении формы выполняем обратное действие — сохраняем в реестре сведения о форме.

Листинг 35.1. Пример работы с классом `TRegistryIniFile`

```
...
const key='software\example';
      subkey = 'position';
```

```
implementation
 Uses Registry;
{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
var Reg : TRegistryIniFile;
begin
  Reg:=TRegistryIniFile.Create(key);
  Form1.Top:=Reg.ReadInteger(subkey, 'TOP',0);
  Form1.Left:=Reg.ReadInteger(subkey, 'LEFT',0);
  Form1.Width:=Reg.ReadInteger(subkey, 'WIDTH',300);
  Form1.Height:=Reg.ReadInteger(subkey, 'HEIGHT',100);
  Reg.Destroy;
end;

procedure TForm1.FormDestroy(Sender: TObject);
var Reg:TRegistryIniFile;
begin
  Reg:=TRegistryIniFile.Create(key);
  Reg.WriteInteger(subkey, 'TOP',Form1.Top);
  Reg.WriteInteger(subkey, 'LEFT',Form1.Left);
  Reg.WriteInteger(subkey, 'WIDTH',Form1.Width);
  Reg.WriteInteger(subkey, 'HEIGHT',Form1.Height);
  Reg.Destroy;
end;
```

Проверив работоспособность предложенного в листинге 35.1 кода, вновь воспользуйтесь Редактором реестра. Откройте в реестре подраздел `HKEY_CURRENT_USER\Software\example` и просмотрите созданные из кода программы параметры — они должны содержать позицию и размеры формы на момент закрытия (рис. 35.3).

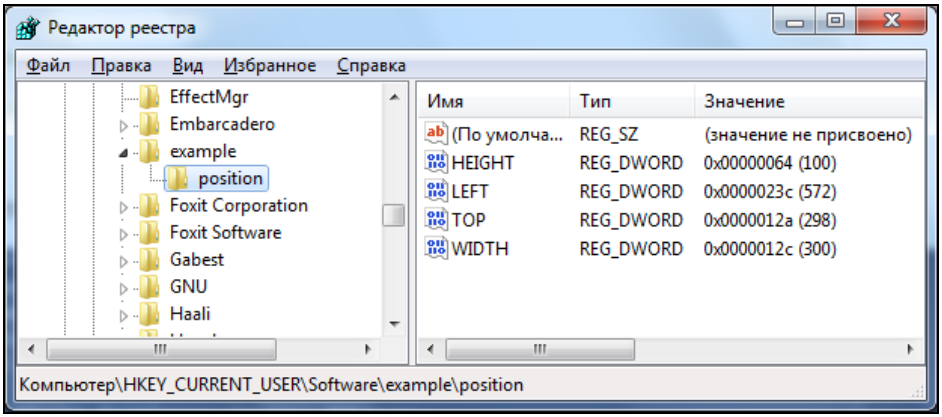


Рис. 35.3. Редактор реестра
с активным подразделом `HKEY_CURRENT_USER\Software\example`

Класс *TRegistry*

Зачастую, при работе с системным реестром Windows, набора возможностей класса `TRegistryIniFile` бывает недостаточно. В этом случае нам способен помочь описанный в модуле `Registry` класс `TRegistry`. В этом классе инкапсулированы практически все нацеленные на работу с системным реестром функции Windows API.

Создание и уничтожение экземпляра реестра

Предусмотрены два варианта создания экземпляра класса:

```
constructor Create; overload;
constructor Create(AAccess : LongWord); overload;
```

В момент создания экземпляр класса подключается к разделу `HKEY_CURRENT_USER` системного реестра. Разница между методами конструирования класса в том, каким образом будет инициализировано свойство

```
property Access: LongWord;
```

устанавливающее уровень доступа к реестру из программы. При обращении к первому варианту `Create()` свойство будет установлено в состояние `KEY_ALL_ACCESS`, что соответствует максимальному уровню доступа. При работе со вторым вариантом метода программист может подобрать подходящий для решаемой задачи флаг из табл. 35.1.

Объект-реестр прекращает свое существование после вызова деструктора `Destroy()`.

Работа с удаленным реестром

Класс `TRegistry` поддерживает возможность подключения к реестру включенного в сеть удаленного компьютера. Сетевое имя компьютера передается в единственный параметр метода

```
function RegistryConnect(const UNCName: String): Boolean;
```

Имя должно начинаться с двух обратных слэшей, например `\\Computer1`. Если соединение закончилось успешно — метод возвратит значение `true`.

Доступ к разделам реестра

В момент создания экземпляра реестра `TRegistry` он инициализируется в корневом разделе `HKEY_CURRENT_USER`, в чем вы сможете убедиться, проверив состояние свойства

```
property RootKey: HKEY;
```

Какой именно подраздел является текущим, мы узнаем благодаря свойству

```
property CurrentKey: HKEY;//только для чтения
```

Для выяснения полного пути к текущему подразделу воспользуйтесь свойством

```
property CurrentPath: string;
```

Так как реестр построен по иерархической схеме, то текущий путь начнется с названия раздела реестра и включит в себя всю последовательность подразделов.

Для проверки факта существования подраздела обращаемся к методу

```
function KeyExists(const Key: string): Boolean;
```

Если подраздел `Key` действительно имеется в реестре — метод возвратит значение `true`.

Если у текущего подраздела имеются дочерние подразделы, то функция

```
function HasSubKeys: Boolean;
```

возвратит значение `true`.

Для смены текущего подраздела предназначен метод

```
function OpenKey(const Key: string; CanCreate: Boolean): Boolean;
```

В строковом параметре `Key` определяется путь (исключая название головного раздела). Параметр `CanCreate` определяет поведение функции в случае, если открываемый ключ физически отсутствует в разделе. В последнем случае (если в `CanCreate` передано значение `true`) несуществующий ключ будет создан. При успешном выполнении метод возвратит значение `true`.

Если мы не планируем осуществлять операции записи в открываемый подраздел, то вместо `OpenKey()` целесообразнее воспользоваться услугами метода

```
function OpenKeyReadOnly(const Key: string): Boolean;
```

Для создания нового ключа в реестре специализируется метод

```
function CreateKey(const Key: string): Boolean;
```

Для удаления ключа и всех связанных с ним данных вам понадобится функция

```
function DeleteKey(const Key: string): Boolean;
```

Для завершения работы с текущим подразделом вызываем метод

```
procedure CloseKey;
```

> ***Внимание!***
> Работа с подразделом реестра обязательно должна завершаться вызовом метода `CloseKey()`.

Некоторые особенности записи данных в реестр зависят от состояния свойства

```
property LazyWrite: Boolean; //по умолчанию true
```

Если свойство установлено в `true`, то экземпляр `TRegistry` работает в режиме отложенной записи. В этом случае все изменения вносятся в реестр только после вызова метода `CloseKey()`. Переведя `LazyWrite` в состояние `false`, мы заставим экземпляр `TRegistry` заносить данные в реестр немедленно.

Для перемещения всех данных из ключа в новое место предназначен метод

```
procedure MoveKey(const OldName, NewName: String; Delete: Boolean);
```

Здесь: `OldName` — старое имя ключа; `NewName` — имя ключа, в котором будет создан дубликат данных из `OldName`. Если в параметр `Delete` передано значение `true`, то после успешного копирования данных старый ключ удаляется.

Чтение и запись значений в параметры

Для получения всего списка параметров текущего подраздела используем метод

```
procedure GetValueNames(Strings: TStrings);
```

Перечень названий будет передан в набор строк `Strings`.

Для чтения значения из параметра в классе `TRegistry` инкапсулированы методы, описанные в табл. 35.4. Все методы объединяет общий аргумент `Name`, определяющий имя параметра реестра, к которому мы получаем доступ.

Таблица 35.4. *Чтение значений параметра*

Метод	Описание
`function ReadBool(const Name: string): Boolean;`	Чтение логического значения
`function ReadString(const Name: string): String;`	Чтение строки
`function ReadInteger(const Name: string): Integer;`	Чтение целого числа
`function ReadFloat(const Name: string): Double;`	Чтение действительного значения
`function ReadCurrency(const Name: string): Currency;`	Чтение денежных значений
`function ReadDate(const Name: string): TDateTime;`	Чтение даты
`function ReadTime(const Name: String): TDateTime;`	Чтение времени
`function ReadDateTime(const Name: string): TDateTime;`	Чтение даты/времени
`function ReadBinaryData(const Name: string; var Buffer; BufSize: Integer): Integer;`	Чтение набора двоичных данных. Данные помещаются в буфер `Buffer`, размер буфера передается в параметр `BufSize`

Перед началом чтения из параметра реестра имеет смысл убедиться в том, что ему сопоставлено какое-то значение. Для этого предназначен метод

```
function ValueExists(const Name: String): Boolean;
```

Если значение обнаружено, то метод возвращает `true`.

Записью значения в параметр ведает группа методов `Write...` По аналогии с методами чтения в параметр `Name` требуется передать название параметра, значение помещается в аргумент `Value` (табл. 35.5).

Таблица 35.5. *Запись значений параметра*

Метод	Описание
`procedure WriteBool(const Name: string; Value: Boolean);`	Запись логического значения
`procedure WriteString(const Name, Value: string);`	Запись текстовой строки

***Таблица 35.5** (окончание)*

Метод	Описание
`procedure WriteInteger(const Name: string; Value: Integer);`	Запись целого числа
`procedure WriteFloat(const Name: string; Value: Double);`	Запись действительного значения
`procedure WriteCurrency(const Name: string; Value: Currency);`	Запись денежного значения
`procedure WriteDate(const Name: string; Value: TDateTime);`	Запись даты
`procedure WriteTime(const Name: string; Value: TDateTime);`	Запись времени
`procedure WriteDateTime(const Name: string; Value: TDateTime);`	Запись даты/времени
`procedure WriteBinaryData(const Name: string; var Buffer; BufSize: Integer);`	Запись двоичной информации из буфера `Buffer` размером `BufSize`

Для очистки параметра используйте метод

```
function DeleteValue(const Name: string): Boolean;
```

Подчеркну, что метод не удаляет параметр, а лишь изымает из реестра связанное с ним значение.

Параметр можно переименовать. Для этого предназначена процедура

```
procedure RenameValue(const OldName, NewName: String);
```

Здесь: `OldName` — старое имя параметра; `NewName` — новое имя параметра.

Получение информации о разделе

Для получения списка всех подразделов вызовем процедуру

```
procedure GetKeyNames(Strings: TStrings);
```

Перечень направляется в набор строк `Strings`.

Подробное описание текущего раздела предоставит метод

```
function GetKeyInfo(var Value: TRegKeyInfo): Boolean;
```

Все данные будут представлены в виде структуры `TRegKeyInfo`, приведенной в листинге 35.2.

Листинг 35.2. Объявление записи `TRegKeyInfo`

```
type TRegKeyInfo = record
  NumSubKeys: Integer;   {число подразделов}
  MaxSubKeyLen: Integer; {количество символов в самом длинном названии
                          подраздела}
  NumValues: Integer;    {количество параметров в разделе}
  MaxValueLen: Integer;  {количество символов в самом длинном
                          названии параметра}
```

```
  MaxDataLen: Integer;   {число символов в самом длинном значении}
  FileTime: TFileTime;   {время последнего изменения раздела}
end;
```

Получение сведений о параметре

Список всех параметров ключа построит процедура

```
procedure GetValueNames(Strings: TStrings);
```

Тип данных параметра с именем `ValueName` возвратит функция

```
function GetDataType(const ValueName: String): TRegDataType;
type TRegDataType = (rdUnknown, rdString, rdExpandString,
                     rdInteger, rdBinary);
```

О размере значения параметра в байтах расскажет метод

```
function GetDataSize(const ValueName: string): Integer;
```

Функционал методов `GetDataType()` и `GetDataSize()` объединяет

```
function GetDataInfo(const ValueName: string;
                     var Value: TRegDataInfo): Boolean;
```

Результат анализа параметра `ValueName` метод возвратит в переменную `Value`.

```
type TRegDataInfo = record
    RegData: TRegDataType;//тип данных
    DataSize: Integer;    //размер данных
end;
```

Экспорт и импорт разделов реестра

В классе `TRegistry` предусмотрены методы, позволяющие архивировать часть данных реестра в файл и восстанавливать данные из архивных копий. Для экспорта данных, содержащихся в разделе с именем `Key`, во внешний файл предназначена функция

```
function SaveKey(const Key, FileName: string): Boolean;
```

Имя архивного файла `FileName` позже пригодится в методах `LoadKey()`, `ReplaceKey()` и `RestoreKey()`. Если экспорт завершился успешно, то функция возвратит значение `true`.

Для восстановления данных в разделе `Key` используют метод

```
function RestoreKey(const Key, FileName: string): Boolean;
```

Метод откроет указанный ключ и перезапишет всю находящуюся в нем информацию данными из файла `FileName`.

Еще один вариант использования данных из архивного файла заключается в переносе информации из реестра в реестр. Например, метод

```
function LoadKey(const Key, FileName: string): Boolean;
```

создает новый раздел `Key` и импортирует в него данные из архива `FileName`. Перед обращением к методу `LoadKey()` необходимо убедиться, что в качестве текущего в системном реестре установлен раздел `HKEY_USERS` или `HKEY_LOCAL_MACHINE`.

Метод

```
function ReplaceKey(const Key, FileName, BackUpFileName: string): Boolean;
```

заменяет архивный файл другим. Таким образом, при рестарте операционной системы ключ получит значения из нового файла. Здесь: `FileName` — имя файла, содержащего новую информацию; `BackUpFileName` — файл, в котором будет сохранена резервная копия заменяемых данных из системного реестра.

Для отказа загрузки из реестра определенного раздела (и всех вложенных в него данных) используйте метод

```
function UnLoadKey(const Key: String): Boolean;
```

Отметим, что метод не удаляет раздел физически, а просто перестает его замечать. Единственное ограничение в том, что игнорируемый раздел должен принадлежать разделу `HKEY_USERS` или `HKEY_LOCAL_MACHINE`.

> ***ЗАМЕЧАНИЕ***
>
> Для того чтобы реестр позволил нам работать с его методами архивирования, экземпляр класса `TRegistry` должен быть создан с режимом доступа `KEY_ALL_ACCESS`.

ГЛАВА 36

Управление процессами

На протяжении всех предшествующих глав мы с вами учились разрабатывать программы для Windows и, надеюсь, стали неплохими специалистами в области программирования на языке Delphi. Однако до настоящих профессионалов нам еще далеко. Разрабатывая приложения, мы опустили весьма важную область знаний о том, как ведет себя программа, выполняющаяся под управлением Windows.

Как бы мы высоко не ценили созданные нами программы, по сути они представляют собой просто статический набор команд. Лишь операционная система превращает программу в полноценное приложение. С точки зрения Windows термин *"приложение"* несколько размыт, вместо него специалисты предпочитают применять термин *"процесс"*. Процесс состоит из:

- *адресного пространства* (address space), представляющего собой набор адресов виртуальной памяти, доступных для процесса;
- исполняемой программы с ее кодом и данными;
- списка дескрипторов системных ресурсов, выделяемых процессу;
- как минимум одного потока, именуемого *потоком управления* (thread of execution), или главным потоком процесса. Именно поток управления и требует от операционной системы времени на обслуживание процесса;
- *контекста защиты* (security context), определяющего привилегии процесса.

Операционная система Windows создает для каждого стартующего процесса иллюзию полной свободы действий. В наиболее распространенных 32-разрядных ОС в первую очередь это выражается в предоставлении процессу 4 Гбайт виртуальной памяти. "Постойте! В моем компьютере всего-навсего 2 Гбайт оперативной памяти! Откуда возьмутся еще 2 Гбайт?" — воскликнет читатель.

Действительно, это так. Но чтобы приложение "увидело" 4 Гбайт, совсем не обязательно, чтобы компьютер реально располагал 4 Гбайт ОЗУ (впрочем, сегодня это не проблема). В нашем случае речь идет не о физической, а о виртуальной памяти. Надо ли повторять, что Windows — большая фокусница? Виртуальная память формируется операционной системой не только из оперативной памяти, но и из файла подкачки, размещенного на жестком диске (или дисках) машины. Как только в ОЗУ заканчивается свободное место, Диспетчер памяти задействует этот файл, перераспределяя данные между ним и оперативной памятью. Вполне естественно, что чем меньше ОЗУ, тем больше диспетчер вынужден обращаться к медленному жесткому диску. Эти операции несколько притормаживают систему, но главное —

результат: приложение и не подозревает, что его так обманывают. Но и это еще не все. Операционная система Windows — многозадачная, это означает, что в ней одновременно могут выполняться несколько процессов. Так что Windows блефует по-крупному — одновременно водя за нос десятки наивных программ, каждая из которых простодушно полагает, что именно она безраздельно господствует на просторах собственных 4 Гбайт памяти.

В момент запуска процесса операционная система создает дескриптор — специальную структуру, описывающую стартующий процесс. Благодаря *дескриптору процесса* легко получить информацию, позволяющую системе управлять процессом. В перечень входят:

- идентификатор процесса (PID, process identificator);
- информация о ресурсах, которыми владеет или собирается завладеть процесс;
- данные о приоритете процесса;
- *контекст задачи*, представляющий собой специальную защищенную область памяти, в которую записываются значения регистров процессора в момент приостановки задачи и, соответственно, откуда считываются данные в момент возобновления выполнения задачи.

Реально, за один и тот же интервал времени, называемый *квантом*, один процессор может выполнять не более одной задачи. Вместе с тем, многозадачная ОС вполне способна одновременно обслуживать несколько процессов. Это достигается благодаря переключению процессора компьютера с исполнения одной задачи на другую. При этом важно помнить, что время выделяется не процессу, а принадлежащим ему потокам. Такое переключение называется *переключением контекста* (context switching). И чем выше производительность процессора, тем более правдоподобна будет иллюзия одновременного выполнении нескольких программ.

Создание процесса

Главную роль в формировании нового процесса в Windows выполняет функция `CreateProcess()`, создающая процесс и поток управления. Сама по себе функция достойна отдельной главы, т. к. обладает широчайшим спектром настроек и, кроме того, может использоваться для синхронизации работы нескольких процессов и потоков. К сожалению, в рамках этой книги такую роскошь мы себе позволить не можем, поэтому ограничимся лишь вводным описанием.

```
function CreateProcess (
                ApplicationName: PWideChar; CommandLine: PWideChar;
                ProcessAttributes, ThreadAttributes: PSecurityAttributes;
                InheritHandles: BOOL; CreationFlags: DWORD;
                Environment: Pointer; CurrentDirectory: PWideChar;
                const StartupInfo: TStartupInfo;
                var ProcessInformation: TProcessInformation): Boolean;
```

В двух первых аргументах функции программист указывает имя запускаемого приложения и (при необходимости) передаваемые ему в командной строке параметры.

Третий и четвертый параметры определяют атрибуты безопасности создаваемого процесса и главного потока, порожденного этим процессом. Физически данные, передаваемые в аргументы, представляют собой указатели на специальную структуру (листинг 37.1) атрибутов защиты.

Листинг 36.1. Структура безопасности TSecurityAttributes

```
type TSecurityAttributes = _SECURITY_ATTRIBUTES;
 PSecurityAttributes = ^TSecurityAttributes;
 _SECURITY_ATTRIBUTES = record
 Length : cardinal; //размер структуры в байтах
 SecurityDescriptor : Pointer;//указатель на дескриптор защиты процесса
 InheritHandle : boolean;//признак наследования возвращаемого дескриптора
end;
```

Защита по умолчанию подразумевает, что полный доступ к процессу получат лишь его создатель и любой член группы администраторов. Все прочие пользователи к процессу не допускаются.

Одной из особенностей Windows является возможность разграничения доступа пользователей к различным ее объектам. Так, например, не обладающий правами администратора пользователь не имеет возможности просматривать системные каталоги или работать с каким-то ресурсом. Более того, обычный пользователь не имеет права запускать процессы или управлять некоторыми процессами, имеющими высокие, недоступные для него атрибуты безопасности. Если вместо указателя этой функции передается `nil`, то включается защита по умолчанию и дескриптор создаваемого процесса наследоваться не сможет.

Особый интерес в структуре `PSecurityAttributes` представляет второе поле — указатель на дескриптор защиты. В этом дескрипторе сосредоточена информация безопасности объекта. Для инициализации дескриптора используется функция `InitializeSecurityDescriptor()`. В параметры безопасности допускается передача неопределенных указателей `nil`, в таком случае ОС присвоит процессу установленный для текущего сеанса уровень прав.

Пятый по счету параметр `InheritHandles` разрешает или запрещает создаваемому процессу наследовать дескрипторы из вызывающего процесса. Обычно при создании процесса в этом параметре передается `false`, это означает, что дочерний процесс не должен наследовать наследуемые описатели, принадлежащие родительскому процессу. При передаче в параметр `true` дочерний процесс копирует все описатели родительского. Таким образом, дескрипторы будут идентичны как в родительском, так и в дочернем процессах. С этого момента все объекты родительского процесса становятся доступны и дочернему. И теперь, чтобы уничтожить какой-то объект, от него должны отказаться оба процесса (например, вызвав функцию `CloseHandle()`).

Набор флагов `CreationFlags` определяет особенности создания процесса (табл. 36.1) и уровень приоритета. Шесть возможных значений приоритета приведены в следующем подразделе. Параметр `Environment` — указатель на настройки окружения нового процесса. Путь к текущему каталогу нового процесса — `CurrentDirectory`. Параметр `StartupInfo` — это специальная структура, описывающая характеристики окна создаваемого приложения. `ProcessInformation` — указатель на структуру, используемую созданным процессом. Если функция выполнилась успешно, то методом возвращается значение `true`.

***Таблица 36.1.** Основные флаги параметра `CreationFlags`*

Флаг	Описание
`CREATE_NEW_CONSOLE`	Для нового процесса создается новая консоль, наследующая все параметры родительской консоли

***Таблица 36.1** (окончание)*

Флаг	Описание
CREATE_SUSPENDED	Процесс создается в приостановленном виде, для его запуска необходимо обращение к функции ResumeThread()
DEBUG_PROCESS	Флаг указывает на то, что данный процесс и все его дочерние процессы запущены в режиме отладки
DEBUG_ONLY_THIS_PROCESS	Процесс запущен в режиме отладки
INHERIT_CALLER_PRIORITY	Процесс наследует приоритет запустившего его процесса

В листинге 36.2 приведен пример запуска процесса Блокнота (файл notepad.exe), в командной строке процессу передается указание открыть файл demo.txt.

Листинг 36.2. Создание процесса

```
var StartupInfo : TStartupInfo;
    ProcessInformation : TProcessInformation;
    SD : TSecurityDescriptor;
    SA : TSecurityAttributes;
begin
  //инициализация и заполнение атрибутов безопасности
  InitializeSecurityDescriptor(@SD,SECURITY_DESCRIPTOR_REVISION);
  SA.nLength:=SizeOf(TSecurityAttributes);
  SA.lpSecurityDescriptor:=@SD;
  SA.bInheritHandle:=True;

  FillChar(StartupInfo,SizeOf(StartupInfo),#0); //очищаем структуру
                                                // StartupInfo
  CreateProcess(PChar('c:\windows\notepad.exe'),//запускаемое приложение
                PChar('open c:\demo.txt'), //командная строка
                @SA,
                @SA,
                false,
                NORMAL_PRIORITY_CLASS,     //нормальный приоритет
                nil,
                nil,
                StartupInfo,
                ProcessInformation);
end;
```

Доступ к процессу

Для получения доступа к определенному процессу, исполняющемуся в ОС, следует воспользоваться функцией

```
function OpenProcess(DesiredAccess: DWORD; InheritHandle: BOOL;
                     ProcessId: DWORD): THandle;
```

Первый параметр функции контролирует права вызвавшего функцию приложения. Наиболее универсальный аргумент — константа PROCESS_ALL_ACCESS, которая сигнализирует, что

мы должны обладать всеми возможными правами доступа. Второй параметр `InheritHandle` вам уже знаком по функции `CreateProcess()`, он определяет особенности наследования дескрипторов. Последний параметр `ProcessId` должен содержать идентификатор процесса. В случае успешного выполнения функция возвратит дескриптор запрашиваемого процесса.

> ***Внимание!***
>
> Для получения идентификатора процесса `ProcessId` можно воспользоваться услугами функций из состава библиотеки Tool Help Library, предоставляющей информацию о протекающих в системе процессах.

Для выяснения дескриптора процесса, созданного нашим приложением, обращаемся к функции

```
function GetCurrentProcess : THandle;
```

Функция в аргументах не нуждается и возвращает дескриптор процесса, в рамках которого она была вызвана.

Приоритет процесса

Количество квантов процессорного времени, выделяемого процессу (точнее, потоку управления), определяется его приоритетом. Зависимость прямо пропорциональная — чем выше приоритет, тем большего внимания его удостоит процессор (или процессоры) вашего компьютера.

Система Windows поддерживает шесть категорий приоритетов процессов:

- фоновый (`IDLE_PRIORITY_CLASS`);
- нормальный заднего плана (`BELOW_NORMAL_PRIORITY_CLASS`);
- нормальный (`NORMAL_PRIORITY_CLASS`);
- нормальный переднего плана (`ABOVE_NORMAL_PRIORITY_CLASS`);
- высокий (`HIGH_PRIORITY_CLASS`);
- процесс реального времени (`REALTIME_PRIORITY_CLASS`).

По умолчанию, при создании нового процесса ему присваивается нормальный приоритет.

Для системного программиста большой практический интерес предоставляет возможность ускорения или замедления выполняющегося процесса. Для достижения этой цели достаточно сделать два шага: во-первых, получить дескриптор процесса (с помощью функции `OpenProcess` или `GetCurrentProcess`) и, во-вторых, изменить его приоритет.

Тем или иным способом приобретя дескриптор интересующего нас процесса, приступаем к главному — меняем приоритет задачи. Для этого вызовем функцию

```
function SetPriorityClass(Prc : THandle;
                          PriorityClass : Cardinal) : Boolean;
```

Первый аргумент требует дескриптор процесса, второй — константа приоритета.

Если планируется выяснить приоритет процесса — воспользуйтесь функцией

```
function GetPriorityClass(Prc : THandle) : Cardinal;
```

В зависимости от приоритета функция возвратит одну из констант приоритета процесса. Представленный в листинге 36.3 код демонстрирует способ определения приоритета выполняющейся задачи.

Листинг 36.3. Выяснение приоритета текущего процесса

```
procedure TfrmMain.btnGetPriorityClassClick(Sender: TObject);
var prc : THandle;
    msg : string;
begin
  prc:=GetCurrentProcess();     //получаем дескриптор процесса
  case GetPriorityClass(prc) of
    IDLE_PRIORITY_CLASS     : msg:='IDLE_PRIORITY_CLASS';
    NORMAL_PRIORITY_CLASS   : msg:='NORMAL_PRIORITY_CLASS';
    HIGH_PRIORITY_CLASS     : msg:='HIGH_PRIORITY_CLASS';
    REALTIME_PRIORITY_CLASS : msg:='REALTIME_PRIORITY_CLASS';
  end;
  ShowMessage(msg);
end;
```

Время выполнения процесса

При анализе производительности системы часто возникает необходимость осуществить хронометраж процесса, т. е. выяснить, какое время уделил процессор компьютера тому или иному из процессов.

```
function GetProcessTimes(Prc : THandle;
 var CreationTime, ExitTime, KernelTime, UserTime : TFileTime) : boolean;
```

Здесь: `Prc` — дескриптор процесса; `CreationTime` — время создания; `ExitTime` — время приостановки процесса; `KernelTime` — время выполнения процесса в режиме ядра, здесь хранится суммарное время выполнения всех потоков в режиме ядра; `UserTime` — время выполнения процесса в пользовательском режиме. При работе в Delphi обработка возвращаемых этой функцией значений сопряжена с некоторыми трудностями преобразования данных: четыре последних параметра представлены типом данных `TFileTime` (листинг 36.4).

Листинг 36.4. Объявление записи `TFileTime`

```
type TFileTime = record
     dwLowDateTime  : Cardinal;
     dwHighDateTime : Cardinal;
end;
```

В этом типе данных время описано в несколько непривычном для программиста Delphi формате: 32-разрядные элементы записи объединяются в учетверенное слово и образуют хранилище для 64-разрядного значения. За единицу измерения принят интервал в 100 наносекунд, а за точку отсчета — полночь 1 января 1601 года. За декодирование значений такого типа отвечают функции Windows API `FileTimeToSystemTime()` и `SystemTimeToFileTime()`.

Функции обеспечивают взаимное преобразование `TFileTime` в `TSystemTime` и наоборот. К сожалению, на этом трудности не заканчиваются, т. к. тип данных `TSystemTime` по-прежнему несовместим с базовым для Delphi типом `TDateTime`. В этом нетрудно убедиться, единожды взглянув на структуру `TSystemTime` (листинг 36.5).

Листинг 36.5. Объявление записи TSystemTime

```
type PSystemTime = ^TSystemTime;
    TSystemTime = record
    wYear: Word;         //год, например, 2013
    wMonth: Word;        //месяц, например 1, 2, ..., 12
    wDayOfWeek: Word;    //день недели 1, 2, ..., 7
    wDay: Word;          //день месяца 1, 2, ..., 31
    wHour: Word;         //час 1, 2, ..., 23
    wMinute: Word;       //минута 1, 2, ..., 59
    wSecond: Word;       //секунда 1, 2, ..., 59
    wMilliseconds: Word;//миллисекунда 1, 2, ..., 999
  end;
```

Однако в этом формате записи время "разложено по полочкам", и уже дело техники — перевод значений, представленных структурой, в TDateTime: вспомните функции EncodeDateTime() и DecodeDateTime(), пример работы с которыми предложен в листинге 36.6.

Листинг 36.6. Время выполнения текущего процесса

```
var prc : THandle;
    CreationTime, ExitTime, KernelTime, UserTime : TFileTime;
    SystemTyme : TSystemTime;
    DateTime   : TDateTime;
begin
  prc:=GetCurrentProcess();
  GetProcessTimes(prc, CreationTime, ExitTime, KernelTime, UserTime);
  FileTimeToSystemTime(CreationTime, SystemTyme);
  DateTime:=EncodeDateTime(SystemTyme.wYear,
                           SystemTyme.wMonth,
                           SystemTyme.wDay,
                           SystemTyme.wHour,
                           SystemTyme.wMinute,
                           SystemTyme.wSecond,
                           SystemTyme.wMilliseconds);
  {...}
end;
```

Завершение процесса

Наилучший вариант завершения процесса — это окончание выполнения процесса только после возврата управления входной функцией потока управления. Только в этом случае можно утверждать, что процесс корректно освободит все свои объекты, а ОС очистит занимаемую стеком главного потока память и уменьшит счетчик на единицу.

Процесс может прекратить свое существование самостоятельно, для этого он должен вызвать процедуру

```
procedure ExitProcess(ExitCode : cardinal);
```

Но если процесс не желает уходить из жизни добровольно, то применяют функцию

```
function TerminateProcess(hProcess : THandle ;
                          ExitCode : cardinal) : Boolean;
```

Функция может быть вызвана родительским процессом или любым другим процессом, знающим дескриптор разрушаемого процесса. В обеих функциях параметр `ExitCode` содержит код завершения процесса. Этот код можно получить с помощью функции `GetExitCodeProcess()`.

Сбор информации о процессах Windows

Для исследования выполняющихся в Windows задач применяются информационные функции из библиотеки Tool Help Library. Этих функций немногим больше десятка, они нацелены на предоставление нам данных о выполняющихся на момент опроса процессах. Особо подчеркну словосочетание "на момент опроса", и вот почему: Windows — это динамическая система, в которой задачи вправе рождаться и умирать ежесекундно. Поэтому функции Tool Help Library делают мгновенный снимок состояния системы в указанный программистом момент времени и затем "препарируют" полученные данные, извлекая из снимка требуемую информацию. Если есть необходимость в непрерывном отслеживании состояния системы, то применяйте таймер, "фотографирующий" ОС с определенной периодичностью. Роль фотографа отводится функции

```
function CreateToolhelp32Snapshot(Flags, ProcessID : Cardinal) : THandle;
```

Параметр `Flags` определяет, какую информацию мы планируем включать в снимок (табл. 36.2). Если мы исследуем конкретный процесс, то во второй параметр направляется дескриптор этого процесса `ProcessID`. При успешном выполнении метод возвращает указатель на структуру в памяти, называемую снимком.

***Таблица 36.2.** Возможные значения параметра `Flags` функции `CreateToolhelp32Snapshot()`*

Значение	Описание
`TH32CS_INHERIT`	Наследовать дескриптор снимка
`TH32CS_SNAPALL`	Включить в снимок всю информацию
`TH32CS_SNAPHEAPLIST`	Включить в список данные о кучах в процессе `ProcessID`
`TH32CS_SNAPMODULE`	Включить в список данные о модулях в процессе `ProcessID`
`TH32CS_SNAPPROCESS`	Включить в снимок данные о процессах
`TH32CS_SNAPTHREAD`	Включить в снимок данные о потоках

Для исследования имеющегося снимка на предмет получения информации о процессах применяют работающие в паре функции:

```
function Process32First(Snapshot : THandle;
                        var lppe : TProcessEntry32) : boolean;
function Process32Next(Snapshot : THandle;
                       var lppe : TProcessEntry32) : boolean;
```

Параметры методов идентичны. Первый из них — дескриптор снимка, второй — указатель на структуру `TProcessEntry32`, в которую они помещают результаты исследований.

Как мы уже отмечали, сбор данных производится совместно. Первоначально вызывается метод `Process32First()`, проверяющий наличие интересующей нас информации в снимке. Если такая информация имеется, то метод возвращает `true`. Затем наступает черед функции `Process32Next()`. Этот метод вызывается в цикле до тех пор, пока им не будет возвращено значение `false`. Это признак того, что все данные собраны.

Переключим внимание на применяемую при исследовании задач структуру `TProcessEntry32` (листинг 36.7).

Листинг 36.7. Объявление записи `TProcessEntry32`

```
type TProcessEntry32 = packed record
    dwSize: DWORD;             //размер записи
    cntUsage: DWORD;           //счетчик ссылок на процесс
    th32ProcessID: DWORD;      //идентификатор процесса
    th32DefaultHeapID: DWORD; //идентификатор кучи процесса
    th32ModuleID: DWORD;       //модуль, связанный с процессом
    cntThreads: DWORD;         //количество дочерних потоков
    th32ParentProcessID: DWORD; //родительский процесс
    pcPriClassBase: Longint;  //базовое значение приоритета процесса
    dwFlags: DWORD;            //не используется
    szExeFile: array[0..MAX_PATH — 1] of Char;//имя исполняемого файла
end;
```

Все поля записи снабжены комментариями и в дополнительном обсуждении не нуждаются, но за небольшим исключением. Перед первым вызовом запись типа `TProcessEntry32` должна быть проинициализирована — полю `dwSize` передается значение размера структуры `SizeOf(TProcessEntry32)`.

Рассмотрим упрощенный пример сбора данных о процессах системы (листинг 36.8). Для этого нам понадобится создать новое приложение и на поверхность формы поместить компонент `TListView`. Переведите его в табличный режим просмотра (`ViewStyle=vsReport`) и в редакторе колонок создайте три колонки для отображения: идентификатора процесса, идентификатора родительского процесса и числа дочерних потоков. Подключите к проекту модуль `tlhelp32` и повторите приведенный в листинге 36.8 код.

Листинг 36.8. Получение "снимка" выполняющихся задач

```
uses tlhelp32;
...
procedure TfrmMain.GetProcessInformation;
var PrcEntry : TProcessEntry32;
    Snapshot : THandle;
begin
  PrcEntry.dwSize:=SizeOf(PrcEntry); //установка размера структуры
  Snapshot:=CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS,0); {получаем
                                                    снимок процессов}
```

```
  {начало цикла перебора процессов в снимке}
  if Process32First(Snapshot, PrcEntry)=true then
    repeat
      {в списке ListView1 создаем новый элемент с информацией о процессе}
      with ListView1.Items.Add do
      begin
        Caption:=PrcEntry.szExeFile;
        SubItems.Add(IntToStr(PrcEntry.th32ProcessID));
        SubItems.Add(IntToStr(PrcEntry.th32ParentProcessID));
        SubItems.Add(IntToStr(PrcEntry.cntThreads));
      end;
    until not Process32Next(Snapshot, PrcEntry); {условие продолжения
                                                  перебора}
  CloseHandle(Snapshot); {освобождение дескриптора снимка процессов}
end;
```

На рис. 36.1 представлен экранный снимок утилиты, собирающей сведения о процессах с помощью библиотеки Tool Help Library.

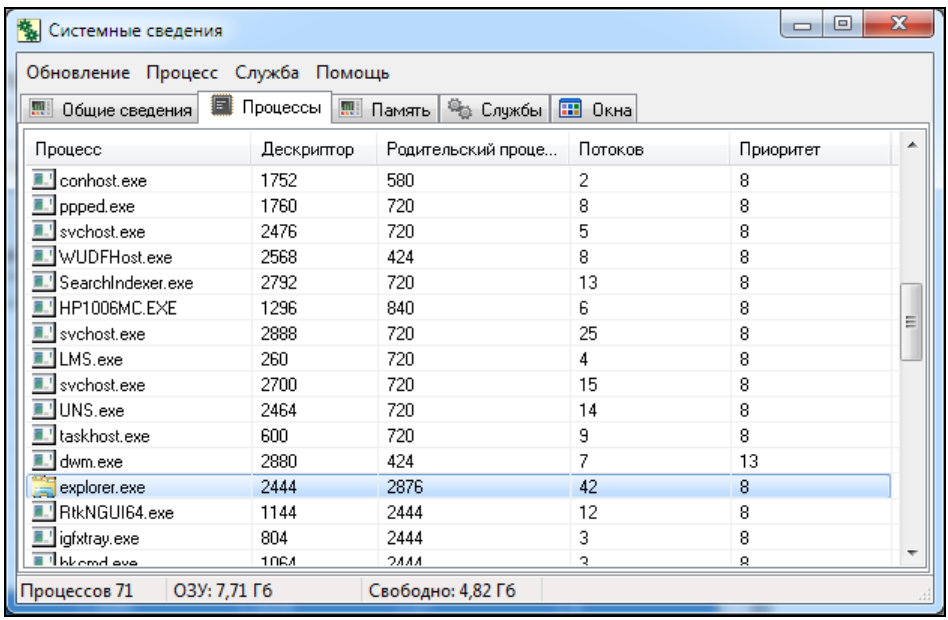


Рис. 36.1. Сбор сведений о процессах Windows

Замечание

Кроме рассмотренных функций, в состав Tool Help Library входят методы, позволяющие получить информацию о выполняющихся в данный момент потоках (`Thread32First` и `Thread32Next`), модулях (`Module32First` и `Module32Next`) и кучах памяти (`Heap32First`, `Heap32Next`, `Heap32ListFirst` и `Heap32ListNext`).

Получение сведений о версии ОС

Зачастую для управления процессами следует владеть полной информацией о версии операционной системы, под управлением которой выполняется наше приложение. Для получения этих сведений в Delphi XE2 следует обратиться к объявленной в модуле `System.SysUtils` записи (листинг 36.9).

Листинг 36.9. Запись `TOSVersion`

```
type TOSVersion = record
  public type
    TArchitecture = (arIntelX86, arIntelX64);
    TPlatform = (pfWindows, pfMacOS);
  private
    class var FArchitecture: TArchitecture;
    class var FBuild: Integer;
    class var FMajor: Integer;
    class var FMinor: Integer;
    class var FName: string;
    class var FPlatform: TPlatform;
    class var FServicePackMajor: Integer;
    class var FServicePackMinor: Integer;
    class constructor Create;
  public
    class function Check(AMajor: Integer): Boolean; overload; static; inline;
    class function Check(AMajor, AMinor: Integer): Boolean; overload; static;
inline;
  class function Check(AMajor, AMinor, AServicePackMajor: Integer): Boolean;
overload; static; inline;
    class function ToString: string; static;
    class property Architecture: TArchitecture read FArchitecture;
    class property Build: Integer read FBuild;
    class property Major: Integer read FMajor;
    class property Minor: Integer read FMinor;
    class property Name: string read FName;
    class property Platform: TPlatform read FPlatform;
    class property ServicePackMajor: Integer read FServicePackMajor;
    class property ServicePackMinor: Integer read FServicePackMinor;
  end;
```

Запись возвращает все сведения об установленной на компьютере ОС Windows или Mac OS X (если идет речь о проекте FireMonkey).

ГЛАВА 37

Многопоточные приложения

В предыдущей главе мы уже говорили, что в момент старта процесса под управлением Windows обязательно создается главный поток процесса (поток управления). Перед главным потоком приложения поставлена серьезная задача — он отвечает за взаимодействие с ОС и за выполнение кода программы. Кроме обязательного потока управления процесс имеет возможность обладать некоторым набором вспомогательных потоков (нитей). Например, текстовый процессор Microsoft Word (в котором набирались эти строки) одновременно с вводом текста разбивает документ на страницы, контролирует мои знания русского языка (безжалостно подчеркивая грамматические и синтаксические оплошности), отправляет задание на печать и решает ряд других менее очевидных для обычного пользователя задач. Все это происходит одновременно благодаря тому, что в MS Word запущено несколько потоков. Каждый из потоков выполняет поставленную перед ним подзадачу и при желании может быть приостановлен или запущен вновь.

Поток *TThread*

Для программной реализации потока в Delphi объявлен абстрактный класс `TThread`, на основе которого программист создаст свой полнофункциональный поток. На палитре компонентов элемент управления `TThread` вы не найдете. Для того чтобы самое обычное приложение сделать многопоточным, требуется к уже существующему приложению добавить специальный программный модуль. Для этого после создания нового приложения выберите пункт меню **File | New | Other** и на вкладке **New** диалогового окна **New Items** найдите значок **ThreadObject**. После нажатия кнопки **OK** в новом диалоговом окне требуется ввести имя нового класса — потока. Например, `TDemoThread`, как показано на рис. 37.1.

В результате выполненных действий Delphi создает представленный в листинге 37.1 шаблон, предназначенный для описания потока. Обратите внимание, что в секции `Protected` среда проектирования уже подготовила процедуру `Execute()`. В теле этой процедуры программист располагает программный код, который станет исполняться в отдельном потоке. Если подзадача вызывается многократно, то строки кода помещаются внутрь тела цикла, выход из которого завершает работу потока.

Листинг 37.1. Модуль потока `TDemoThread`

```
unit Unit2;

interface
uses System.Classes;
```

```
type TDemoThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TDemoThread }

procedure TDemoThread.Execute;
begin
  { код потока }
end;
end.
```

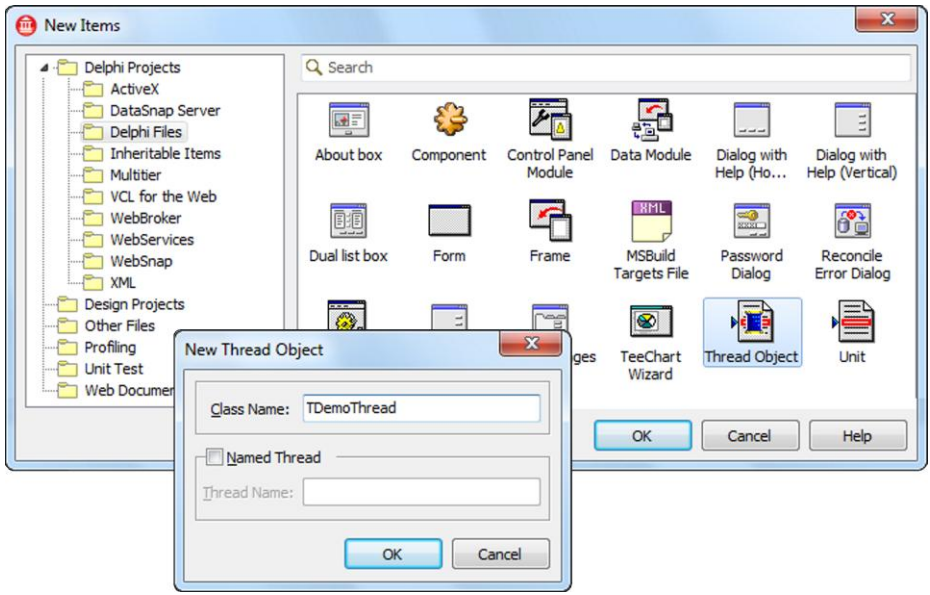


Рис. 37.1. Окна создания потока и назначения имени класса потока

При старте приложения автоматическое создание потока не произойдет, для этого следует явно обратиться к конструктору потока:

```
constructor Create(CreateSuspended : Boolean);
```

Единственный параметр конструктора определяет, каким образом производится запуск потока. Если параметр `CreateSuspended` установлен в `false`, то поток стартует немедленно сразу после создания экземпляра класса. Иначе поток создается в спящем режиме. В последнем случае для запуска потока вызывайте метод

```
procedure Start;
```

Говоря образно, эта процедура заставит биться сердце потока.

Для временной приостановки выполнения потока используйте процедуру

```
procedure Suspend;
```

Для возобновления работы приостановленного потока

```
procedure Start;
```

Причем возобновление выполнения потока произойдет именно с того места (с той строки кода), где он был приостановлен. Два названных метода дублируются свойством

```
property Suspended : Boolean;
```

Если вы установите свойство в `true`, то поток приостановится, `false` — вновь активизируется.

> ***Замечание***
>
> В более ранних версиях Delphi вместо метода `Start()` применялся метод `Resume()`. Начиная с Delphi 2010, применение этого метода не рекомендуется.

Для временной приостановки потока программисты Delphi часто пользуются процедурой класса

```
class procedure Sleep(Timeout: Integer); static;
```

В качестве параметра передается число миллисекунд, на которые необходимо заморозить подзадачу.

> ***Замечание***
>
> Основное достоинство метода `Sleep()` в том, что поток, переведенный в состояние сна, не снижает производительность системы, т. к. не требует выделения ему квантов времени.

Признаком того, что поток завершил выполнение своей задачи, выступает значение `true`, возвращаемое свойством

```
property Finished: Boolean;
```

Для полной остановки потока вызывайте метод

```
procedure Terminate;
```

Этот способ остановки называется мягким. Процедура просто-напросто присваивает значение `true` свойству

```
property Terminated: Boolean;
```

В этом случае Delphi не настаивает на немедленном прекращении работы потока и разрешает ему достичь логического конца.

Если есть мягкий способ остановки потока, то где-то в недрах Windows можно обнаружить и "жесткий". Если вы сторонник радикальных решений и категорически настаиваете на том, чтобы поток завершился немедленно, то пригодится функция из арсенала Windows API.

```
function TerminateThread(hThread: THandle; ExitCode: Cardinal): boolean;
```

Здесь: `hThread` — дескриптор потока; `ExitCode` — программный код выхода из потока, в простейшем случае передавайте значение 0. Если функция завершится успешно, то она вернет ненулевое значение.

Очень часто проверка состояния свойства служит сигналом к выходу из цикла обработки задачи потока (листинг 37.2).

Листинг 37.2. Шаблон кода основного метода потока

```
procedure TDemoThread.Execute;
begin
  repeat
    {исполняемый код потока}
  until Terminated;
end;
```

Останов потока сопровождается вызовом обработчика события

```
property OnTerminate: TNotifyEvent;
```

Если установить в `true` свойство

```
property FreeOnTerminate: Boolean;
```

то сразу после останова уничтожится экземпляр потока. Иначе за вызов деструктора потока отвечает программист.

Если выполнение метода `Execute()` прервалось в результате ошибки, и ошибка не была обработана в рамках этого метода, то в свойство только для чтения

```
property FatalException: TObject; //только для чтения
```

будет записан объект ошибки.

Еще одним способом, косвенно проверяющим корректность завершения работы потока, является использование свойства

```
property ReturnValue: Integer;
```

Это служебная переменная, в которую можно записывать данные, например, о количестве запусков потока. Свойство `ReturnValue` часто применяется совместно с методом ожидания `WaitFor()`. Функция предназначена для организации ожидания, пока какой-нибудь другой поток не выполнит поставленную перед ним задачу.

Существует альтернативный доступ к значению `ReturnValue`, его обеспечивает метод класса

```
class procedure SetReturnValue(Value: Integer); static;
```

Благодаря методу мы сможем изменить возвращаемое потоком значение, даже не имея прямого доступа к экземпляру потока.

Для взаимодействия с функциями Windows API может пригодиться дескриптор потока, его вы найдете в свойстве

```
property Handle: THandle;
```

Современные версии Windows поддерживают работу с несколькими процессорами, распределяя задачи между процессорами (или их ядрами, если речь идет о многоядерном процессоре). Класс `TThread` способен проинформировать программиста о процессоре компьютера. Свойство

```
class property IsSingleProcessor: Boolean;
```

в состоянии `true` уведомляет разработчика, что поток будет запущен на однопроцессорной станции, если же свойство возвратит `false`, то стоит обратиться к свойству

```
class property ProcessorCount: Integer;
```

возвращающему число процессоров (ядер).

ВНИМАНИЕ!

Разработчику рекомендуется не запускать в приложении более 16 потоков на один процессор, иначе при превышении этого значения будет существенно снижена производительность системы.

Если в приложении допускается исполнение нескольких потоков, то мы имеем право попросить текущий процессор перейти на обслуживание очередного потока. Для этого предназначен метод класса

```
class procedure Yield; static;
```

Заметим, что окончательное решение на постановку потока на обслуживание всегда останется за операционной системой.

Метод ожидания

При проектировании многопоточного приложения, в котором несколько потоков обслуживают один и тот же ресурс (файл, коммуникационный порт и т. п.) или выполняют общий участок кода, надо исключить конфликт совместного доступа к ресурсу. Большую помощь в этом оказывает метод ожидания, позволяющий одному потоку дождаться завершения другого.

```
function WaitFor: LongWord;
```

Функция возвращает значение, содержащееся в свойстве `ReturnValue` ожидаемого потока (табл. 37.1).

Таблица 37.1. *Значения, возвращаемые методом* `WaitFor()`

Значение	Описание
`WAIT_OBJECT_0`	Контролируемый объект перешел в сигнальное состояние. Это признак того, что выполняющийся ранее поток корректно завершил свою задачу
`WAIT_TIMEOUT`	Хотя время тайм-аута истекло, но состояние опрашиваемого объекта синхронизации несигнальное. Это означает, что ресурс пока захвачен другим потоком
`WAIT_ABANDONED`	Выполнение обладавшего объектом синхронизации потока завершилось, но объект по-прежнему захвачен этим потоком. В виду некорректности такого положения вещей объект синхронизации передается во владение вызвавшего его потока и переводится в несигнальное состояние
`WAIT_FAILED`	Ошибка выполнения метода

ЗАМЕЧАНИЕ

Метод `WaitFor()` применяется для совместной работы не только с потоками, но и с такими объектами синхронизации, как события, мьютексы и семафоры. Ключевая особенность метода ожидания заключается в том, что он не возвращает результат немедленно, а ждет, пока не будет удовлетворен некоторый перечень заданных программистом критериев. Таким образом, поток, обратившийся к объекту синхронизации при помощи метода `WaitFor()`, превращается в пленника, ожидая от метода ответа. Соответственно выполнение потока приостанавливается, и таким образом решается задача синхронизации.

Управление приоритетом потока

Обычный поток приложения (точно так, как и поток управления приложения) обладает определенным приоритетом, от которого зависит, сколько квантов времени будет выделено процессором компьютера для обработки подзадачи. Приоритет потока определяется значениями двух составляющих: приоритетом процесса-владельца потока и собственно приоритетом этого потока. Поэтому приоритет обычного потока часто называют *относительным приоритетом*. Такое название объясняется тем, что два потока, скажем, с приоритетом `tpNormal`, но порожденные процессами с различными приоритетами, не смогут претендовать на одинаковое процессорное время. Реальный приоритет потока определяется приоритетом порождающего его процесса и своим приоритетом. В Windows обычно приоритет потока оценивается значением, лежащим в диапазоне от 1 до 31 (табл. 37.2).

Таблица 37.2. *Расчет приоритета потока приложения*

Приоритет процесса (потока управления)	Приоритет обычного потока					
	Фон	**Низший**	**Пониженный**	**Нормальный**	**Повышенный**	**Высший**
Фоновый	1	2	3	4	5	6
Нормальный заднего плана	1	5	6	7	8	9
Нормальный	1	6	7	8	9	10
Нормальный переднего плана	1	7	8	9	10	11
Высокий	1	11	11	12	14	15
Реального времени	16	22	23	24	25	26

В Delphi приоритет потока определяется свойством

```
property Priority: TThreadPriority; //по умолчанию tpNormal
type TThreadPriority = (tpIdle,          //фоновый приоритет
                        tpLowest,        //низкий приоритет
                        tpLower,         //пониженный приоритет
                        tpNormal,        //нормальный приоритет
                        tpHigher,        //повышенный приоритет
                        tpHighest,       //высочайший приоритет
                        tpTimeCritical); //приоритет реального времени
```

Внимание!

Без особой на то необходимости старайтесь не создавать потоков с приоритетом выше нормального. Такой поступок может привести к тому, что системные ресурсы будут нацелены на обслуживание только этой задачи.

Замечание

В состав Delphi входит специальная утилита, предназначенная для просмотра потоков приложения. Для обращения к этой утилите выберите пункт меню **View | Debug Windows | Threads**.

Время выполнения потока

В составе Windows API предусмотрена возможность оценки затрат процессорного времени на любой из потоков процесса:

```
function GetThreadTimes(hThread : THandle; var CreationTime,
                  ExitTime, KernelTime, UserTime : TFileTime) : boolean;
```

В первый аргумент функции следует направить дескриптор исследуемого потока, остальные параметры и порядок их применения аналогичны параметрам функции `GetProcessTimes()`.

Синхронный и асинхронный вызовы внешнего метода

Если из тела основного метода потока `Execute()` необходимо осуществить вызов внешнего метода или обратиться к свойству компонента библиотеки VCL, то не следует это делать напрямую, вместо этого надо использовать специализированные методы `Synchronize()` или `Queue()`. Задача методов — исключение взаимных блокировок потоков.

> ***ЗАМЕЧАНИЕ***
>
> Блокировка потоков возникает в том случае, когда поток или несколько потоков пытаются одновременно получить доступ к одному и тому же ресурсу. Если этот ресурс захвачен третьей стороной, например главным потоком приложения, и корректный захват ресурса невозможен, то потоки попадают в тупик. Такая ситуация называется *взаимной блокировкой*.

Существует несколько перегружаемых версий методов, мы остановимся на наиболее простых:

```
procedure Synchronize(Method: TThreadMethod); overload;
procedure Queue(AMethod: TThreadMethod); overload;
```

Методы гарантируют, что к объекту (методу объекта) одновременно получит доступ только один поток. В качестве аргумента процедуры передается имя вызываемого метода. Разница между `Synchronize()` или `Queue()` заключается в порядке ожидания ответа от вызванного внешнего метода. Процедура `Synchronize()` работает в *синхронном режиме*. Это означает, что поток, обратившись к внешнему методу, приостановит свою работу до тех пор, пока не получит ответа о том, что внешний метод выполнен. Процедура `Queue()` отдает предпочтение *асинхронному режиму*, поток просто ставит задачу в очередь и, не дожидаясь ответа о ее выполнении, продолжает свою работу.

Пример многопоточного приложения

Для приобретения практических навыков разработки многопоточного приложения предлагаю рассмотреть небольшой пример. Мы напишем приложение, в составе которого будут функционировать три одинаковых потока, выполняющих идентичные задачи по заполнению трех массивов случайными значениями. Для этого создаем новый проект. Традиционно переименуем главную форму проекта в `frmMain`. Сохраняя проект, соответствующий главной форме, модуль `unit1` назовем main.pas, имя проекта выберите на свое усмотрение.

Разместите на главной форме следующие элементы управления:

- три шкалы `TProgressBar`, они позволят визуализировать работу каждого из потоков;
- в помощь шкалам — три метки `TLabel`, которые станут отображать число ячеек массива, заполненных каждым из потоков;

- три флажка `TCheckBox`. Эти элементы управления позволят приостанавливать/возобновлять работу потоков;
- три ползунка `TTrackBar`. С помощью них будут изменяться приоритеты потоков. Одновременно выберите все компоненты `TTrackBar` и их свойству `Max` присвойте значение 3. Значение 0 будет соответствовать минимальному приоритету `tpIdle`, значение 3 — нормальному `tpNormal`.

У соответствующих компонентов `TCheckBox` и `TTrackBar` поменяйте значение свойства `tag`:

- `CheckBox1` и `TrackBar1` — `tag=1`;
- `CheckBox2` и `TrackBar2` — `tag=2`;
- `CheckBox3` и `TrackBar3` — `tag=3`.

С настройкой элементов управления покончено. Займемся созданием класса потока. Для этого выберите пункт меню **File | New | Other**. На вкладке **New** найдите значок **ThreadObject** и нажмите кнопку **OK**. Назовите класс создаваемого потока `TArrayThread`. Вновь созданный модуль с шаблоном кода потока сохраните под именем ThreadUnit.pas.

В соответствии с приведенным в листинге 37.3 кодом внесите изменения в модуль `ThreadUnit`.

Листинг 37.3. Код модуля потока

```
unit ThreadUnit;
interface

uses Classes, Windows, SysUtils, StdCtrls, ComCtrls;

type
  TArrayThread = class(TThread)
  protected
    procedure Execute; override;
  public
     Ar:Array of Integer;
     ProgressBar : TProgressBar;
     Lbl : TLabel;
  end;

implementation

procedure TArrayThread.Execute;
var i : integer;
begin
  Randomize;
  i:=Low(Ar);
  while (Terminated=FALSE) AND (i<High(Ar)-1) do
  begin
    SetReturnValue(i);
    Ar[i]:=Random(1000);
    INC(i);
```

```
    Synchronize(procedure {Анонимная процедура}
                begin
                  ProgressBar.Position:=ProgressBar.Position+1;
                  Lbl.Caption:=IntToStr(ReturnValue);
                end);
  end;
  Terminate;
  Lbl.Caption:='Завершено';
end;
end.
```

В секции public потока опубликованы два поля: ProgressBar:TProgressBar и lbl:TLabel. Они предназначены для организации взаимодействия потока с размещенными на главной форме шкалой и меткой (эту связь мы установим позднее в событии создания главной формы проекта). Основной метод Execute() выполняется до тех пор, пока не будет заполнен массив или не будет разрушен поток. Внутри метода реализован цикл while..do, предназначенный для заполнения массива случайным числами (а точнее говоря, для "пожирания" процессорного времени). В последнем операторе цикла осуществляется вызов метода Synchronize(), организующего обновление данных в принадлежащих потоку шкале и метке. На этом работа с модулем потоков завершена — переходим к главной форме проекта.

В секции переменных модуля главной формы объявите три потока и константу, определяющую число элементов в массиве (листинг 37.4).

Листинг 37.4. Переменные и константы главной формы проекта

```
var frmMain: TfrmMain;
    Thread1,Thread2,Thread3:TArrayThread; //3 потока
const N=100000;                           //размер массива
```

В обработчике события OnCreate() формы вызываемого в момент создания формы поместите строки кода, создающие и запускающие все три потока (листинг 37.5).

Листинг 37.5. Создание главной формы

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  //предельные значения для всех шкал
  ProgressBar1.Max:=N; ProgressBar2.Max:=N; ProgressBar3.Max:=N;

  Thread1:=TArrayThread.Create(True); //1 поток
  Thread1.Priority:=tpIdle; //самый низкий приоритет
  Thread1.ProgressBar:=ProgressBar1; //связь со шкалой
  Thread1.Lbl:=Label1;     //связь с меткой
  SetLength(Thread1.Ar,N); //распределяем память под массив

  Thread2:=TArrayThread.Create(True); //2 поток
  Thread2.Priority:=tpIdle;
  Thread2.ProgressBar:=ProgressBar2;
```

```
  Thread2.Lbl:=Label2;
  SetLength(Thread2.Ar,N);

  Thread3:=TArrayThread.Create(True); //3 поток
  Thread3.Priority:=tpIdle;
  Thread3.ProgressBar:=ProgressBar3;
  Thread3.Lbl:=Label3;
  SetLength(Thread3.Ar,N);

  Thread1.Start; Thread2.Start; Thread3.Start; //старт всех потоков
end;
```

В момент создания потоков мы связываем их с обслуживающими поток элементами управления `TProgressBar` и `TLabel`. Кроме того, распределяется память для массива целых чисел. Обращаю внимание на то, что все три экземпляра потока создаются приостановленными и стартуют практически вместе после того, как к ним были применены все настройки.

Правила хорошего тона требуют, чтобы программист самостоятельно уничтожал все созданные вручную потоки. Это мы сделаем в рамках события закрытия формы (листинг 37.6).

Листинг 37.6. Закрытие главной формы

```
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Thread1.Terminate;
  Thread2.Terminate;
  Thread3.Terminate;
end;
```

Настал звездный час элемента `TTrackBar`, его задача — устанавливать приоритет для соответствующего потока. Выберите любой из этих компонентов и опишите его обработчик события `OnChange()` так, как предложено в листинге 37.7.

Листинг 37.7. Изменение приоритета потока

```
procedure TfrmMain.TrackBar1Change(Sender: TObject);
var Priority : TThreadPriority;
begin
  Priority:=TThreadPriority((Sender as TTrackBar).Position);
  {0 = tpIdle, 1 = tpLowest, 2 = tpLower, 3 = tpNormal}
  case (Sender as TTrackBar).Tag of
    1 : Thread1.Priority:=Priority; //приоритет 1 потока
    2 : Thread2.Priority:=Priority; //приоритет 2 потока
    3 : Thread3.Priority:=Priority; //приоритет 3 потока
  end;
end;
```

Назначьте описанный выше обработчик события `OnChange()` общим для всех компонентов `TTrackBar`.

Внимание!

По сравнению со всеми своими предшественницами операционная система Windows 7 осуществляет более глубокий контроль за потоками. В частности, ОС может самостоятельно принять решение на увеличение (уменьшение) приоритета потока и, по возможности, осуществляет динамическое перераспределение нагрузки на ядра процессора.

Последним штрихом к нашей программе станет листинг 37.8. Он содержит обработчик события `OnClick()` любого из компонентов `TCheckBox`. Задача события — приостановка/продолжение выполнения потока.

Листинг 37.8. Приостановка/возобновление потока

```
procedure TfrmMain. CheckBox1Click(Sender: TObject);
var Suspended : boolean;
begin
  Suspended:=(Sender as TCheckBox).Checked;
  case (Sender as TCheckBox).Tag of
    1 : Thread1.Suspended:=Suspended;
    2 : Thread2.Suspended:=Suspended;
    3 : Thread3.Suspended:=Suspended;
  end;
end;
```

Сделайте этот обработчик общим для всех элементов `TCheckBox`. Приложение готово, сохраните его и смело запускайте. Результат нашей работы отражает рис. 37.2.

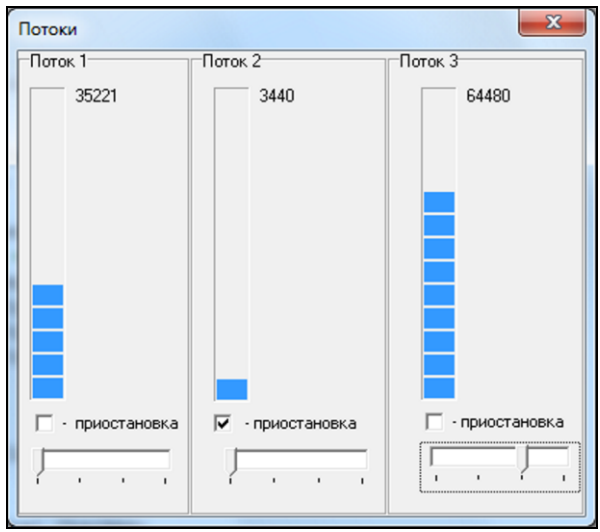


Рис. 37.2. Многопоточное приложение в действии

Синхронизация потоков

Одной из серьезных проблем, с которой сталкиваются разработчики многопоточных приложений, заключается в организации совместного доступа двух (и более) потоков к общему ресурсу. Конкурируя за обладание общим ресурсом, потоки из помощников превращаются

в хулиганов, инициируя неприятные ошибки. Поэтому при реализации взаимодействующих потоков программист обязан решить задачу корректного доступа потоков к разделяемому ресурсу. Для исключения ошибок при подключении к общему ресурсу потокам необходимо синхронизировать свои действия так, чтобы ресурс принадлежал не более чем одному потоку одновременно. При попытке обращения к захваченному ресурсу остальных потоков они должны быть приостановлены до момента его освобождения.

Операционная система Windows — большая мастерица по синхронизации процессов и потоков. Для этого реализовано несколько механизмов управления процессами и потоками, при которых Диспетчер задач не выделяет ожидающим освобождения ресурса ни кванта лишнего времени. Синхронизация производится с помощью объектов синхронизации: событий, критических секций, мьютексов, семафоров и т. д.

Объект синхронизации обладает двумя устойчивыми состояниями: *сигнальным* (signaled) и *несигнальным* (nonsignaled). Сигнальное состояние разрешает приостановленному потоку приступить к работе, несигнальное состояние объекта синхронизации, напротив, предлагает ожидающему своей очереди потоку спать дальше.

> ***Замечание***
>
> Все рассматриваемые в этой главе объекты синхронизации описаны в модуле `SyncObjs`.

Синхронизация событием *TEvent*

К числу наиболее простых средств синхронизации потоков относится синхронизация с помощью описанного в модуле `SyncObjs` события `TEvent`. Как и все объекты синхронизации Windows, событие может находиться в одном из двух состояний: сигнальном и несигнальном. Синхронизируемые потоки способны выяснить, в каком расположении духа находится экземпляр класса `TEvent`, и приостанавливать или, наоборот, возобновлять свою работу.

Как всегда в VCL, создание объекта начинается с вызова его конструктора:

```
constructor Create(EventAttributes: PSecurityAttributes;
              ManualReset, InitialState : Boolean; const Name: string);
```

Первый параметр конструктора `EventAttributes` — это дескриптор безопасности, применяемый в Windows. Параметр `ManualReset` разрешает (значение `true`) переводить событие в несигнальное состояние вручную (это умеет делать метод `ResetEvent()`), иначе сброс произойдет только после старта потока, ожидающего этот объект синхронизации. Параметр `InitialState` определяет, в какое из состояний переходит событие сразу после создания: в сигнальное (`true`) или несигнальное (`false`). Последний параметр назначает имя создаваемому объекту синхронизации. Будьте внимательны, имя должно быть уникальным. Впрочем, имя понадобится, только если этим событием надо воспользоваться извне процесса, создавшего экземпляр `TEvent`.

При обсуждении конструктора события `TEvent` упоминался метод

```
procedure ResetEvent;
```

предназначенный для перевода объекта-события в несигнальный режим "вручную". Обратная операция, включающая событие, обеспечивается процедурой

```
procedure SetEvent;
```

В составе класса `TEvent` предусмотрен метод ожидания:

```
function WaitFor(Timeout: DWORD): TWaitResult;
type TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);
```

Время ожидания перехода объекта синхронизации в сигнальное состояние определяется единственным аргументом метода `Timeout` и измеряется в миллисекундах. Если по истечении заданного тайм-аута ничего не произошло, то метод возвратит `wrTimeout`; если же за это время объект-событие перейдет в сигнальное состояние, то метод, не дожидаясь истечения тайм-аута, вернет `wrSignaled`.

Дескриптор объекта синхронизации `Event` доступен из свойства:

```
property Handle: THandle;
```

Код последней ошибки, связанной с объектом, доступен из свойства:

```
property LastError: Integer;
```

Признаком того, что все в порядке, будет нулевое значение.

Как уже отмечалось в начале подраздела, синхронизация событием очень проста. Создайте новый проект и разместите на его главной форме три компонента: шкалу `ProgressBar1:TProgressBar`, метку `Label1:TLabel` и кнопку `Button1:TButton`. Сохраните проект, переименовав главную форму в `frmMain`, а соответствующий ей модуль в main.pas. Имя проекта выберите на свое усмотрение. Добавьте к проекту новый поток, присвоив ему имя `TProgressThread`. Полный код программного модуля потока вы найдете в листинге 37.9.

Листинг 37.9. Модуль потока `TProgressThread`

```
unit ThreadUnit;
interface

uses Classes, SyncObjs, SysUtils, main;

type TProgressThread = class(TThread)
  private
    procedure ChangeProgress;  //процедура управления шкалой
  protected
    procedure Execute; override;
  end;

var Event:TEvent; //событие синхронизации
implementation

procedure TProgressThread.Execute;
begin
  Event:=TEvent.Create(nil,true,false,'Event1');//создание события
  repeat
    if Event.WaitFor(1000)<>wrSignaled then //ожидание 1000 мс
       begin
         //... другие операции потока, например, загрузка данных ...
         Synchronize(ChangeProgress); //изменение позиции шкалы
       end;
  until Terminated;
  Event.Free;
end;
```

```
procedure TProgressThread.ChangeProgress;
begin
  if frmMain.ProgressBar1.Position+1<frmMain.ProgressBar1.Max then
  begin
    frmMain.ProgressBar1.Position:=frmMain.ProgressBar1.Position+1;
    frmMain.Label1.Caption:=
            Format('%d',[frmMain.ProgressBar1.Position])+'%';
  end else frmMain.ProgressBar1.Position:=0;
end;
end.
```

Просмотрев листинг 37.9, вы увидите, что в рамках программного модуля объявляется событие синхронизации `Event:TEvent`. Создание события осуществляется в методе `Execute()` потока. Визуальной стороной работы потока становится изменение положения шкалы `ProgressBar1`, эта операция производится в методе `ChangeProgress()`.

Теперь переключим наше внимание на главную форму проекта. Нам понадобится описать два события. В событии `OnShow()`, вызываемом в момент вывода формы на экран, создается низкоприоритетный поток. Щелчок по кнопке формы вызывает диалог редактирования позиции шкалы `ProgressBar1`. Для того чтобы в момент ввода нового значения поток приостановил свою работу, мы переводим событие `Event` в сигнальное состояние. Завершив редактирование, сигнальный режим отключается (листинг 37.10).

Листинг 37.10. Раздел реализации программного модуля главной формы

```
implementation
uses ThreadUnit; //ссылка на программный модуль потока
{$R *.dfm}

procedure TfrmMain.FormShow(Sender: TObject);
begin
  with TProgressThread.Create(True) do //создание потока
  begin
    Priority:=tpLowest;
    FreeOnTerminate:=True;
    Start;
  end;
end;

procedure TfrmMain.Button1Click(Sender: TObject);
var Value: string;
begin
  Event.SetEvent; //перевод события Event в сигнальный режим
  Value:=IntToStr(ProgressBar1.Position);

  if InputQuery('Шкала','Новое значение',Value)=true then
    ProgressBar1.Position:=StrToIntDef(Value,ProgressBar1.Position);

  Event.ResetEvent; //отключение сигнального режима
end;
end.
```

Критическая секция *TCriticalSection*

Критическая секция обычно применяется для синхронизации параллельно выполняющихся потоков-близнецов (т. е. потоков, порожденных от одного класса и имеющих один и тот же код) или для защиты разных потоков, обращающихся к одному и тому же коду. Критическая секция — это специальным образом помеченный участок кода, из которого и производится обращение к разделяемому ресурсу. Если код в этой секции выполняется хотя бы одним потоком, то вход в критическую секцию для остальных потоков запрещается.

Создание критической секции начинается с обращения к методу `Create()`. Появившаяся на свет в результате вызова конструктора секция приступит к выполнению своих прямых функциональных обязанностей после обращения к методу

```
procedure Enter;
```

С этого момента объект синхронизации встает на защиту отмеченного участка кода и не допустит к нему ни один из потоков до тех пор, пока не будет вызван метод

```
procedure Leave;
```

извещающий все остальные потоки, что критический участок кода больше не используется.

Критическая секция может ждать своего выполнения бесконечно долго, поэтому при разработке многопоточного приложения с критической секцией программист отвечает за то, чтобы:

- при аварийном завершении потока критическая секция немедленно освобождалась;
- время, затрачиваемое конкурирующими процессами на вход в критическую секцию, было минимальным.

Обычно код входа в критическую секцию оформляется в защищенной от ошибок секции `try..finally`, что гарантирует освобождение заблокированного кода (листинг 37.11).

Листинг 37.11. Использование критической секции

```
var CriticalSection:TCriticalSection;
...
procedure TDemoThread.Execute;
begin
  repeat
    //код потока
    try
      CriticalSection.Enter;
      Synchronize(<вызов внешнего метода>);
    finally
      CriticalSection.Leave;
    end;
  until Terminated;
end;
```

Мьютекс *TMutex*

Как и критическая секция, объект синхронизации *мьютекс* (mutex) может принадлежать только одному потоку. Если мьютекс свободен, то он устанавливается в сигнальное состояние, если же он кем-то захвачен — переходит в несигнальный режим. При попытке полу-

чить доступ к "приватизированному" другим владельцем мьютексу поток получает отказ и переводится в состояние ожидания его освобождения. Таким образом, мьютекс обеспечивает взаимное исключение (mutual exclusion) доступа потоков к разделяемому ресурсу.

> ***Замечание***
>
> Разница между мьютексом и критической секцией заключается в том, что мьютекс может быть доступен не только внутренним потокам приложения, но и другим процессам, протекающим в операционной системе. Единственным условием для этого является наличие у мьютекса уникального имени. Таким образом, мьютекс представляет собой средство синхронизации не только потоков внутри приложения, но и всех процессов операционной системы.

В VCL мьютекс описывается классом `TMutex`. Существуют три варианта создания мьютекса. Простейший из них обеспечивает конструктор

```
constructor Create(UseCOMWait: Boolean = False); overload;
```

В итоге на свет появляется неименованный объект синхронизации, доступный только внутренним потокам одного приложения. Единственный параметр конструктора определяет способность мьютекса взаимодействовать с компонентами STA (single-threaded apartment) COM.

Наиболее востребованной версией конструктора считается

```
constructor Create(MutexAttributes: PSecurityAttributes;
                   InitialOwner: Boolean; const Name: string;
                   UseCOMWait: Boolean = False); overload;
```

Первый параметр метода представляет собой указатель на запись атрибутов защиты `pSecurityAttributes = ^TSecurityAttributes`. Параметр определяет атрибуты безопасности, в простейшем случае в качестве этого аргумента допускается передача неопределенного указателя `nil`. Параметр `InitialOwner` указывает, станет ли поток, создающий мьютекс, его владельцем. Для этого параметру передавайте значение `true`. Последний аргумент содержит имя мьютекса. Если мьютекс не планируется разделять с внешними процессами, то метод удовлетвориться пустым аргументом.

Третий конструктор открывает уже существующий мьютекс, обращаясь к нему по имени `Name`.

```
constructor Create(DesiredAccess: LongWord; InheritHandle: Boolean;
            const Name: string; UseCOMWait: Boolean = False); overload;
```

Порядок доступа зависит от первого параметра, в который направляется комбинация флагов доступа: `MUTEX_ALL_ACCESS`, `MUTEX_MODIFY_STATE`, `SYNCHRONIZE`, `_DELETE`, `READ_CONTROL`, `WRITE_DAC`, `WRITE_OWNER`.

Для управления мьютексом достаточно пары методов:

```
procedure Acquire; override;
procedure Release; override;
```

Первая из процедур захватывает мьютекс, вторая — освобождает.

> ***Внимание!***
>
> Не забывайте освобождать мьютекс, иначе первый поток (захватив мьютекс) станет единственным владельцем критического участка кода и остановит работу оставшихся потоков по крайней мере до момента своего физического уничтожения методом `Terminate()`. В этом случае мьютекс перейдет в разряд покинутых, и операционная система передаст права на него другому потоку.

Семафор *TSemaphore*

Семафор (semaphore) — логическое развитие мьютекса. Как вы помните, объект взаимного исключения мьютекс позволял владеть собой только одному потоку. В отличие от него семафор может находиться в совместном использовании сразу нескольких потоков, но при этом он умеет управлять количеством этих потоков — выступает в роли ограничителя их "популяции". Для этого он снабжен счетчиком, определяющим максимальное число потоков, которым разрешено выполняться совместно. С запуском каждого нового потока значение счетчика уменьшается на единицу, и когда оно достигнет нуля, семафор перейдет в несигнальное состояние. Это означает, что последующие потоки приостанавливаются и будут дожидаться момента, пока какой-нибудь из потоков не освободит семафор. Характерным примером области применения семафоров может послужить программное обеспечение какого-нибудь аппаратного устройства, с которым может работать только ограниченное количество потоков, или реализация Web-сервера (сервера баз данных), допускающего одновременный доступ ограниченного числа посетителей.

Семафор обладает сразу тремя перегружаемыми версиями конструктора:

```
constructor Create(UseCOMWait: Boolean = False); overload;
constructor Create(SemaphoreAttributes: PSecurityAttributes;
            AInitialCount, AMaximumCount: Integer;
            const Name: string; UseCOMWait: Boolean = False); overload;
constructor Create(DesiredAccess: LongWord; InheritHandle: Boolean;
        const Name: string; UseCOMWait: Boolean = False); overload;
```

Первый конструктор наиболее прост, он создает незатейливый семафор, по своим характеристикам почти не отличающийся от мьютекса. Наиболее полезна вторая версия конструктора, именно она формирует полноценный объект синхронизации. Третий конструктор предназначен не для создания, а для получения доступа к семафору (обычно создаваемому в другом приложении). Параметры конструкторов представлены в табл. 37.3.

Таблица 37.3. *Параметры конструктора семафора*

Параметр	Описание
SemaphoreAttributes	Ссылка на структуру TSecurityAttributes с атрибутами безопасности, допускается nil
AInitialCount	Начальное и максимальное значения счетчика семафора. При превышении максимального значения семафор приостанавливает доступ к ресурсу
AMaximumCount	
Name	Чувствительное к регистру символов уникальное имя семафора
UseCOMWait	Параметр устанавливается в true, если блокируемый ресурс должен поддерживать технологию STA COM
DesiredAccess	Уточняет особенности доступа к семафору: SEMAPHORE_ALL_ACCESS, SEMAPHORE_MODIFY_STATE, SYNCHRONIZE, _DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER
InheritHandle	Разрешает дочерним процессам наследовать дескриптор

Захват семафора осуществляется после обращения потока к методу

```
procedure Acquire;
```

Внутренний счетчик семафора получает приращение, и если он превысил ограничение на максимальное число потоков, доступ к ресурсу временно блокируется. Блокировка снимается только после того, как другой поток освободит семафор с помощью метода

```
procedure Release; overload; override;
function Release(AReleaseCount: Integer): Integer; reintroduce; overload;
```

Несмотря на то, что семафор — достаточно сложный объект синхронизации, работа с ним не вызовет особых затруднений. Листинг 37.12 отражает процесс создания экземпляра семафора.

Листинг 37.12. Создание семафора

```
uses SyncObjs,...
var Semaphore:TSemaphore; //глобальная переменная
...
procedure TfrmMain.FormCreate(Sender: TObject);
var AMaximumCount:Integer;
begin
  AMaximumCount:=TThread.ProcessorCount*10;
  Semaphore:=TSemaphore.Create(nil,1,AMaximumCount,'DemoSemaphor');
  ...
```

Обратите внимание на то, как в коде определяется максимальное число потоков, контролируемое семафором. Мы узнаем число процессоров в компьютере и, исходя из этого, ограничиваем количество потоков. Напомню, что не рекомендуется создавать более 16 потоков на одно ядро процессора.

Листинг 37.13 демонстрирует работу с семафором потока.

Листинг 37.13. Применение семафора

```
procedure TDemoThread.Execute;
begin
try
  Semaphore.Acquire;
  //работа потока с ресурсом (данными)
  Semaphore.Release;
end;
```

> ***Внимание!***
>
> Будьте аккуратны при использовании семафоров в зацикленных потоках (потоках, в методе `Execute()` которых используются операторы цикла). В таких случаях высока вероятность блокирования потока.

ГЛАВА 38

Взаимодействие процессов

Уже давно прошло то время, когда в силу своей самодостаточности программе хватало опоры только на операционную систему. Современное приложение должно быть максимально открытым и способным взаимодействовать с другими выполняющимися в системе процессами. Любое взаимодействие нуждается в организации обмена данными. Но каким образом это осуществить? В этой главе мы получим представление о наиболее распространенных способах обмена данными между процессами, выполняющимися на одном компьютере. Мы рассмотрим такие вопросы, как обмен данными при посредничестве буфера обмена, обмен сообщениями и механизм файлового отображения.

Обмен данными через буфер обмена

В Microsoft Windows каждому процессу выделяется собственное виртуальное адресное пространство. Для 32-разрядных процессов его размер составляет 4 Гбайт, для 64-разрядных — 8 Тбайт. Адресное пространство одного процесса наглухо закрыто для другого, поэтому даже самые назойливые процессы не могут напрямую вмешиваться в деятельность своих коллег. Безусловно, один из процессов вправе в своем виртуальном адресном пространстве по какому-то адресу `h12345678` разместить данные, которыми он не прочь поделиться с другими приложениями. Но беда в том, что для других процессов по адресу `h12345678` может находиться все, что угодно, но только не требуемые данные, ведь речь идет об адресе в виртуальном адресном пространстве отдельного процесса, а он не имеет ничего общего с реальным физическим адресом.

Выход из положения заключается в создании области памяти, разделяемой всеми процессами и доступной для осуществления операций чтения/записи. Наиболее показательным примером решения такой проблемы выступает буфер обмена Windows.

Буфер обмена предназначен для передачи различного типа данных между приложениями или внутри одного приложения. Функционально буфер обмена представляет собой некоторый перечень методов Windows API, осуществляющих операции с областью памяти. Содержание операций заключается в действиях по загрузке в память и выгрузки из нее данных различного типа. Для обеспечения максимально удобной работы с буфером обмена в Delphi введен специальный класс `TClipboard`, он описан в модуле `Clipbrd` (который необходимо подключать к проекту в строке `uses`). С присоединением данного модуля программист получит доступ к буферу обмена при посредничестве функции

```
function Clipboard: TClipboard;
```

Внимание!

Для создания экземпляра класса `TClipboard` не требуется обращение к конструкторам, процесс построения буфера обмена инкапсулирован в функции `Clipboard()`.

Буфер обмена служит универсальным хранилищем данных и позволяет размещать в себе обычную текстовую информацию, растровые и векторные рисунки, объекты OLE и т. п. Однако для различных типов данных, направляемых в буфер, требуется уникальный способ их хранения. Этот способ определяется *форматом буфера обмена*. Для ключевых, наиболее часто используемых типов объектов в Windows заранее подготовлены специальные зарегистрированные форматы буфера обмена. Однако если приложение предполагает размещать в буфере какую-то экзотическую информацию, то программист имеет полное право описать и зарегистрировать в системе собственный — *пользовательский формат*.

Для выяснения, какие форматы доступны в буфере обмена компьютера, требуется обратиться к свойству

```
property Formats[Index: Integer]: Word;//только для чтения
```

Свойство позволяет выяснить числовые значения — коды, зарегистрированных на данный момент форматов буфера обмена. Причем общее количество зарегистрированных форматов мы найдем в свойстве

```
property FormatCount: Integer;//только для чтения
```

Расшифровка наиболее распространенных форматов представлена в табл. 38.1.

Таблица 38.1. *Распространенные форматы буфера обмена*

Константа	Значение	В буфере обмена находится
`CF_TEXT`	1	Оканчивающиеся нулем символы в ASCII-кодировке
`CF_BITMAP`	2	Зависящий от устройства битовый образ
`CF_METAFILEPICT`	3	Рисунок — метафайл с дополнительной информацией, описываемый структурой `METAFILEPICT`
`CF_SYLK`	4	Специализированный формат данных Microsoft Symbolic Link (SYLK), применяемый, например, для передачи данных в буфер обмена программой Excel
`CF_DIF`	5	Формат данных Data Interchange Format (DIF)
`CF_TIFF`	6	Изображения формата Tag Image File Format (TIFF)
`CF_OEMTEXT`	7	Текст в кодировке OEM
`CF_DIB`	8	Независящий от устройства битовый образ
`CF_PALETTE`	9	Цветовая палитра
`CF_WAVE`	12	Стандартные аудиоданные в кодировке PCM
`CF_UNICODETEXT`	13	Текст в кодировке UNICODE
`CF_ENHMETAFILE`	14	Расширенный метафайл
`CF_HDROP`	15	Список файлов
`CF_LOCALE`	16	Текстовые данные в соответствии с кодовой страницей
`CF_PRIVATEFIRST`	200	Диапазон значений, зарезервированный для пользовательских (нестандартных) форматов буфера обмена
`CF_PRIVATELAST`	767	

Для проверки нахождения в буфере обмена данных определенного типа применяют метод

```
function HasFormat(Format: Word): Boolean;
```

На роль параметра претендуют константы из табл. 38.1 или соответствующие им числовые значения. Если формат действительно зарегистрирован системе, то метод возвращает `true`.

Просмотрите код примера в листинге 38.1. В нашем распоряжении есть командный объект `acPasteFromClipboard:TAction`, отвечающий за вставку текста из буфера обмена. В рамках события `OnUpdate()` мы проверяем наличие в буфере обмена текстовых данных и активируем (или переводим в пассивный режим) команду.

Листинг 38.1. Проверка наличия в буфере обмена текстовых данных

```
procedure TfrmMain.acPasteFromClipboardUpdate(Sender: TObject);
begin
  acPasteFromClipboard.Enabled:=Clipboard.HasFormat(CF_TEXT);
end;
```

Изначально буфер обмена создавался для переноса символьных строк, именно поэтому наиболее просто работать с текстовой информацией (формат `CF_TEXT`). Для помещения текстовых данных в буфер или наоборот, чтения текста из него, используйте свойство

```
property AsText: string;
```

Для размещения в буфере текстовых данных воспользуйтесь процедурой

```
procedure SetTextBuf(Buffer: PChar);
```

Обратная задача — извлечение текста из буфера обмена — реализуется методом:

```
function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;
```

На содержимое буфера обмена укажет `Buffer`, параметр `BufSize` определяет, сколько символов надо копировать. Функция возвращает реальное количество скопированных символов.

Для отправки в буфер обмена любых других данных используйте процедуру

```
procedure SetAsHandle(Format: Word; Value: THandle);
```

Указатель на копируемый объект передается в параметре `Value`. Параметр `Format` определяет формат буфера обмена.

Для получения указателя на данные, находящиеся в буфере обмена Windows, применяют функцию

```
function GetAsHandle(Format: Word): THandle;
```

Еще один способ копирования данных в буфер обеспечивает метод

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

Обратите внимание на то, что источник должен быть наследником класса `TPersistent`. В листинге 38.2 продемонстрированы способ копирования изображения в буфер обмена и обратная операция получения битового образа из буфера.

Листинг 38.2. Работа с графикой через буфер обмена

```
var BMP:TBitmap;
begin
  BMP:=TBitmap.Create;                  //создаем объект растровой графики
  BMP.LoadFromFile('c:\picture.bmp'); //загружаем в него
                                        //изображение из файла
  Clipboard.Assign(BMP);                //копируем в буфер обмена
  BMP.Destroy;                          //уничтожаем объект BMP
  Image1.Picture.Assign(Clipboard);     //достаем изображение из буфера
end;
```

ЗАМЕЧАНИЕ

Практически все компоненты Delphi, имеющие отношение к редактированию текста (`TEdit`, `TMemo` и т. д.), и классы, предназначенные для отображения графики (`TBitmap`, `TImage` и т. д.), обладают собственными методами работы с буфером обмена.

Для очистки содержимого буфера предназначен метод

```
procedure Clear;
```

Надо отметить, что буфер очищается автоматически перед помещением в него новых данных.

Иногда у приложения возникает необходимость забрать буфер обмена в монопольное использование, т. е. предотвратить возможность изменения содержимого буфера другими программами в отрезок времени, необходимый для выполнения каких-то операций нашим приложением. Для этой цели потребуется пара процедур:

```
procedure Open;
procedure Close;
```

Метод `Open()` захватывает буфер обмена, метод `Close()` — освобождает. Эти процедуры всегда трудятся в паре, и мы должны быть уверены, что в любом случае, даже при возникновении ошибки во время выполнения программы, метод `Close()` будет вызван гарантированно.

Регистрация пользовательского формата буфера обмена

Для профессионального разработчика очень важно уметь создавать собственные форматы буфера обмена. Предположим, что в разрабатываемой программе часто используются структурированные данные, например, записи, содержащие поля: наименование товара, производитель, единица измерения, дата поступления и цена. Зачастую пользователю приходится вводить десяток практически одинаковых записей (отличающихся ценой или датой, или еще чем-нибудь), но для этого ему приходится монотонно набивать одни и те же данные. Не плохо бы в подобной ситуации оказать пользователю помощь и научить нашу программу реализовать методы копирования в буфер и вставки из буфера структурированных данных — записи. Но в Windows не предопределен формат буфера обмена, способный хранить данные такого рода. В подобных ситуациях приходится немного поработать головой.

Создайте новое приложение, состоящее из одной главной формы. При сохранении проекта я традиционно переименую форму в `frmMain` и соответствующий ей программный модуль

в main.pas. Подключите к проекту модуль Clipbrd, объявите запись TDBRecord (позднее она определит новый формат буфера обмена), объявите переменную CF_MyFormat (в ней окажется код нового зарегистрированного формата) и опишите секцию инициализации так, как предложено в листинге 38.3.

Листинг 38.3. Подготовка модуля к работе с новым форматом буфера обмена

```
unit main;
interface

uses Windows, ..., Clipbrd;

  type TDBRecord = packed record //будущий формат буфера обмена
    Name,Creator : string[40];   //товар, производитель
    Measure : string[5];         //единица измерения
    Money : currency;            //цена за единицу
    Date : TDate;                //дата поступления
  end;

type TfrmMain = class(TForm)
  public
  //...
  end;

var frmMain: TfrmMain;
    CF_MyFormat : Word; //код формата после регистрации
implementation
{$R *.dfm}

initialization
  CF_MyFormat:=RegisterClipboardFormat('CF_MyFormat'); //регистрация
end.
```

Первое, что происходит после старта модуля — регистрация нашего пользовательского формата данных. Для этого в секции инициализации вызывается функция Windows API:

```
function RegisterClipboardFormat(szFormat : PChar) : Integer;
```

Функцию регистрации формата интересует единственный параметр, содержащий название нового формата. В случае успешной регистрации метод возвращает число, идентифицирующее новый формат буфера обмена. Полученное значение будет находиться в диапазоне от hC000 до hFFFF. С этого времени формат станет доступным в списке форматов буфера обмена (вспоминайте методы HasFormat() и Formats()). Если несколько позднее любое другое приложение попытается зарегистрировать одноименный формат, то вместо повторной регистрации метод RegisterClipboardFormat() вернет идентификатор, полученный еще в момент первой регистрации. Таким образом, все приложения, знающие имя и структуру формата, получают право обмениваться данными через буфер обмена.

Возвращаемся к нашему примеру. В секции публичных объявлений главной формы опишите заголовки методов копирования данных в буфер обмена и загрузки данных из буфера (листинг 38.4).

Листинг 38.4. Объявление методов работы с буфером обмена

```
  public
   Procedure CopyToClipboard(DBRecord:TDBRecord);
   Function LoadFromClipboard:TDBRecord;
  end;
...
implementation
{$R *.dfm}
procedure TfrmMain.CopyToClipboard(DBRecord: TDBRecord);
var h : THandle;
    FirstByte : Pointer;
begin
  h :=GlobalAlloc(GMEM_MOVEABLE,SizeOf(TDBRecord));
  try
    FirstByte:=GlobalLock(h);
    Move(DBRecord,FirstByte^,SizeOf(TDBRecord));
    Clipboard.SetAsHandle(CF_MyFormat,h);
  finally
    GlobalUnLock(h);
  end;
end;

function TfrmMain.LoadFromClipboard: TDBRecord;
var h : THandle;
    FirstByte : Pointer;
begin
  h :=Clipboard.GetAsHandle(CF_MyFormat);
  if h<>0 then
  begin
    FirstByte:=GlobalLock(h);
    try
      Move(FirstByte^,Result,SizeOf(TDBRecord));
    finally
      GlobalUnLock(h);
    end;
  end;
end;
```

Метод `CopyToClipboard()` несет ответственность за загрузку в память данных записи. Для этих целей функция `GlobalAlloc()` в общей куче распределяет некую область памяти, необходимую для размещения в ней содержимого записи `TDBRecord`. В первом параметре метода (при помощи константы `GMEM_MOVEABLE`) мы указываем, что данная область памяти может быть перемещаемой, а во втором параметре определяем размер этой области в байтах. Функция возвращает дескриптор созданного объекта, а мы, в свою очередь, сохраняем его в переменную `h`. Следующая функция `GlobalLock()`, получив дескриптор только что сформированного объекта `h`, транслирует его в физический адрес этого объекта — адрес его первого байта в области памяти. Помимо этого `GlobalLock()` информирует операционную систему о том, что мы приступили к работе с объектом `h`. Физически это выглядит, как приращe-

ние счетчика ссылок на объект на единицу. Процедура `Move()` перемещает содержимое `DBRecord` в зарезервированную область памяти в байт с адресом `FirstByte^`. Нам остается воспользоваться уже знакомой процедурой `SetAsHandle()` из коллекции методов класса `TClipboard`. В последних строках, в секции `finally` с помощью функции `GlobalUnLock()` мы уменьшаем на единицу счетчик ссылок на эту область памяти, тем самым информируем Windows о том, что приложение прекратило работать с объектом `h`.

Перейдем к методу `LoadFromClipboard()`, извлекающему данные из буфера обмена. С помощью функции `GetAsHandle()` в дескриптор `h` записывается указатель буфера обмена. Если дескриптор не равен нулю (признак того, что буфер не пуст), то узнаем адрес первого байта области памяти буфера и при помощи метода `Move()` копируем данные из буфера в структуру `TDBRecord`.

Для того чтобы проверить работоспособность кода, повторите пользовательский интерфейс приложения для работы с буфером обмена (рис. 38.1). Для этого разместите на главной форме строки ввода (`TEdit`) и кнопки (`TButton`).

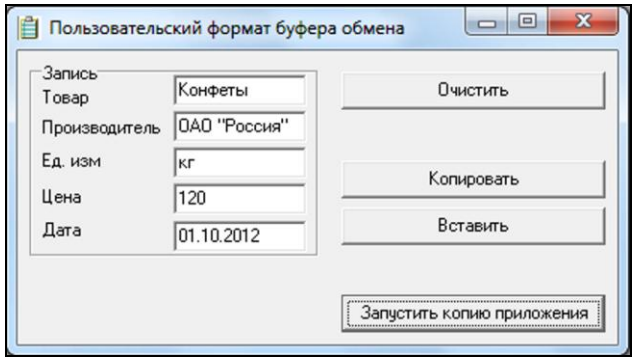


Рис. 38.1. Интерфейс приложения для работы с буфером обмена

Нам понадобятся 5 строк ввода `TEdit` для редактирования полей записи и кнопки для копирования данных в буфер и получения из буфера обмена (листинг 38.5).

Листинг 38.5. Объявление методов работы с буфером обмена

```
procedure TfrmMain.btnCopyToClipboardClick(Sender: TObject);
var DBRecord : TDBRecord;
begin
  DBRecord.Name    :=edName.Text;
  DBRecord.Creator:=edCreator.Text;
  DBRecord.Measure:=edMeasure.Text;
  DBRecord.Money   :=StrToFloatDef(edMoney.Text,0);
  DBRecord.Date    :=mcDate.Date;
  CopyToClipboard(DBRecord);
end;

procedure TfrmMain.btnLoadFromClipboardClick(Sender: TObject);
var DBRecord : TDBRecord;
begin
  DBRecord       :=LoadFromClipboard;
```

```
  edName.Text   :=DBRecord.Name;
  edCreator.Text:=DBRecord.Creator;
  edMeasure.Text:=DBRecord.Measure;
  edMoney.Text  :=FloatToStr(DBRecord.Money);
  edDate.Text   :=DateToStrDef(DBRecord.Date,Now);
end;
```

Для тестирования кода откомпилируйте программу и запустите две копии приложения. Заполните все поля данных в одном из приложений и нажмите кнопку копирования данных в буфер обмена. Переключитесь на другой экземпляр программы. Нажатие кнопки вставки данных приведет к появлению в полях редактора данных из буфера обмена. Таким образом, мы реализовали один из самых распространенных способов обмена данными между приложениями.

Обмен сообщениями

Наиболее востребованным способом взаимодействия процессов в Windows является обмен сообщениями. *Сообщение* представляет собой средство, с помощью которого Windows уведомляет процесс о том, что в системе что-то произошло, например, пользователь щелкнул кнопкой мыши, была нажата клавиша или системный таймер произвел очередной отсчет интервала времени.

Любое предназначенное для работы под управлением Windows приложение обладает хотя бы одним окном, называемым главным окном приложения. Напомню, что это ни в коем случае ни главная форма приложения. Если приложение разработано в среде Delphi, то с его стартом создается невидимое главное окно, и для того чтобы добраться до него, нам потребуется свойство `Handle` объекта `Application`. На протяжение всего периода выполнения приложения оно находится под неусыпным контролем со стороны операционной системы. Для этого с момента создания до момента разрушения главного окна приложения операционная система поддерживает с ним активный диалог. Беседа ведется посредством отправки сообщений оконной процедуре главного окна приложения. Что понимается под словосочетанием "отправить сообщение"? Это значит, что операционная система осуществит запись о каком-то событии в область памяти, доступную оконной процедуре. Указанная область памяти называется *очередью сообщений*. Очередь может хранить несколько сообщений, для этого она устроена по принципу стека "первым пришел — первым вышел" (FIFO, First Input First Output). Программа также вправе отправить сообщение Windows. В этом случае сообщение от программы помещается в другую, общесистемную очередь Windows. Заметьте, что общесистемная очередь существует в одном-единственном экземпляре, и в нее помещаются сообщения от всех окон.

Окно также в состоянии отправлять сообщение другому окну, причем окна могут принадлежать и различным процессам. Именно на умении оконного элемента управления получать и обрабатывать сообщения основывается программирование приложений, управляемых событиями. Если вдруг главное окно приложения разучится "выуживать" отправленную ему информацию из очереди сообщений, можно смело говорить о том, что программа "зависла". Помимо главного окна, приложение включает десятки и даже сотни обычных окон — это оконные элементы управления (потомки `TWinControl`), размещаемые программистом на формах проекта.

Для того чтобы приложение не прозевало предназначенное ей послание, оно периодически просматривает свой "почтовый ящик" на предмет новых поступлений. Для этого в любой

программе (кстати, и в самой Windows) организуется *цикл обработки сообщений* (message loop). И если по какой-то причине у приложения остановится этот цикл, то оно безнадежно "зависнет", а если остановится цикл операционной системы, то нас спасет только перезагрузка.

> ***Замечание***
>
> В библиотеке VCL получать сообщения умеют только оконные элементы управления (элементы управления, основанные на `TWinControl`).

Поиск окна

Для того чтобы сообщение дошло до окна-получателя, мы должны знать его дескриптор. Если же дескриптор нам неизвестен, то можно воспользоваться функцией

```
function FindWindow(ClassName, WindowName : PChar):THandle;
```

Задача функции — среди всех окон процессов попытаться найти именно то, которому предназначено наше послание.

Для поиска главного окна приложения первый параметр устанавливают в `nil`, во второй параметр передается указатель на строку, заканчивающуюся нулем и содержащую название требуемого окна. В случае успеха функция вернет значение, отличное от нуля (листинг 38.6).

Листинг 38.6. Объявление методов работы с буфером обмена

```
var h : hWnd;
begin
  h:=FindWindow(nil,'Project1');  //поиск окна с заголовком Project1
  if h>0 then
  begin
    ...
  end;
end;
```

Существует еще один способ поиска нужного окна. Так, функция

```
function EnumWindows(lpEnumFunc: TFNWndEnumProc; lParam: LPARAM): BOOL;
```

позволяет построить список всех имеющихся в системе главных окон. В первый параметр функции следует передать ссылку на функцию обратного вызова, второй параметр предназначен для отправки в функцию обратного вызова дополнительных данных.

Для исследования функции нужно создать новое приложение и разместить на нем список `ListView1:TListView`. Переключите список в табличный режим представления данных (`ViewStyle=vsReport`) и в редакторе колонок создайте две колонки для хранения дескриптора и заголовка окна. Все остальное вы найдете в листинге 38.7.

Листинг 38.7. Построение списка окон

```
//функция обратного вызова
function EnumWindowsProc(wnd: HWND; Param: LPARAM): boolean; stdcall;
var buf: array[0..99] of char;
    s:string;
```

```
begin
  frmMain.ListView1.Items.BeginUpdate;
  GetWindowText(Wnd,Buf,High(buf)); //текст в заголовке окна
  s:='h'+IntToHex(wnd,8);
  with frmMain.ListView1.Items.Add do
  begin
    Caption:=s;
    Subitems.Add(StrPas(buf));
    Data:=@Wnd;
  end;
  frmMain.ListView1.Items.EndUpdate;
end;

procedure TfrmMain.FormShow(Sender: TObject);
var P:Pointer;
begin
  ListView1.Clear;
  P:=@EnumWindowsProc; //указатель на функцию обратного вызова
  EnumWindows(@EnumWindowsProc,0);//построение списка окон
end;
```

В момент вывода окна на экран осуществляется обращение к функции `EnumWindows()`, которая, в свою очередь, обращается за помощью к функции обратного вызова `EnumWindowsProc()`. Обратите внимание на то, что при описании функции обратного вызова мы воспользовались стандартным соглашением о вызовах `stdcall`, в противном случае код утратит работоспособность. В результате выполнения кода вы получите перечень из нескольких сотен окон, о существовании которых вы до этого момента даже и не подозревали (рис. 38.2).

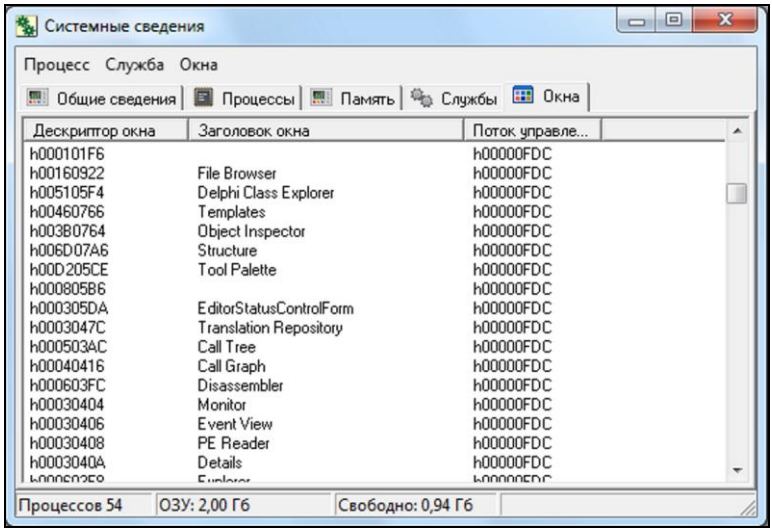


Рис. 38.2. Построение перечня окон с помощью функции `EnumWindows()`

Регистрация пользовательских сообщений

В Windows зарегистрирован впечатляющий перечень сообщений, отправляемых для активизации окна, минимизации или распахивания его на весь экран. Сообщения появляются на свет при нажатии и отпускании клавиш, при перемещении мыши или щелчке ее кнопки... Все сообщения описываются именованными константами, имена констант, как правило, начинаются с аббревиатуры "WM_" (сокращение от Windows Message). Обязательно обратитесь к системе помощи для программистов MSDN (**http://msdn.microsoft.com**) и просмотрите обширный перечень зарегистрированных сообщений. Более того, ряд элементов управления способен воспринимать специфичные сообщения. Например, сообщения для комбинированного списка TComboBox начинаются с "CB_", для дерева TTreeView — "TVM_", для кнопки — "BN_".

Операционная система допускает создание нестандартных сообщений — сообщений, определяемых программистами. Пользовательские сообщения могут применяться с целью организации взаимодействия между несколькими программами или в рамках одного приложения. Для регистрации своего сообщения применяется функция.

```
function RegisterWindowMessage(S: PChar) : integer;
```

Единственный параметр метода рассчитывает получить имя сообщения. Функция зарегистрирует пользовательское сообщение и возвратит целое число — идентификатор сообщения. Если окажется, что сообщение с таким именем уже было заявлено ранее (например, другим приложением), то функция вернет идентификатор, зарегистрированный другим приложением.

В адрес окна могут поступать сообщения от самых различных отправителей, как находящихся внутри этого же приложения, так и от внешних источников (других приложений и операционной системы). При поступлении сообщений к главному окну приложения у Application возникает событие

```
property OnMessage: TMessageEvent;
type TMessageEvent = procedure (var Msg: TMsg;
                                var Handled: Boolean) of object;
```

Содержимое сообщения передается в структуре Msg:TMsg (листинг 38.8).

Листинг 38.8. Объявление структуры TMsg

```
type TMsg = packed record
    hwnd: HWND;      //дескриптор получателя сообщения
    message: UINT;   //зарегистрированный идентификатор сообщения
    wParam: WPARAM; //необязательный старший параметр
    lParam: LPARAM; //необязательный младший параметр
    time: DWORD;     //время создания сообщения
    pt: TPoint;      //экранные координаты указателя мыши
  end;
```

Если в параметре Handled программист возвращает операционной системе значение true, то это означает, что сообщение обработано полностью. В свою очередь, значение false станет сигналом Windows о том, что при остром желании система может продолжить обрабатывать сообщение.

Для получения доступа к обработчику события `OnMessage()` можно пойти двумя путями. Первый из них — воспользоваться услугами компонента `TApplicationEvents`. Упомянутый компонент инкапсулирует все события приложения. Второй путь требует написания нескольких дополнительных строк кода, но освобождает программиста от необходимости "захламлять" свой проект излишними компонентами.

Пример обмена сообщениями между процессами

Спроектируем два небольших приложения, демонстрирующих на практике процесс обмена сообщениями. Приложение-передатчик станет отправлять сообщение со своими экранными координатами в адрес приложения-приемника. Приложение-приемник (получив координаты) укажет пользователю направление на приложение-передатчик.

Начнем работу с приложения-передатчика. Создайте новый проект, на единственной форме проекта разместите компонент `ApplicationEvents1: TApplicationEvents`. Уменьшите размер формы до минимальных размеров. Сохраните проект под любым именем.

Чтобы научить приложение отправлять сообщения, понадобится минимум усилий, весь исходный код главной формы проекта предложен в листинге 38.9.

Листинг 38.9. Исходный код приложения-передатчика

```
unit transmit;

interface
uses Windows, Messages, SysUtils, Classes, Graphics, Forms, StdCtrls,
     AppEvnts, Controls;

type TfrmTransmit = class(TForm)
    ApplicationEvents1: TApplicationEvents;
    procedure ApplicationEvents1Idle(Sender: TObject; var Done: Boolean);
  end;
var
  frmTransmit: TfrmTransmit;
  WM_MSG : Cardinal; //здесь будем хранить идентификатор сообщения

implementation
{$R *.dfm}

procedure TfrmTransmit.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
var h:hWnd;
    FormCenter:TPoint;
begin
 h:=FindWindow(nil,'RECEIVER'); {поиск главного окна с названием RECEIVER}
 if h>0 then // если окно обнаружено
 begin
   FormCenter:=Point(frmTransmit.ClientWidth div 2,
                     frmTransmit.ClientHeight div 2);
```

```
   Windows.ClientToScreen(frmTransmit.Handle,FormCenter);
   PostMessage(h, WM_MSG, FormCenter.X, FormCenter.Y);
 end;
end;

initialization {регистрация в Windows пользовательского сообщения}
   WM_MSG:=RegisterWindowMessage('WM_SENDPOSITION');
end.
```

В момент инициализации приложение регистрирует пользовательское сообщение "WM_SENDPOSITION" и результаты регистрации помещает в переменную WM_MSG. После этого в момент возникновения события OnIdle (событие будет генерироваться регулярно в момент перехода приложения-передатчика в состояние простоя, или, попросту говоря, "безделья") мы узнаем экранные координаты центра приложения-передатчика и отправляем их приемнику.

Обратите внимание, что поиск приемника осуществляет уже знакомая нам функция FindWindow(). Функция старается обнаружить окно с заголовком "RECEIVER". Это означает, что при проектировании приемника нам надо не забыть присвоить это имя заголовку главной формы.

Для отправки сообщения в адрес приложения-приемника используется функция

```
function PostMessage(
       Wnd : HWND;      //дескриптор окна-получателя
       Msg : UINT;      //отправляемое сообщение
       wParam : WPARAM; //дополнительный старший параметр сообщения
       lParam : LPARAM  //дополнительный младший параметр сообщения
   ) : boolean;
```

У функции PostMessage() есть коллега с идентичным набором параметров:

```
function SendMessage(
         Wnd : HWND;         //дескриптор окна-получателя
         Msg : UINT;         //отправляемое сообщение
         wParam : WPARAM;    //дополнительный старший параметр сообщения
         lParam : LPARAM     //дополнительный младший параметр сообщения
     ) : LRESULT;
```

Однако, несмотря на внешнюю схожесть, поведение функций существенно отличается друг от друга. Функция SendMessage() работает в синхронном режиме, она отправляет сообщение непосредственно процедуре окна и ожидает окончания его обработки. В отличие от нее асинхронная функция PostMessage() стандартным образом помещает сообщение в очередь Windows и сразу возвращает управление вызвавшей ее программе, не дожидаясь результатов обработки сообщения.

Синхронная функция SendMessage() относится к разряду потенциально опасных, т. к. если окно-получатель по каким-то причинам не сможет ответить на сообщение, то вызвавшее функцию приложение может даже потерять работоспособность. Поэтому вместо SendMessage() зачастую используют SendMessageTimeout(). Указанная функция ожидает ответ в течение установленного тайм-аута, и, если за этот промежуток времени ответа получено не было, она гарантированно возвратит управление своей программе.

Исходя из разницы в работе функций, различаются и возвращаемые ими значения. SendMessage() возвращает значение, полученное в результате обработки сообщения;

`PostMessage()` всего лишь информирует, удалось ли ей поставить сообщение в очередь (значение `true`) или нет (`false`).

> ***ЗАМЕЧАНИЕ***
>
> Для организации широковещательной рассылки в первый параметр функций `SendMessage()` и `PostMessage()` следует передать константу `HWND_BROADCAST`.

Переходим ко второй части нашей работы — разработке приложения-приемника. Создаем новое приложение и сразу же в свойство `Caption` главной формы проекта заносим заголовок окна "`RECEIVER`". Разместите на форме единственный компонент — метку `Label1:TLabel` — и сохраните проект под любым именем. Нам осталось разобраться с исходным кодом (листинг 38.10).

Листинг 38.10. Исходный код приложения-приемника

```
var frmMain: TfrmMain;
  TransmiterPos:TPoint; //переменная для получения координат передатчика
  WM_MSG : Cardinal;    //идентификатор сообщения
implementation
{$R *.dfm}

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  Application.OnMessage:=OnMessage_Event;  //связь события с методом
  Label1.Caption:='';
end;

procedure TfrmMain.OnMessage_Event(var Msg: TMsg; var Handled: Boolean);
begin //обработка поступающих сообщений
  if Msg.message=WM_MSG then
  begin
    Label1.Caption:=Format('Координаты (%d,%d)',[Msg.wParam,Msg.lParam]);
    TransmiterPos.X:=Msg.wParam;
    TransmiterPos.Y:=Msg.lParam;
    Repaint;
  end;
end;

procedure TfrmMain.FormPaint(Sender: TObject);
begin //рисуем стрелку, указывающую на передатчик
  with frmMain.Canvas do
  begin
    FillRect(frmMain.ClientRect);
    MoveTo(frmMain.ClientWidth div 2, frmMain.ClientHeight div 2);
    Windows.ScreenToClient(frmMain.Handle, TransmiterPos);
    LineTo(TransmiterPos.X, TransmiterPos.Y);
  end;
end;

initialization //инициализация
   WM_MSG:=RegisterWindowMessage('WM_SENDPOSITION');
end.
```

Для работы приемника понадобятся две глобальные переменные. В переменную `TransmiterPos` мы станем записывать поступающие от приложения-передатчика экранные координаты, в `WM_MSG` после инициализации приложения окажется числовой идентификатор сообщения "`WM_SENDPOSITION`". Обработка поступающих сообщений производится в рамках метода `OnMessage_Event()`. На этот раз мы отказались от применения в проекте компонента `TApplicationEvents` и обеспечили реакцию на событие "вручную".

Откомпилируйте и запустите оба приложения. На рис. 38.3 представлен экранный снимок разработанных приложений, благодаря механизму обмена сообщениями приемник владеет информацией о местонахождении главного окна передатчика и проводит линию в его направлении.

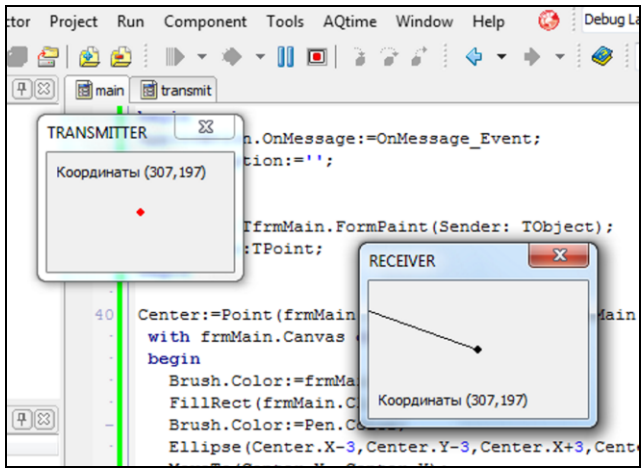


Рис. 38.3. Экранный снимок работы приложений передатчика и приемника

Примечание

Для обмена данными между процессами часто применяют специальное сообщение `WM_COPYDATA`. Перед отправкой сообщения программист заполняет особую структуру `COPYDATASTRUCT`. Ссылка на нее передается через параметр `lParam` сообщения. Через параметр `wParam` необходимо передать идентификатор окна, посылающего сообщение.

Файловое отображение

Каждому выполняющемуся под управлением Windows процессу выделяется индивидуальное адресное пространство. Таким образом, не имея прямой возможности изменять данные в чужом адресном пространстве, приложения защищаются от взаимных посягательств. Но это достоинство превращается в недостаток, когда нескольким процессам требуется манипулировать данными, принадлежащим адресному пространству одного из приложений. Выход из подобной ситуации очевиден — надо научиться отображать некоторый объем совместно используемой памяти в адресное пространство взаимодействующих приложений.

Одним из способов, позволяющих организовать работу приложения с разделяемыми данными, является применение отображаемых в память файлов. Для создания объекта файлового отображения используют функцию

```
Function CreateFileMapping(hFile : Cardinal; SA: pSecurityAttributes;
              Protect : Cardinal; dwMaxSizeHigh, dwMaxSizeLow : Cardinal;
              pName : pAnsiChar) : THandle;
```

Перед попыткой создать именованный объект отображения функция обязательно проверит систему на наличие в ней объекта с именем `pName`. Если объект отображения с таким именем уже существует (был сформирован ранее из этого или стороннего процесса), то вместо создания его дубликата функция возвратит дескриптор уже существующего объекта.

Первый параметр метода определяет дескриптор файла, на основе которого создается объект отображения, в нашем случае этому параметру присваивается константа `INVALID_FILE_SIZE = DWORD($FFFFFFFF)`. Это указание создать отображение на основе файла подкачки. Второй параметр содержит указатель на структуру безопасности `TSecurityAttributes`. Если мы не намерены накладывать ограничения на доступ к объекту отображения, то вместо указателя на структуру атрибутов безопасности передаем неопределенный указатель `nil`. Разрешения на операции чтения/записи определяются в параметре `Protect`. Это может быть одна из трех констант: `PAGE_READONLY` (доступ только для чтения), `PAGE_READWRITE` (доступ для чтения и записи) и `PAGE_WRITECOPY` (в спроецированную область разрешено копирование и запись). Два очередных параметра определяют размер проецируемого файла, `dwMaxSizeHigh` — старшие 32 бита, `dwMaxSizeLow` — младшие 32 бита. Благодаря этой паре параметров мы сможем формировать 64-битный адрес. Если размер файла меньше 4 Гбайт, то в `dwMaxSizeHigh` передается нулевое значение. Последний параметр `pName` — указатель на строку с именем файла отображения. В случае успешного выполнения метод возвращает дескриптор объекта отображения файла, иначе функция возвратит неопределенный указатель `nil`.

Распределенная посредством вызова `CreateFileMapping()` память может быть разделенной между несколькими процессами потому, что фактически она располагается вне адресного пространства процессов. Разделяемый участок в файле подкачки переносится в ОЗУ и может стать доступным для всех процессов, знающих имя отображаемого файла.

Функция `CreateFileMapping()` только создала файловый объект. Для отображения файла в адресное пространство нам понадобится помощь функции

```
function MapViewOfFile(hMapObject : THandle;
        Access, OffsetHigh, OffsetLow, Map : Cardinal) : pointer;
```

Здесь параметр `hMapObject` — возвращенный методом `CreateFileMapping()` дескриптор объекта отображения. Параметр `Access` определяет тип доступа к представлению. Различают три основных режима:

- `FILE_MAP_WRITE` или `FILE_MAP_ALL_ACCESS` — доступ для чтения и записи;
- `FILE_MAP_READ` — доступ для чтения;
- `FILE_MAP_COPY` — доступ для копирования.

Параметры `OffsetHigh` и `OffsetLow` соответственно задают старшие и младшие 32 бита смещения внутри файла отображения. `Map` — количество байтов файла, которые должны быть отображены; если параметр установлен в 0, то отображается весь файл. В случае успешного завершения метод возвратит указатель на самый первый байт спроецированного файла (базовый адрес объекта отображения).

Для отмены отображения представления файла из адресного пространства процесса предназначен метод

```
function UnMapViewOfFile(BaseAddress : pointer) : Boolean;
```

Единственный параметр функции — базовый адрес объекта отображения (результат работы метода `MapViewOfFile`).

Для практической реализации механизма файлового отображения нам потребуется создать новый проект и разместить на нем единственный компонент — группу `RadioGroup1:TRadioGroup`. Все остальное вы обнаружите в листинге 38.11.

Листинг 38.11. Демонстрация механизма файлового отображения

```
unit main;

interface

uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, StdCtrls, ExtCtrls;

type TfrmMain = class(TForm)
     RadioGroup1: TRadioGroup;
     procedure FormClose(Sender: TObject; var Action: TCloseAction);
     procedure FormCreate(Sender: TObject);
     procedure RadioGroup1Click(Sender: TObject);
  private
    procedure OnMessage_Event(var Msg: TMsg; var Handled: Boolean);
  end;

var frmMain: TfrmMain;
    hMFile  : THandle;
    pMFile  : pointer;
    WM_MSG  : DWORD;
const mFileName='FileMappingDemo';
implementation

{$R *.dfm}
procedure TfrmMain.FormCreate(Sender: TObject);//создание формы
var Ch:byte;
begin
  //заполняем группу переключателей
  for Ch:=Byte('A') to Byte('Z') do RadioGroup1.Items.Add(Char(Ch));
  Application.OnMessage:=OnMessage_Event;  //связь события с методом
end;

procedure TfrmMain.OnMessage_Event(var Msg: TMsg; var Handled: Boolean);
var ind : string;
begin
  if Msg.message=WM_MSG then //обрабатываем только свое сообщение
  begin
    ind:=PChar(pMFile); //узнаем индекс
    RadioGroup1.ItemIndex:=StrToIntDef(ind,0); //включаем кнопку
  end;
end;
```

```
procedure TfrmMain.RadioGroup1Click(Sender: TObject);
var ind:string;
begin
  ind:=IntToStr(RadioGroup1.ItemIndex);     //преобразуем индекс в символ
  StrCopy(pMFile,PChar(ind));               //копируем индекс в объект
  PostMessage(HWND_BROADCAST,WM_MSG, 0,0); //широковещательная рассылка
end;

procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  UnMapViewOfFile(pMFile); //отключаемся от объекта файлового отображения
  CloseHandle(hMFile);     //закрываем дескриптор
end;

initialization
  hmFile:=CreateFileMapping($FFFFFFFF,nil,PAGE_READWRITE,
                                        0,4,'FILEMAPPINGDEMO');
  pMFile:=MapViewOfFile(hMFile,FILE_MAP_ALL_ACCESS,0,0,0);
  WM_MSG:=RegisterWindowMessage('WM_FILEMAPPING');//регистрация сообщения
end.
```

Объект файлового отображения создается в момент инициализации проекта. Кроме того, в разделе `initialization` мы регистрируем пользовательское сообщение, которое будет отправляться всем другим процессам в тот момент, как наше приложение осуществит запись в файловый объект.

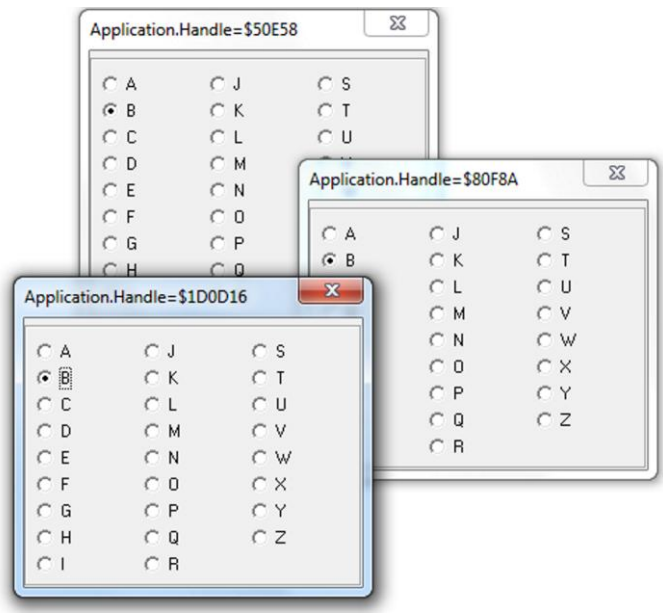


Рис. 38.4. Обмен данными между приложениями с помощью файлового отображения

В момент создания формы мы заполняем группу `RadioGroup1` переключателями, в заголовке которых окажутся символы латинского алфавита. После этого указываем приложению, что в момент поступления в его адрес сообщения следует выполнять метод `OnMessage_Event()`.

Запись данных в объект файлового отображения осуществляется при выборе пользователем того или иного переключателя в группе `RadioGroup1`. Индекс отмеченного переключателя передается в файл, и сразу осуществляется широковещательная рассылка сообщения `WM_FILEMAPPING` всем окнам.

Проект завершен. Запустите несколько экземпляров нашего приложения и проверьте работоспособность кода (см. рис. 38.4). Щелчок по любому из переключателей должен приводить к синхронному выбору этого же компонента в других приложениях.

ГЛАВА 39

Сетевое взаимодействие

В состав системы Microsoft Windows интегрирован обширный перечень сетевых технологий различного назначения и уровней сложности, позволяющих создавать сетевое программное обеспечение. В самом общем случае задача сетевого программного обеспечения заключается в обмене данными между двумя и более приложениями, выполняющимися на разных сетевых станциях. Обычно взаимодействующие в сети процессы подразделяются на серверные и клиентские. Клиент просит сервер оказать ему услугу, отправляя ему какой-то запрос. В свою очередь сервер, в силу своих возможностей, выполняет поручение. Никто не запрещает при разработке своего программного обеспечения одновременно наделять приложение полномочиями как сервера, так и клиента.

В этой главе мы рассмотрим несколько сетевых технологий. Свою работу начнем с наиболее простого *механизма почтовых слотов* (mailslots). Используя названную технологию, приложение получает право отправить сообщение в какой-то почтовый слот, в произвольный момент времени владелец этого слота может письмо "прочитать". Упомянутый механизм допускает отправку сообщения как одному компьютеру, так и рассылку почты нескольким компьютерам одновременно, главное, чтобы они были объединены в единый домен. Еще один механизм, с которым мы познакомимся в этой главе, называется *каналом* (pipe). Взаимодействие между процессами с использованием именованных каналов широко применяется в клиент-серверных технологиях. Кроме того, в настоящей главе большое внимание уделяется организации обмена данными с помощью интерфейса *сокетов* (технология WinSock).

Взглянув рис. 39.1, вы сразу поймете, что почтовые слоты и каналы являются высокоуровневыми технологиями, т. к. они расположились на стыке прикладного и представительного уровней 7-уровневой модели OSI (Open System Interconnection). Сокеты Windows — относительно низкоуровневый сервис, он предоставляет программисту более гибкие возможности по разработке программного обеспечения, одновременно требуя больше знаний и затрат времени на отладку приложения.

Почтовые слоты

Своим названием технология почтовых слотов обязана схожестью принципа работы с обычным почтовым ящиком. Серверное приложение имеет возможность создать собственный почтовый ящик, корреспонденцию в который имеет возможность отправить любой клиент (при условии обладания адресом слота).

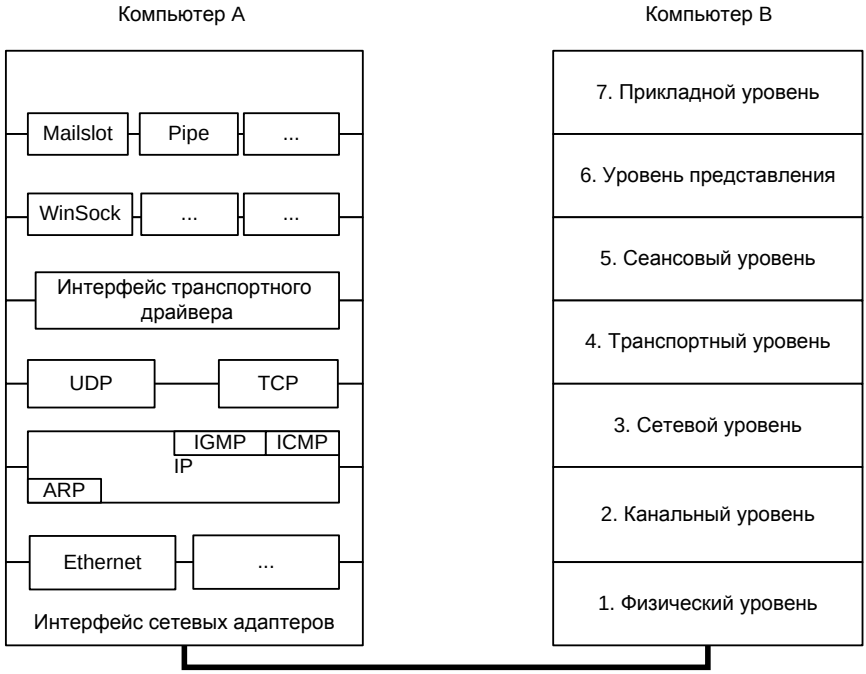


Рис. 39.1. Место сетевых технологий в модели Open System Interconnection

Механизм mailslots односторонний. Передаваемое в виде дейтаграммы сообщение, попав (или не попав) в предназначенный для него почтовый слот, не уведомляет отправителя, что оно прибыло по назначению. Так что о гарантированной доставке почты речи идти не может.

Почтовые слоты часто используются для широковещательной рассылки небольших сообщений. Надо знать, что размер почтового отправления ограничен — он не должен превышать 424 байта.

Организация работы с почтовым слотом сравнительно проста. Сервер регистрирует файл-слот и размещает его в памяти компьютера. Правом читать данные из слота также обладает только сервер, причем для доступа к слоту применяются методы, весьма напоминающие процесс чтения из обычного файла. В свою очередь клиент имеет право лишь отправлять сообщения в почтовый слот.

Определение имени почтового слота

Во время регистрации почтового слота ему присваивается уникальный сетевой адрес. Существует правило, согласно которому склеивается строка с текстом адреса:

- адрес начинается с двух наклонных черт, затем идет точка, вновь наклонная черта и ключевое слово "mailslot";
- необязательный путь, также начинающийся с наклонной черты;
- адрес завершается наклонной чертой и именем слота.

Например:

```
const MailSlotName='\\.\mailslot\sample_slot';
```

При обращении к почтовому ящику другого компьютера алгоритм создания адресной строки незначительно изменяется. В этом случае строка также начинается с двух наклонных черт, но теперь вместо точки указывается сетевое имя компьютера, к которому отправляется корреспонденция или имя домена, если мы одновременно обращаемся к целой группе компьютеров:

```
\\имя_компьютера\mailslot\[путь]имя_слота
\\имя_домена\mailslot\[путь]имя_слота
```

Управление почтовым слотом

За создание почтового ящика отвечает приложение-сервер, оно обращается к функции прикладного программного интерфейса Windows:

```
function CreateMailslot(MailName : PWideChar;
                        MaxMessageSize, ReadTimeout : Cardinal;
                        SecurityAttr : pSecurityAttributes) : THandle;
```

Здесь `MailName` — указатель на строку с адресом слота. `MaxMessageSize` — ограничитель максимальной длины послания в байтах, если параметру присвоить нулевое значение, то ограничения снимаются. Очередной параметр `ReadTimeout` оказывает влияние на процесс чтения корреспонденции из создаваемого ящика. Он задает время ожидания в миллисекундах. Если передается 0, то при обращении к пустому ящику функция, читающая корреспонденцию, возвратит результат немедленно. В противном случае она выждет установленный тайм-аут в надежде, что за этот промежуток времени нам кто-нибудь напишет. Для вечного ожидания передайте константу `MAILSLOT_WAIT_FOREVER`. Последний параметр `SecurityAttr` — указатель на структуру атрибутов безопасности, в простейшем случае в него передают `nil`. В случае успешного выполнения метод возвращает дескриптор созданного слота, иначе — нулевое значение.

Кроме функции создания почтового слота, Windows API способен похвастаться еще двумя функциями работы с mailslot. Во-первых, это функция, позволяющая настраивать время тайм-аута `ReadTimeOut` в слоте, заданном дескриптором `MailSlot:THandle`:

```
function SetMailslotInfo(MailSlot : THandle;
                         ReadTimeOut : Cardinal) : boolean;
```

Во-вторых, это функция, информирующая о состоянии почтового слота:

```
function GetMailslotInfo(MailSlot : THandle; MaxMessageSize : pointer,
  var NextSize : cardinal; MessageCount, ReadTimeout : Pointer) : boolean;
```

Эта функция вновь требует указать дескриптор интересующего нас слота в параметре `MailSlot`. А в результате метод выдаст данные о: `MaxMessageSize` — максимальном размере сообщения в байтах; `NextSize` — размере следующего сообщения, и если такового нет, то в параметре окажется константа `MAILSLOT_NO_MESSAGE`; `MessageCount` — общем числе сообщений в слоте; `ReadTimeout` — выбранном времени тайм-аута почтового ящика.

Получение и отправка корреспонденции

Чтение находящихся в почтовом слоте данных возможно только из программы-сервера, создавшей этот самый слот. Зная дескриптор слота, приложение обращается к нему, как к обыкновенному файлу. Для операции чтения подходят как методы Windows API `ReadFile()` и `ReadFileEx()`, так и их наследник — метод работы с файлами Delphi `FileRead()`.

Для отправки почты клиентское приложение просто осуществляет запись в виртуальный файл, олицетворяющий почтовый слот сервера-получателя. Обычно для этого применяют методы `FileCreate()`, `FileWrite()` и `FileClose()`.

Пример почтового приложения

Приступим к разработке приложения, поддерживающего технологию почтовых слотов. Для этого потребуется создать новый проект, на главной форме которого надо разместить следующие элементы управления:

- строку ввода `cbComputers`, предназначенную для ввода сетевого имени компьютера, в адрес которого отправится почтовая корреспонденция;
- многострочный редактор `MemoOut:TMemo` позволит пользователю вводить текст сообщения, предназначенного для отправки;
- многострочный редактор `MemoIn:TMemo` раскроет входящие сообщения;
- менеджер команд `ActionManager:TActionManager` станет центром управления приложением.

Сохраните проект, предварительно переименовав главную форму в `frmMain`, название проекта выберите на свое усмотрение. Программирование начнем с регистрации почтового слота. Для этого объявите константу `MailSlotName`, в которую мы внесем имя почтового слота, и обратитесь к событию `OnCreate()`, вызываемому в момент создания главной формы (листинг 39.1).

Листинг 39.1. Создание приложения и регистрация слота

```
const MailSlotName='\\.\mailslot\sample_slot';
implementation

{$R *.dfm}
procedure TfrmMain.FormCreate(Sender: TObject);
var ErrorCod : Cardinal;
    s : string;
    buf : array[0..MAX_COMPUTERNAME_LENGTH] of char;
    w : cardinal;
begin
  MailSlot:=CreateMailSlot(MailSlotName, //создание слота
                           0,
                           MAILSLOT_WAIT_FOREVER,
                           nil);
  if MailSlot=INVALID_HANDLE_VALUE then //вызов метода завершился неудачей
  begin
    ErrorCod:=GetLastError();       //узнаем код ошибки
    s:=SysErrorMessage(ErrorCod);   //расшифровка ошибки
    MessageBox(frmMain.Handle,
               PChar(Format('Ошибка %d'+#13+'%s',[ErrorCod, s])),
               PChar('Ошибка'), MB_ICONERROR+MB_OK);
  end;
  MemoOut.MaxLength:=244; //ограничим длину вводимого текста
end;
```

После создания жизненный цикл почтового слота будет продолжаться до тех пор, пока мы не обратимся к функции `CloseHandle()`, освобождающей экземпляр слота. Эту операцию мы проведем в момент закрытия главной формы (листинг 39.2).

Листинг 39.2. Завершение приложения

```
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  CloseHandle(MailSlot);
end;
```

Разработаем метод чтения почты из слота. Для этого в секции частных объявлений опишем заголовок процедуры `ReadMail()`, обладающий единственным параметром `aMailSlot`, в который следует передавать дескриптор почтового ящика, откуда будет прочитано почтовое отправление. Реализация процедуры представлена в листинге 39.3.

Листинг 39.3. Получение почты

```
procedure TfrmMain.ReadMail(aMailSlot : THandle);
var NextSize, MessageCount : cardinal;
    Buf: PAnsiChar;
begin
  //узнаем о содержимом почтового слота
  GetMailslotInfo(aMailSlot, nil, NextSize, @MessageCount, nil);
  //если ящик не пуст, то просматриваем его
  while MessageCount>0 do //собираем все сообщения из ящика
  begin
    try
      Buf:=PAnsiChar(AllocMem(NextSize+1));  //распределяем область памяти
      FileRead(aMailSlot,Buf^,NextSize + 1); //читаем данные из ящика
      MemoIn.lines.add( {дата-время отправки сообщения}
      DateTimeToStr(FileDateToDateTime(FileGetDate(aMailSlot))));
      MemoIn.lines.add(StrPas(Buf));// тест передаем в memo-поле
    finally
      SysFreeMem(Buf);  //освобождаем память
    end;
    Dec(MessageCount); //уменьшаем счетчик сообщений
  end;
end;
```

Обратите внимание на то, что при чтении корреспонденции мы предполагаем, что текст представлен в ANSI-кодировке. Это соглашение следует учесть и при разработке метода, отправляющего сообщение.

> ***Замечание***
>
> Кроме чтения содержимого сообщения, мы также имеем право узнать время его отправления. Помощь в этом окажет метод `FileGetDate()`.

Технология почтовых слотов не предусматривает возможности автоматического контроля за получением корреспонденции. Поэтому, для того чтобы организовать динамический просмотр почтового ящика, в приложении-сервере придется применить таймер или поток.

В секции частных объявлений опишите очередной метод SendMail(). Новый метод возьмет ответственность за отправку текстового сообщения MailText корреспонденту с адресом MailAddres. Реализация метода предложена в листинге 39.4.

Листинг 39.4. Отправка почты

```
procedure TfrmMain.SendMail(const MailAddres, MailText: string);
var  hFile: THandle;
     Buf : PAnsiChar;
begin
  hFile:=FileCreate(MailAddres);//обращаемся к слоту как к файлу
  if hFile>0 then //в случае получения дескриптора
  begin
    try
      Buf:=AllocMem(Length(MailText)+1); //распределяем память для буфера
      StrPCopy(Buf,MailText); //помещаем в буфер сообщение для передачи
      FileWrite(hFile,Buf^,Length(MailText)+1); //отправляем сообщение
    finally
      FileClose(hFile); //закрываем файл
      SysFreeMem(Buf);  //освобождаем буфер
    end;
  end;
end;
```

Приложение уже почти готово. Нам осталось дважды щелкнуть по менеджеру команд ActionManager1 и создать в нем команды acSendMail и acReadMail, отвечающие соответственно за отправку и получение почты. Листинг 39.5 раскрывает функциональную нагрузку этих команд.

Листинг 39.5. Управление приложением

```
procedure TfrmMain.acSendMailExecute(Sender: TObject);//отправка почты
var MailAddress:string;
begin
  {назначаем адрес компьютера-получателя
   по правилу \\имя_компьютера\mailslot\[путь]имя_слота}
  MailAddress:=cbComputers.Text+
                      Copy(MailSlotName,4,Length(MailSlotName)-3);
  SendMail(MailAddress, MemoOut.Text);
  MemoOut.Clear;
end;

procedure TfrmMain.acReadMailExecute(Sender: TObject);//получение почты
begin
  ReadMail(MailSlot);
end;
```

Программирование завершено. Подключите команды к элементам управления (например, кнопкам). После этого разверните приложение на паре подключенных в сеть компьютеров и протестируйте его работоспособность (рис. 39.2).

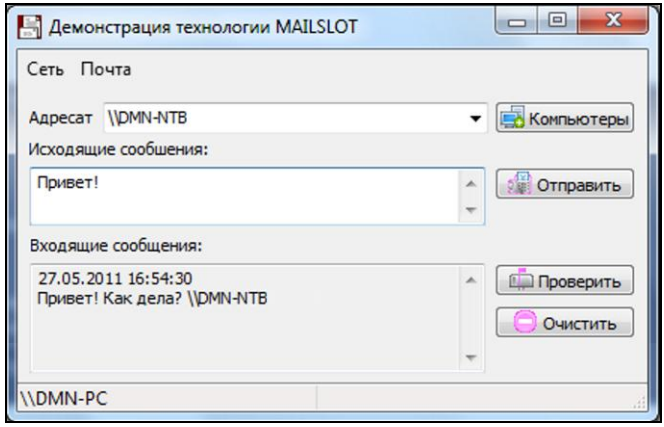


Рис. 39.2. Интерфейс приложения почтового слота

Именованные каналы

Выбор термина *"канал"* (pipe) обусловлен тем, что связь между приложениями осуществляется по некоторому виртуальному каналу, проложенному в сети между двумя компьютерами. Механизм каналов построен таким образом, что приложения по большому счету и не задумываются, каким это чудесным образом их послания преодолевают расстояния между компьютерами. Приложения не интересуют, какие сетевые протоколы задействуются, а какие нет, как работают модемы или сетевые адаптеры. Это не их дело... Они считают, что канал — это разделяемая между процессами область памяти, используемая приложениями для связи между собой.

Каналы классифицируют как *неименованные* (anonymous pipe) и *именованные* (named pipe). Реализация анонимных каналов проще, они вообще не предназначены для работы в сети, но готовы выполнять функционал обмена данными между сервером и клиентом в рамках одного и того же компьютера. С помощью функции `CreatePipe()` сервер неименованного канала формирует разделяемую область памяти, а взамен получает два описывающих эту область дескриптора. Первый дескриптор предназначен для операций чтения из памяти, второй — для записи данных в эту же память. Операции чтения и записи осуществляются так, как будто клиент и сервер работают с файлом. Для этого превосходно подходят методы `FileRead()` и `FileWrite()` из модуля `SysUtils`. Для освобождения дескриптора, как всегда, вызывается метод `CloseHandle()`. Вот в двух словах перечень подвигов, на которые способны неименованные каналы, нас же в большей степени интересуют их именованные коллеги. Ведь благодаря способностям именованного канала можно связать сервер и одно или несколько клиентских приложений, выполняющихся на разных компьютерах, узами односторонней (симплексной) или двусторонней (дуплексной) связи.

Определение имени именованного канала

Для определения имени канала во время разработки сервера программист должен придерживаться следующего синтаксиса: `\\.\pipe\<имя_канала>`.

Имя канала не чувствительно к регистру символов и не должно содержать более 256 символов. Например: `const PipeName = '\\.\pipe\mypipe'`.

При обращении к именованному каналу клиентские приложения заменяют точку в его имени на сетевое имя сервера, предоставляющего услугу именованных каналов:

```
const PipeName='\\mainserver\pipe\mypipe';
```

Создание именованного канала

Приложение-сервер, инициирующее создание именованного канала, вызывает весьма непростую функцию `CreateNamedPipe()`. Все особенности эксплуатации созданного при помощи нее именованного канала напрямую зависят от применяемых при вызове функции параметров. В свою очередь, характеристики созданного именованного канала в дальнейшем определят порядок разработки клиентских приложений.

```
function CreateNamedPipe(PipeName : pWideChar;
                         OpenMode, PipeMode, MaxInstances, OutBufferSize,
                         InBufferSize, DefaultTimeOut : Cardinal;
                         SecurityAttr : pSecurityAttributes) : THandle;
```

За присвоение каналу имени отвечает параметр `PipeName`.

Второй параметр `OpenMode` решает триединую задачу: во-первых, определяет тип обмена данными и направление потока данных (табл. 39.1), во-вторых, устанавливает поведение метода при передаче больших объемов данных и управляет кэшированием (табл. 39.2) и, в-третьих, управляет атрибутами безопасности (табл. 39.3).

Таблица 39.1. *Возможные значения аргументов для параметра* `OpenMode`

Режим доступа	Права сервера	Права клиента	Тип обмена
`PIPE_ACCESS_INBOUND`	Только чтение	Только запись	Симплекс
`PIPE_ACCESS_OUTBOUND`	Только запись	Только чтение	Симплекс
`PIPE_ACCESS_DUPLEX`	Чтение и запись	Чтение и запись	Дуплекс

Наступил черед третьего по счету параметра — `PipeMode`, он определяет:

- каким образом данные направляются в канал — в виде последовательности байтов `PIPE_TYPE_BYTE` или как поток сообщений `PIPE_TYPE_MESSAGE`;
- как производится чтение данных из канала — как поток байтов `PIPE_READMODE_BYTE` или поток `PIPE_READMODE_MESSAGE` сообщений;
- как будет происходить возврат управления методами, осуществляющими запись или чтение данных. Флаг `PIPE_WAIT` приказывает этим методам не отдавать управление программе, пока не будет завершена операция (такой режим называют синхронным блокирующим). Напротив, флаг `PIPE_NOWAIT` разрешает методам возвращать управление программе немедленно (синхронный неблокирующий режим).

> ***Замечание***
>
> Режимы передачи данных в канал и чтение данных из него должны быть согласованы. Если отправляем сообщение в виде потока байтов `PIPE_TYPE_BYTE`, то и прием обязательно должен осуществляться потоком `PIPE_READMODE_BYTE`. Режим `PIPE_TYPE_MESSAGE` менее критичен и допускает чтение с любыми значениями флагов.

Четвертый параметр `MaxInstances` накладывает ограничение на максимальное число экземпляров создаваемого канала. Допускается любое число в пределах от 1 до `PIPE_UNLIMITED_INSTANCES` (255 соединений). Наиболее интересным (но не единственным) решением, реали-

Таблица 39.2. *Режимы обмена данными, определяемые параметром* `OpenMode`

Режим обмена	Описание
`FILE_FLAG_OVERLAPPED`	Асинхронный режим ввода/вывода. Включение флага рекомендуется в случаях, когда объем передаваемых в канал данных достаточно велик и процесс их передачи может занимать ощутимый промежуток времени. Созданный с этим флагом канал допускает производить передачу данных в фоновом режиме. В асинхронном режиме обмена функции, осуществляющие передачу данных, не дожидаются завершения отправки сообщения и немедленно возвращают управление программе. При включенном флаге `FILE_FLAG_OVERLAPPED` операции ввода и вывода должны быть реализованы с непременным заполнением структуры `OVERLAPPED`
`FILE_FLAG_WRITE_THROUGH`	Этот флаг имеет смысл, только если данные передаются потоком байтов. Если флаг включен, то данные немедленно направляются в канал, если же флаг отключен, то перед началом передачи данные накапливаются в кэше и отправляются к корреспонденту в составе единого пакета только после его заполнения. Таким образом, оптимизируется сетевой трафик, т. к. вместо сотни одиночных отправлений передается объединенный пакет сообщений. Если этот флаг при создании канала не применялся, то основные параметры пакета (размер и время ожидания) настраиваются операционной системой или методом `SetNamedPipeHandleState()`
`FILE_FLAG_FIRST_PIPE_INSTANCE`	Допускает создание только одного экземпляра канала, попытка создания второго экземпляра вызывает ошибку

Таблица 39.3. *Атрибуты безопасности, определяемые параметром* `OpenMode`

Флаг	Описание
`WRITE_DAC`	Вызывающий процесс должен иметь права доступа на запись к произвольному управляющему списку доступа именованного канала (access control list, ACL)
`WRITE_OWNER`	Вызывающий процесс должен иметь права доступа на запись к процессу, владеющему именованным каналом
`ACCESS_SYSTEM_SECURITY`	Процесс должен иметь права доступа на запись к управляющему списку доступа именованного канала ACL

зующим многоканальное приложение, станет создание для каждого канала отдельного программного потока. В этом случае клиентское приложение обращается к серверу в своем выделенном канале, а обслуживание клиента на серверной стороне осуществляется в отдельном потоке. В качестве другого варианта можно рассмотреть поочередное предоставление одного и того же канала для нескольких клиентов.

Пятый и шестой параметры метода (`OutBufferSize` и `InBufferSize`) определяют размеры выходного и входного буферов, в которых кэшируются отправляемые и принимаемые данные. Определение размера — дело творческое, безусловно, размер должен быть согласован с объемом передаваемых по каналам сообщений. Однако при этом не стоит упускать из виду, что за резервирование памяти для буферов отвечает ядро операционной системы, с созданием каждого канала его буферам отводится своя область в памяти, причем память выде-

ляется не в страничном пуле, а прямо в ОЗУ. Так что при определении размера кэша не стоит разбрасываться мегабайтами...

Время тайм-аута ожидания клиентом канала в миллисекундах определяет параметр DefaultTimeOut. Установленное в нем значение имеет смысл лишь тогда, когда на клиентской стороне процесс ожидания освобождения канала обеспечивается методом WaitNamedPipe() с включенным флагом NMPWAIT_USE_DEFAULT_WAIT.

Последний параметр SecurityAttributes представляет собой указатель на структуру, определяющую политику безопасности в вопросе получения доступа к дескриптору создаваемого именованного канала различными клиентскими процессами. По умолчанию этот параметр допускает передачу пустого указателя nil.

Некоторые изменения в настройки уже созданного экземпляра канала можно внести с помощью функции

```
function SetNamedPipeHandleState(hPipe : THandle; var Mode : cardinal;
          MaxCollectionCount, CollectDataTimeout : pointer) : boolean;
```

Первый параметр — дескриптор канала. Параметр Mode определяет, каким образом данные передаются в канал (флаг PIPE_READMODE_BYTE или PIPE_READMODE_MESSAGE) и как производится возврат из методов, обратившихся к каналу (флаг PIPE_WAIT или PIPE_NOWAIT). Еще два указателя MaxCollectionCount и CollectDataTimeout имеют смысл, если канал создавался без флага FILE_FLAG_WRITE_THROUGH. В этом случае указатели определяют основные параметры передаваемого пакета. Для оптимизации сетевого трафика перед отправкой данные накапливаются в пакете и передаются в канал, только после достижения пакетом определенного размера MaxCollectionCount или по истечении времени ожидания CollectDataTimeout.

Управление соединением с клиентом

Создавший именованный канал сервер на правах старшинства управляет процессом подключения к каналу и отключения от него клиентских приложений. Такая привилегия подкреплена двумя серьезными функциями — ConnectNamedPipe() и DisconnectNamedPipe(). Вызов метода ConnectNamedPipe() переводит сервер в режим ожидания, пока к экземпляру канала не подключится какое-нибудь клиентское приложение. Вызов функции целесообразен сразу после создания экземпляра канала (в особенности, если сервер создает в отдельных потоках сразу несколько каналов) или сразу после отключения от канала клиента.

```
function ConnectNamedPipe(hPipe : THandle;
                          OVERLAPPED : pOVERLAPPED):Boolean;
```

Здесь: hPipe — дескриптор именованного канала; OVERLAPPED — указатель на структуру, используемую при операциях асинхронного ввода/вывода. Указатель на эту структуру передается лишь в тех случаях, когда канал был создан с применением флага FILE_FLAG_OVERLAPPED. Если канал перевелся в режим готовности к подключению, метод возвратит значение true. Для канала, созданного в синхронном блокирующем режиме (с использованием PIPE_WAIT), рассматриваемая функция переходит в состояние ожидания соединения с клиентским процессом. Если канал создан в синхронном неблокирующем режиме (PIPE_NOWAIT), метод ConnectNamedPipe() немедленно возвращает управление с кодом true, если только клиент был отключен от данной реализации канала и возможно подключение этого клиента. В противном случае возвращается значение false.

Замечание

Возврат методом `ConnectNamedPipe()` значения `false` не может считаться однозначным признаком того, что подключение завершилось неудачно. После выполнения этой функции программисту следует проанализировать значение последней ошибки. Так метод `GetLasError()` возвратит код ошибки `ERROR_PIPE_LISTENING` в случае, если вообще нет желающих для подключения к именованному каналу. Кроме этого, высока вероятность встретить ошибки: `ERROR_PIPE_CONNECTED` — канал уже занят, `ERROR_NO_DATA` — клиент отказался от канала, но не был отключен от него сервером методом `DisconnectNamedPipe()`.

Особенность совместной работы клиента и сервера именованных каналов в том, что отказ клиента от канала (вызов им функции `CloseHandle()`) физически не отключает его от сервера. Канал по-прежнему остается в занятом состоянии, хотя данные по нему уже не транслируются. В клиент-серверной технологии окончательный разрыв связи считается привилегией сервера, в частности при работе с именованным каналом для этого вызывается функция

```
function DisconnectNamedPipe(hPipe : THandle) : boolean;
```

Программная логика сервера перед отключением канала должна предусматривать действия, заставляющие принудительно передать в ресурс все накопленные в кэше данные. Для этого вызывается функция `FlushFileBuffers()`.

Состояние канала

Как сервер именованных каналов, так и клиентское приложение имеют право ознакомиться с основными техническими характеристиками работающего канала. Для этого предназначена функция

```
function GetNamedPipeInfo(hPipe : THandle; var Flags : cardinal;
         OutBufferSize, InBufferSize, MaxInstances : pointer) : boolean;
```

Первый параметр метода — дескриптор опрашиваемого канала, все остальные параметры информационные. Так `Flags` расскажет о типе канала (поток байтов `PIPE_TYPE_BYTE` или поток сообщений `PIPE_TYPE_MESSAGE`) и кем вызван метод — сервером `PIPE_SERVER_END` или клиентом `PIPE_CLIENT_END`. Параметры `OutBufferSize` и `InBufferSize` проинформируют о размерах выходного и входного буферов. Параметр `MaxInstances` вернет данные о максимально допустимом количестве каналов. В случае успешного завершения функция возвращает значение `true`, а при ошибке — `false`.

Передав дескриптор именованного канала `hPipe` в функцию

```
function GetNamedPipeHandleState(hPipe : THandle;
    State, CurInstances, MaxCollectionCount, CollectDataTimeout : pDWORD;
    UserName : PWideChar; MaxUserNameSize : DWORD) : boolean;
```

мы получим подробные сведения о канале.

О состоянии канала расскажет параметр `State`. Этот указатель на двойное слово `DWORD` может возвратить два флага. Флаг `PIPE_READMODE_MESSAGE` свидетельствует о том, что канал специализируется на передаче сообщений. Флаг `PIPE_NOWAIT` признак того, что при обращении к каналу функции будут возвращать управление в программу немедленно, не дожидаясь окончания выполнения.

О числе существующих экземпляров канала поведает параметр `CurInstances`.

Клиент именованного канала осуществляет передачу данных не по мере их поступления, а накапливает их в пакет и потом отправляет серверу в составе пакета. Максимальный размер пакета (в байтах) будет возвращен при помощи указателя `MaxCollectionCount`.

Чтобы информация, находящаяся в пакете, была гарантированно передана серверу (даже если пакет не будет наполнен целиком), процесс передачи дополнительно контролируется таймером. О времени тайм-аута нас информирует параметр `CollectDataTimeout`.

> ***Замечание***
>
> К сожалению, нельзя обращаться к параметрам: `MaxCollectionCount` и `CollectDataTimeout` в случае, если серверный и клиентский концы канала находятся на одной и той же станции. В этом случае вместо указателя на переменную надо передавать `nil`.

Если вызов метода осуществляется со стороны сервера и клиент получил доступ к каналу с флагом `SECURITY_IMPERSONATION`, то также возможно узнать об имени пользователя, работающего на клиентском компьютере. Эти данные возвратит указатель `UserName`. При этом максимальная длина имени ограничивается последним параметром `MaxUserNameSize`.

Подключение к каналу клиентского приложения

Подключение клиентского приложения к каналу осуществляется либо методом `FileCreate()`, либо при помощи метода `CallNamedPipe()`. Первый вариант подходит в ситуациях, когда по каналу предполагается передавать поток байтов, второй, когда канал специализируется на работе с сообщениями. Если канал свободен, то метод возвращает дескриптор канала, если же канал занят, то клиентское приложение имеет право подождать освобождения канала, но для этого потребуется вызвать метод `WaitNamedPipe()`.

Функция

```
function CallNamedPipe(PipeName : pChar; InBuffer : Pointer;
                       InBufferSize : Cardinal; OutBuffer : Pointer;
                       OutBufferSize : Cardinal; var BytesRead : Cardinal;
                       TimeOut : Cardinal) : boolean;
```

подключается к именованному каналу, при необходимости ждет его освобождения, осуществляет единовременную операцию чтения и записи и отключается от канала. Здесь: `PipeName` — имя канала; `InBuffer` — указатель на входной буфер записи (в котором будут накапливаться данные, направляемые в канал); `InBufferSize` — размер буфера записи в байтах; `OutBuffer` и `OutBufferSize` — соответственно указатель и размер буфера чтения; `BytesRead` — переменная, в которую будут передаваться данные о количестве байтов, считанных из канала. Время ожидания готовности канала определяется в миллисекундах и находится в последнем параметре — `TimeOut`. Кроме этого, вместо точного значения тайм-аута в этот параметр допустимо передавать одну из констант, представленных в табл. 39.4.

***Таблица 39.4.** Управление тайм-аутом в функции `CallNamedPipe()`*

Значение	Описание
`NMPWAIT_NOWAIT`	Никогда не ждать. Если канал по какой-то причине недоступен — сразу возвращать ошибку подключения
`NMPWAIT_WAIT_FOREVER`	Неограниченное ожидание
`NMPWAIT_USE_DEFAULT_WAIT`	В качестве тайм-аута применять время, определенное сервером в параметре `DefaultTimeOut` метода `CreateNamedPipe()`

Для организации ожидания занятого канала клиентское приложение имеет право воспользоваться функцией

```
function WaitNamedPipe(NamedPipeName : pWideChar;
                       TimeOut : cardinal) : boolean;
```

Название канала передается указателем NamedPipeName(), время ожидания назначается в миллисекундах во втором параметре TimeOut функции. Кроме этого, тайм-аут воспринимает две константы: NMPWAIT_WAIT_FOREVER и NMPWAIT_USE_DEFAULT_WAIT. Константа NMPWAIT_WAIT_FOREVER переводит метод в состояние вечного ожидания, в этом состоянии WaitNamedPipe() не возвращает управления вызвавшей ее программе до тех пор, пока экземпляр вызываемого канала не станет доступным. Вторая константа NMPWAIT_USE_DEFAULT_WAIT определяет, что время ожидания должно соответствовать времени тайм-аута, установленного при создании именованного канала. В результате, если запрашиваемый канал освободился за интервал времени, установленный в тайм-ауте, то метод вернет значение true. Если по указанному адресу вообще отсутствуют экземпляры каналов, то метод завершит работу немедленно, не обращая внимания на значение своего тайм-аута.

Разработка класса сервера именованного канала

При проектировании сервера именованного канала стоит объединить достоинства высокоскоростных функций прикладного программного интерфейса Windows API и удобства объектно-ориентированного языка Delphi и реализовать сервер в виде самостоятельного класса.

Листинг 39.6 содержит объявление класса TPipeServer, способного создать отдельный именованный канал. Класс содержит минимальный набор полей и методов. Поля Handle и Overlapped предназначены для хранения дескриптора канала и структуры, обеспечивающей асинхронный режим работы. Помимо конструктора и деструктора в арсенал возможностей класса войдет метод установления соединения и методы чтения/записи данных.

Листинг 39.6. Объявление класса сервера именованного канала

```
type
  EPipeBroken = exception;        //ИС (искл. ситуация) разрыва канала
  EPipeConnectError = exception;  //ИС ошибки соединения
  EPipeError = exception;         //ИС ошибки канала
  TPipeServer= class
  private
    Handle:THandle;          //дескриптор канала
    Overlapped:TOverlapped; //для асинхронного режима работы
  public
    constructor Create(PipeName:String; BufferSize:DWORD);
    destructor Destroy;
    function Connect:boolean;             //функция соединения
    function Read(var Buf; Count:Integer):integer; //чтение
    function Write(var Buf; Count:Integer):integer;//запись
  end;
```

Канал создается в рамках конструктора класса (листинг 39.7).

Листинг 39.7. Конструктор класса сервера именованного канала

```
constructor TPipeServer.Create(PipeName: String;BufferSize:DWORD);
var SD : TSecurityDescriptor;
    SA : TSecurityAttributes;
```

```
begin
  //инициализация атрибутов безопасности
  InitializeSecurityDescriptor(@SD,SECURITY_DESCRIPTOR_REVISION);
  SA.nLength:=SizeOf(TSecurityAttributes);
  SA.lpSecurityDescriptor:=@SD;
  SA.bInheritHandle:=True;
  //подготовка структуры для асинхронного режима
  FillChar(Overlapped, 0, SizeOf(TOverlapped));
  Overlapped.hEvent := CreateEvent (nil, True, False, nil);
  //создание канала
  Handle := CreateNamedPipe(pChar(PipeName),
              PIPE_ACCESS_DUPLEX OR FILE_FLAG_OVERLAPPED,
              PIPE_WAIT OR PIPE_READMODE_MESSAGE OR PIPE_TYPE_MESSAGE,
              PIPE_UNLIMITED_INSTANCES,
              BufferSize,
              BufferSize,
              0,
              @SA);

  if Handle <= 0 then raise EPipeError.CreateFmt(
                         'Ошибка создания канала %d!', [GetLastError]);
end;
```

В первых строках метода инициируем структуру `TSecurityAttributes` для того, чтобы предоставить возможность обращаться к каналу клиентским процессам, не обладающим правами администратора. Затем распределяем память под структуру `Overlapped`, поддерживающую асинхронный режим работы, и создаем событие синхронизации. Для упрощения примера в конструктор передаются всего два аргумента: имя канала `PipeName` и размер входного и выходного буферов именованного канала — `BufferSize`. Далее формируем канал с наиболее востребованными характеристиками: дуплексный, асинхронный, работающий в режиме обмена сообщениями, с максимально допустимым числом экземпляров.

После создания экземпляра канала наступает очередь метода `Connect()`, отвечающего за перевод сервера в режим готовности к соединению (листинг 39.8).

Листинг 39.8. Метод соединения

```
function TPipeServer.Connect: boolean;
begin
Result:=ConnectNamedPipe(Handle,@Overlapped); //готовность к соединению
if Result=false then
  case GetLastError of
    ERROR_PIPE_CONNECTED: Result := true;
    ERROR_IO_PENDING:
      if WaitForSingleObject(Overlapped.hEvent, INFINITE)=
                             WAIT_OBJECT_0 then Result := true;
      else raise EPipeConnectError.Create('Не готов к соединению!');
  end;
end;
```

Настал черед методов, отвечающих за операции чтения/записи в канал. Для этого достаточно задействовать классические для Windows API функции чтения и записи в файл (листинги 39.9 и 39.10).

Листинг 39.9. Метод чтения данных

```
function TPipeServer.Read(var Buf; Count: Integer): integer;
var NumberOfBytesToRead : Cardinal;
begin
  if ReadFile(Handle, Buf, Count, NumberOfBytesToRead, @Overlapped)=
    false then
    case GetlastError of
      ERROR_IO_PENDING: begin
                        WaitForSingleObject(Overlapped.hEvent, INFINITE);
        GetOverlappedResult(Handle, Overlapped, NumberOfBytesToRead, TRUE);
                        end;
      ERROR_BROKEN_PIPE: raise EPipeBroken.Create('Канал разорван')
        else raise EPipeError.Create('Сбой чтения, ошибка');
    end
  else Result := NumberOfBytesToRead;
  Result := NumberOfBytesToRead;
end;
```

Листинг 39.10. Метод записи данных

```
function TPipeServer.Write(var Buf; Count: Integer): integer;
var NumberOfBytesToWrite : Cardinal;
begin
  if WriteFile(Handle, Buf, Count, NumberOfBytesToWrite, @Overlapped)=false
  then
    case GetLastError of
      ERROR_IO_PENDING : WaitForSingleObject (Overlapped.hEvent, INFINITE);
      ERROR_NO_DATA    : raise EPipeError.Create('Ошибка записи');
    end;
  GetOverlappedResult(Handle, Overlapped, NumberOfBytesToWrite, True);
  Result := NumberOfBytesToWrite;
end;
```

Работа над классом завершается описанием канала деструктором (листинг 39.11). В коде деструктора предусматривается не только уничтожение экземпляра канала, но и очистка буфера с данными и разрушение события синхронизации.

Листинг 39.11. Деструктор класса сервера именованного канала

```
Destructor TPipeServer.Destroy;
begin
  FlushFileBuffers(Handle); //отправка оставшихся данных и очистка буфера
  DisconnectNamedPipe(Handle); //разъединение с клиентом
  CloseHandle(Overlapped.hEvent); //освобождение события синхронизации
  CloseHandle(Handle); //закрываем дескриптор канала
end;
```

Разработка класса клиента именованного канала

Разработаем класс TPipeClient, предназначенный для обслуживания именованного канала на клиентской стороне. Клиентский класс более прост, т. к. пользуется услугами уже существующего канала, поэтому кроме конструктора и деструктора нам потребуются всего два метода чтения и записи (листинг 39.12).

Листинг 39.12. Объявление класса клиента именованного канала

```
type TPipeClient = class
  private
    Handle:THandle; //дескриптор канала
    OverlappedRead, OverlappedWrite : TOverlapped;
  public
    constructor Create(const PipeName: String);
    destructor Destroy;
    function Read(var Buffer; Count: Longint): Longint;
    function Write(const Buffer; Count: Longint): Longint;
end;
```

Основная особенность клиента именованного канала в том, что он работает с каналом так, как будто это обычный файл. Этим объясняется вся логика функционирования класса TPipeClient. В момент создания экземпляра класса мы "обманываем" клиента, предоставляя в его распоряжение дескриптор канала под видом дескриптора файла (листинг 39.13).

Листинг 39.13. Реализация конструктора класса клиента именованного канала

```
constructor TPipeClient.Create(const PipeName: String);
begin
  Handle := CreateFile(pCHAR(PipeName), GENERIC_READ OR GENERIC_WRITE,
                  0, nil, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);
  if Handle <=0  then
    case GetLastError of
      ERROR_SEEK_ON_DEVICE :
              raise ENamedPipeError.Create('Ошибка устройства');
      ERROR_FILE_NOT_FOUND :
              raise ENamedPipeError.Create('Канал не обнаружен');
      else    raise ENamedPipeError.Create('Ошибка открытия канала');
    end;
  FillChar(OverlappedRead, SizeOf(OverlappedRead), 0);
  FillChar(OverlappedWrite, SizeOf(OverlappedWrite), 0);
  OverlappedRead.hEvent := CreateEvent(nil, true, false, nil);
  OverlappedWrite.hEvent := CreateEvent(nil, true, false, nil);
end;
```

Операции чтения и записи осуществляются стандартными функциями Windows ReadFile() и WriteFile(), при этом мы не забываем, что нам следует учитывать асинхронный режим доступа к каналу (листинги 39.14 и 39.15).

Листинг 39.14. Метод чтения

```
function TPipeClient.Read(var Buffer; Count: Integer): Longint;
var NumberOfBytesToRead : Cardinal;
begin
  Result := 0;
  if ReadFile(Handle, Buffer, Count,
              NumberOfBytesToRead, @OverlappedRead)=false then
    raise ENamedPipeError.Create('Ошибка чтения ');
  else  Result := NumberOfBytesToRead;
end;
```

Листинг 39.15. Метод записи

```
function TPipeClient.Write(const Buffer; Count: Integer): Longint;
var NumberOfBytesToWrite : Cardinal;
begin
  Result := 0;
  if WriteFile(Handle, Buffer, Count,
               NumberOfBytesToWrite, @OverlappedWrite)=false then
     raise ENamedPipeError.Create('Ошибка записи')
  else Result := NumberOfBytesToWrite;
end;
```

ПРИМЕЧАНИЕ

Объем книги не позволяет предоставить листинги приложений сервера и клиента именованных каналов в полном объеме. Вы их найдете в электронном архиве к книге *(см. приложение 10)*.

Сокеты

Одним из наиболее востребованных механизмов взаимодействия сетевых приложений считается технология *сокетов* (sockets). Важное преимущество сокетов в том, что они поддерживаются практически всеми современными операционными системами и поэтому позволяют организовывать сетевой обмен между ОС различных производителей.

ЗАМЕЧАНИЕ

Появлению сокетов мы в первую очередь обязаны Калифорнийскому университету Беркли и операционной системе UNIX, в которой идея нашла свое первое применение. В программных продуктах корпорации Microsoft технология претерпела ряд доработок и сегодня классифицируется как Windows Sockets 2 или сокращенно WinSock.

Технология WinSock построена по принципу "клиент-сервер". Сразу после создания сервер переходит в режим ожидания связи (для этого предложен термин *"прослушивание"* — listening). Услышав "стук в дверь", сервер открывает сеанс с клиентом, в рамках которого осуществляется односторонний или двусторонний обмен данными.

Классы сокетов в VCL

В отличие от почтовых слотов и каналов (для которых не существует официальных компонентов VCL) сокеты более удачливы. Открыв страницу **Internet** палитры компонентов

Delphi, вы обнаружите тройку компонентов `TTCPServer`, `TTCPClient` и `TUDPClient`, инкапсулирующих интерфейс сокетов. У этих компонентов очень много схожего, в первую очередь их объединяют общие предки — классы `TBaseSocket` и `TIpSocket` (рис. 39.3).

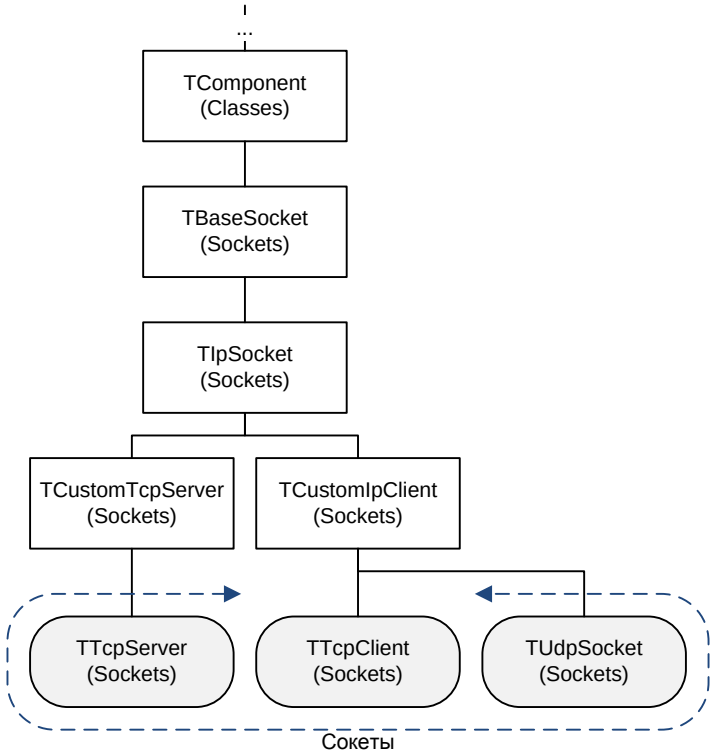


Рис. 39.3. Компоненты-сокеты VCL

Внимание!

В составе Delphi имеется большой перечень компонентов Indy, существенно упрощающих проектирование сетевых приложений. Эти компоненты разработаны вне стен Embarcadero, более подробную информацию вы сможете получить на сайте **http://www.indyproject.org**.

Общие черты сокетов, опорный класс *TIPSocket*

Традиционно, перед изучением конкретных компонентов уделим внимание их предку — классу `TIPSocket`. Благодаря такому подходу мы познакомимся с общими чертами всех компонентов, а потом настанет время и для деталей.

Начнем с того, что хотя интерфейс сокетов может опираться на различные сетевые протоколы, но класс `TIPSocket` и, соответственно, все его потомки изначально ориентированы на доминирующий сегодня стек протоколов TCP/IP и UDP/IP. В недрах компонентов спрятано свойство

```
property Protocol: TSocketProtocol;
```

Тип данных `TSocketProtocol` — это не что иное, как беззнаковое целое `Word`. Если вы обратитесь к этому свойству у компонентов `TTCPServer` и `TTCPClient`, то в ответ получите нуле-

вое значение. Это означает, что компоненты "исповедуют" протокол IP. Их коллега TUDPSocket ориентирован на протокол пользовательских дейтаграмм UDP, поэтому он возвратит значение 17. Возвращаемые свойством значения соответствуют константам из модуля Winapi.WinSock (табл. 39.5).

Таблица 39.5. *Константы протокола сокета, объявленные в модуле* `Winapi.WinSock`

Константа	Значение	Описание
IPPROTO_IP	0	Internet Protocol
IPPROTO_ICMP	1	Internet Control Message Protocol, протокол управляющих сообщений Интернета. В первую очередь предназначен для построения таблиц маршрутизации и диагностики проблем
IPPROTO_IGMP	2	Internet Group Management Protocol, протокол группового управления Интернетом. Обеспечивает передачу дейтаграмм группе хостов
IPPROTO_TCP	6	Transmission Control Protocol. Протокол управления передачей обеспечивает сервис гарантированной доставки информации в виде потока байтов
IPPROTO_UDP	17	User Datagram Protocol, протокол пользовательских дейтаграмм. Дополнительный компонент протокола TCP, поддерживающий выполняющуюся без подключений службу дейтаграмм, не гарантирующую ни доставку, ни правильную последовательность доставленных пакетов
IPPROTO_ND	77	Протокол поддержки сетевых дисков
IPPROTO_RAW	255	Неструктурированный IP-пакет

Очередная важная характеристика сокетов доступна благодаря свойству

```
property SockType : TSocketType;
```

Свойство определяет способ передачи данных между сетевыми приложениями (табл. 39.6).

Таблица 39.6. *Тип сокета*

Константа	Значение	Описание
stStream	1	Значение по умолчанию, представляет собой наиболее надежный формат обмена пакетами. Работает с протоколом TCP
stDgram	2	Данные поступают в виде простой дейтаграммы, доставка дейтаграммы не гарантируется. Работает с протоколом UDP
stRaw	3	Разрешенный в WinSock 2.2 обмен в виде неструктурированных (raw) пакетов данных в интересах приложений более высокого уровня
stRdm	4	Reliably delivered message — надежная доставка сообщений
stSeqPacket	5	Организует поток псевдопакетов, основанный на дейтаграммах

По умолчанию сервер TTCPServer и клиент TTCPClient нацелены на работу в режиме stStream, а их коллега TUDPSocket предпочитает дейтаграммный обмен stDgram.

ЗАМЕЧАНИЕ

Корректное преобразование значения, возвращаемого свойством SockType, в соответствующий ей числовой код производится методом MapSockType().

Еще одна базовая характеристика сокета связана с определением семейства протоколов, в которых должен работать сокет

```
property Domain: TSocketDomain;
type TSocketDomain = (pfUnspec, pfUnix, pfInet, pfImpLink, pfPup, pfChaos, pfIpx,
pfNs, pfIso, pfOsi, pfEcma, pfDataKit, pfCcitt, pfSna, pfDecNet, pfDli, pfLat,
pfHylink, pfAppleTalk, pfVoiceView, pfFireFox, pfUnknown1, pfBan, pfMax);
```

Стандартное состояние всех изучаемых в этой главе компонентов соответствует семейству `pfInet`.

> ***Замечание***
>
> Корректное преобразование значения, возвращаемого свойством `Domain`, в соответствующий ей числовой код производится методом `MapDomain()`.

При организации операций ввода/вывода программисту следует определиться с порядком ожидания ответа одного корреспондента сетевого обмена от другого — асинхронным или синхронным (или, используя терминологию сокетов, не блокирующим (`bmNonBlocking`) и блокирующим (`bmBlocking`) режимами). За установку режима блокировки отвечает свойство

```
property BlockMode: TSocketBlockMode;
```

У компонента `TTCPServer`, отвечающего за реализацию серверного окончания WinSock, предусмотрен еще один режим блокировки — `bmThreadBlocking`.

Адрес сокета

Полагаю, что читатель знаком с особенностями адресации хостов в Интернете и ему не нужен подробный экскурс в IP-адресацию. Поэтому ограничусь напоминанием о том, что для однозначной идентификации хоста в сетях, построенных на стеке протокола TCP/IP, ему необходимо присвоить уникальный 4-октетный (или 6-октетный) IP-адрес. Для обращения к конкретной службе, выполняющейся на компьютере, следует знать номер порта. Да простят меня сетевые специалисты, но для быстрого ввода начинающего в курс дела позволю себе сравнить IP-адрес с адресом дома, а номер порта — с номером квартиры в этом доме.

В первую очередь стоит отметить, что на уровне `TIpSocket` уже реализована поддержка адреса как клиентской стороны

```
property LocalHost: TSocketHost;//локальный хост
property LocalPort: TSocketPort;//локальный порт
```

так и адрес корреспондента

```
property RemoteHost: TSocketHost;//удаленный хост
property RemotePort: TSocketPort;//удаленный порт
```

Хотя адресные данные передаются в сокет в виде обычной текстовой строки, но сам сокет предпочитает хранить адрес в более удобной для него специализированной структуре `TSockAddr`. Преобразование адреса с человеческого языка на язык WinSock осуществляет метод

```
function GetSocketAddr(h: TSocketHost; p: TSocketPort): TSockAddr;
```

Однако для удобства пользователя большинство методов класса представляют адресные данные в понятном виде. Эти методы предназначены для взаимного преобразования различных типов адресов, а также способны возвратить отдельные элементы адреса как нашего, так и удаленного сокета (табл. 39.7).

Таблица 39.7. *Методы управления адресом*

Метод	Описание
`function LookupHostName(const ipaddr: string): TSocketHost;`	Преобразует IP-адрес в имя хоста
`function LookupHostAddr(const hn: string): TSocketHost`	Преобразует имя хоста `hn` в его IP-адрес. В случае неудачи будет возвращена строка '0.0.0.0'
`function LookupProtocol(const pn: string): TSocketProtocol;`	Преобразует имя протокола `pn` в его номер
`function LookupPort(const sn: string; pn: pchar = nil): word;`	Возвратит номер порта сокета по имени сервиса `sn` и номеру протокола `pn`
`function LocalDomainName: string;`	Доменное имя локального хоста
`function LocalHostName: TSocketHost;`	Имя локального хоста
`function LocalHostAddr: TSocketHost;`	IP-адрес локального хоста

Отправка и получение данных

Для отправки корреспонденту информации в базовый класс заложены два ключевых метода:

```
function SendBuf(var buf; bufsize: integer; flags: integer = 0): integer;
function SendStream(AStream: TStream): Integer;
```

Функция `SendBuf()` передает данные из буфера `buf` размером `bufsize`. Флаг `flags` необходим только для работы под управлением ОС Linux, так что оставляйте его в покое со значением по умолчанию. Второй метод `SendStream()` специализируется на передаче данных из памяти (точнее, потока) `AStream`. Над методом `SendBuf()` существует надстройка для работы с текстовыми данными

```
function Sendln(s: string; const eol: string = CRLF): integer;
```

Этот метод предназначен для отправки строки `s`. Второй параметр метода `eol` определяет символ окончания строки. Рассматриваемый метод вызывает функцию `SendBuf()` и заполняет передающий буфер порцией данных до символа-разделителя.

Задачу получения данных решают схожие по параметрам методы

```
function ReceiveBuf(var buf; bufsize: integer;
                    flags: integer = 0): integer;
function Receiveln(eol: string = CRLF): string;
```

Для копирования содержащихся в буфере данных может пригодиться метод:

```
function PeekBuf(var buf; bufsize: integer): integer;
```

Эта функция может применяться совместно с методами `SendBuf()` и `ReceiveBuf()`.

Отправка и получение данных находят свое отражение в паре событий

```
property OnSend: TSocketDataEvent;
property OnReceive: TSocketDataEvent;
```

Событие `OnSend()` генерируется перед передачей данных, а событие `OnRecive()` возникает только после получения данных сокетом. Оба события могут использоваться для дополни-

тельной обработки данных, например для форматирования текста. Оба события описаны однотипной процедурой

```
type TSocketDataEvent = procedure (Sender: TObject; Buf: pChar;
                                   var DataLen: integer) of object;
```

Здесь: `Sender` — ссылка на компонент-сокет; `Buf` — буфер, содержащий передаваемые (или принимаемые) данные; `DataLen` — количество байтов данных в буфере.

Счетчики количества отправленных и принятых байтов данных доступны из свойств

```
property BytesReceived: cardinal;
property BytesSent: cardinal;
```

Сервер, компонент *TTCPServer*

Серверное окончание интерфейса WinSock в визуальной библиотеке компонентов реализовано в виде класса `TTCPServer`. Сервер `TTCPServer` — это полноценный невизуальный компонент, который вы обнаружите на странице **Internet** палитры компонентов Delphi. На его базе достаточно легко реализовать элементарный сервер WinSock для работы в сетях под управлением протоколов TCP/IP и UDP/IP.

На самой первой стадии разработки проекта сервера программист принимает решение о режиме блокировки, в которой будет работать сервер. Выбор осуществляется свойством

```
property BlockMode : TServerSocketBlockMode; //bmThreadBlocking
type TServerSocketBlockMode =
                    (bmBlocking, bmNonBlocking, bmThreadBlocking);
```

В нашем распоряжении всего три режима. Два из них уже упоминались при обсуждении родительских по отношению `TTCPServer` классов. Блокирующее состояние сервера `bmBlocking` активирует синхронный режим обработки, здесь операции ввода/вывода не возвращают управление вызвавшему их приложению до окончания своего выполнения. Таким образом, работа приложения приостанавливается на неопределенное время. Напротив, неблокирующее состояние `bmNonBlocking` активирует асинхронную деятельность сокета и разрешает методам чтения/записи немедленно возвращать управление программе, не дожидаясь своего завершения, но в этом случае возникают определенные сложности в программной реализации сервера, например в обработке ошибок в поступающих данных. В режиме `bmThreadBlocking` процесс сбора заявок на обслуживание происходит в рамках специального потока `TServerSocketThread`, а для непосредственного обслуживания вошедшего в соединение клиента применяется отдельный поток — экземпляр класса `TClientSocketThread` (потока обслуживания). Таким образом, привередливые синхронные операции осуществляются в фоновом режиме, благодаря чему не блокируют выполнение приложения в целом.

Если сервер будет работать в синхронном режиме `bmThreadBlocking`, то доступ к управляющему потоку предоставляет свойство

```
property ServerSocketThread: TServerSocketThread;
```

Сам по себе экземпляр класса `TTCPServer` даже после вызова конструктора еще не является сокетом в полном смысле этого слова. Это пока только пустая оболочка. Компонент `TTCPServer` превращается в сокет после вызова метода `Open()`. Именно в нем посредством обращения к инкапсулированному методу Windows API `Socket()` создается экземпляр сокета-сервера. Новорожденный сокет сразу же интегрируется в любезно предложенный нами каркас VCL — компонент `TTCPServer`.

Но это еще не все, кроме физического создания сокета в рамках `Open()`:

- автоматически заполняется локальный адрес созданного серверного сокета;
- вызывается функция Windows API `listen()`, переводящая сервер в состояние прослушивания канала. Если свойство `BlockMode` установлено в `bmThreadBlocking`, то прослушивание проводится в отдельном потоке, за управление которым отвечает экземпляр класса `TServerSocketThread`.

Такие исключительные подробности об этапах выполнения `Open()` вносят ясность в череду событий, которыми сопровождаются эти этапы. Сразу после создания сокета сервера вызывается событие

```
property OnCreateHandle: TSocketNotifyEvent;
```

Затем, прежде чем сокет приступит к прослушиванию, инициируется событие

```
property OnListening: TNotifyEvent;
```

Кстати, косвенным признаком того, что сервер в текущий момент находится в состоянии прослушивания, служит значение `true`, возвращаемое свойством

```
property Listening: Boolean; //только для чтения
```

К последствиям вызова метода `Open()` также относится незамедлительный переход свойства `Active` в состояние `true`.

Рано или поздно к серверу обращается клиент с просьбой об обслуживании, а если сервер пользуется популярностью, то подобных просителей наберется немало. В ожидании обслуживания клиенты выстроятся в очередь. Решение о начале обслуживания клиента принимается при посредничестве метода

```
function Accept(var ClientSocket: TCustomIpClient): boolean;
function Accept: boolean;
```

В режиме `bmThreadBlocking` метод `Accept()` вызывается автоматически в момент, когда к своим прямым обязанностям приступит поток обслуживания. Метод из череды томящихся в ожидании сеанса клиентских сокетов извлекает сокет, который занял очередь первым, и узнает все его характеристики (имя хоста, номер порта, тип сокета, режим блокировки). Получив исчерпывающую информацию о потенциальном клиенте, функция `Accept()` возвратит `true` — это признак, что клиентский сокет может быть принят для обслуживания. Если речь идет о первом варианте метода, то, кроме того, будет возвращена ссылка на клиента в параметре `ClientSocket`.

В момент приема клиента на обслуживание генерируется событие:

```
property OnAccept: TSocketAcceptEvent;
type TSocketAcceptEvent = procedure (Sender: TObject;
                                     ClientSocket: TCustomIpClient) of object;
```

Это самое важное событие для сервера. В нем мы вновь встретимся со ссылкой на ожидающий обслуживания клиентский сокет — `ClientSocket`.

Если сервер функционирует в режиме `bmThreadBlocking`, то событию `OnAccept()` будет предшествовать событие `OnGetThread()`. Оно вызывается перед тем, как поток управления серверного сокета приступит к созданию потока обслуживания. В этот момент программист может подменить создаваемый по умолчанию поток обслуживания `TClientSocketThread` собственной версией.

Разрыв соединения производится методом

```
procedure Close;
```

В результате свойство Active примет значение true, и процедура уничтожит обслуживающий поток TServerSocketThread, но последнее произойдет при условии, что сервер использовал режим блокировки bmThreadBlocking. И в финале Close() вызовет событие

```
property OnDestroyHandle: TSocketNotifyEvent;
```

Клиенты, компоненты *TTCPClient* и *TUDPSocket*

Компоненты TTCPClient и TUDPSocket инкапсулируют функции Windows API, достаточные для создания клиента, работающего под управлением протоколов TCP и UDP.

Для инициирования соединения клиент обращается к методу Open(), в результате:

- создается экземпляр клиентского сокета, появление сокета на свет сопровождается генерацией события OnCreateHandle();
- вызывается функция Windows API Connect();
- если сервер удовлетворит запрос на соединение — вызывается обработчик события OnConnect().

Насколько успешно состоялось соединение, можно судить по свойству

```
property Connected: Boolean; //только для чтения
```

Признаком того что клиент вошел в контакт сервером, будет значение true, возвращаемое этим свойством.

> ***ЗАМЕЧАНИЕ***
>
> Единственное отличие компонента TUDPSocket от TTCPClient заключается в его конструкторе. В момент создания экземпляра класса у компонента, ориентированного на работу с протоколом пользовательских дейтаграмм UDP, изменяется тип сокета и протокола: вместо SockType=stStream, установленного у TTCPClient в классе TUDPSocket, свойство SockType принимает состояние stDgram и вместо Protocol:=IPPROTO_IP у TTCPClient для компонента TUDPSocket родным является Protocol:=IPPROTO_UDP.

Для завершения соединения применяют метод Close(). Процесс разрыва соединения сопровождается двумя последовательно возникающими событиями OnDisconnect() и OnDestroyHandle(). Вместо методов Open() и Close() можно использовать их аналоги:

```
function Connect: Boolean;      //соединение
procedure Disconnect;  //разрыв соединения
```

Результат их выполнения идентичный, так что, какой из методов-близнецов применять в своих проектах — дело личных предпочтений программиста.

Пример приложения

Рассмотрим порядок работы с сокетами на примере следующей задачи: нам требуется разработать сетевое приложение, позволяющее администратору сети наблюдать за рабочими столами компьютеров рабочей группы. Подобное приложение может оказаться полезным в школах и вузах, где на занятиях по информационным технологиям педагогу необходимо контролировать действия своих подопечных.

Решить подобную задачу можно, разработав приложение-сервер, получающее от клиентских приложений экранные снимки по транспорту WinSock.

Сокет-клиент

Каким образом научить клиентское приложение получать экранный снимок рабочего стола? На самом деле, в этом нет ничего сложного. Нам следует получить дескриптор дисплея, узнать размеры рабочего стола и с помощью функции `BitBlt()` скопировать пикселы экрана в битовый образ, созданный функцией `CreateCompatibleBitmap()`. Код функции, решающей эту задачу, едва превышает 10 строк (листинг 39.16), а в результате мы получаем дескриптор битового образа.

Листинг 39.16. Получение дескриптора битового образа со снимком рабочего стола

```
function GetScreenBitmap:HBitmap;
var memDC,DC:hDC;
    W,H:Integer;
begin
  DC:=CreateDC(PChar('DISPLAY'),nil, nil, nil);//дескриптор экрана
  memDC:=CreateCompatibleDC(DC); //совместимый дескриптор памяти

    W:=GetDeviceCaps(DC,HORZRES); //горизонтальный размер
    H:=GetDeviceCaps(DC,VERTRES); //вертикальный размер
    Result:=CreateCompatibleBitmap(DC,W,H);//создаем битовый образ
    SelectObject(memDC,Result);            //передаем образ в память
    BitBlt(memDC,0,0,W,H,DC,0,0,SRCCOPY);  //копируем экран в память
  DeleteDC(memDC); //освобождаем дескрипторы
  DeleteDC(DC);
end;
```

После этого нам осталось написать процедуру, позволяющую при посредничестве компонента `TCPClient1:TTCPClient` отправлять по заданному адресу снимок экрана. При разработке клиентской процедуры необходимо учесть тот факт, что растровая картинка современного 19-дюймового экрана с разрешением 1280×1024 пиксела с глубиной цвета 24 бит/пиксел займет более 3,5 Мбайт памяти, поэтому нам следует задуматься над решением проблемы снижения передаваемого в сеть потока данных. В листинге 39.17 для этих целей мы воспользовались услугами графического формата JPEG, который позволяет существенно сжимать растровые рисунки.

Листинг 39.17. Передача графических данных в адрес серверного сокета

```
const BufSize=262143; //размер буфера 2^18

procedure TfrmMainClient.SendScreenShot(
                          const RemoteHost, RemotePort : string);
var B:TBitmap;
    J:TJpegImage;
    MS:TMemoryStream;
    Buf:array[0..BufSize] of byte;
```

```
begin
  with TCPClient1 do
  begin
    RemoteHost:=edRemoteHost.Text;
    RemotePort:=edRemotePort.Text;
    Open;
  end;
  if TCPClient1.Connected=true then
    try
      B:=TBitmap.Create;
      B.Handle:=GetScreenBitmap; //подключаем снимок рабочего стола
      J:=TJpegImage.Create;      //с помощью JPEG сжимаем снимок
      J.Assign(B);               //забираем исходную картинку
      J.CompressionQuality:=50;  //средняя степень сжатия
      J.JPEGNeeded;              //сжимаем снимок
      MS:=TMemoryStream.Create;  //загружаем снимок в поток MS
      J.SaveToStream(ms);        //отправляем картинку в поток
      MS.Seek(0,0); //позиционируемся на первом байте потока
      MS.ReadBuffer(Buf,MS.Size); //читаем данные из потока в буфер
      TCPClient1.SendBuf(Buf,MS.Size); //отправляем снимок серверу
    finally
      B.Free; J.Free; MS.Free;
    end;
  TCPClient1.Close;
end;
```

Сокет-сервер

Для реализации приложения-сервера нам потребуются три ингредиента:

- сокет-сервер `TTCPServer`. Проконтролируйте, чтобы его свойство `BlockMode` было установлено в состояние `bmThreadBlocking`;
- компонент `Image1:TImage`, способный отобразить поступивший из сети рисунок со снимком рабочего стола;
- отдельный поток `TJPEGThread=class(TThread)`, который позволит бесконфликтно передавать поступившие от клиентских приложений снимки в `Image1`.

Реализацию сервера начнем с объявления потока `TJPEGThread`. Это мы сделаем прямо в коде программного модуля `main`, соответствующего главной форме приложения `frmMain` (листинг 39.18).

Листинг 39.18. Фрагмент модуля главной формы с объявлением потока

```
unit main;
interface

uses windows, ..., jpeg;
type
 TJPEGThread=class(TThread) //поток загрузки поступивших данных
    MS:TMemoryStream; //в память мы загрузим данные от клиента
```

```
    procedure Execute;  override;
    procedure GetJpeg; //декодирование рисунка
    constructor Create(CreateSuspended: Boolean); overload;
    destructor Destroy; override;
end;
```

Обратите внимание, что в качестве поля в класс `TJPEGThread` включен поток в памяти `MS:TMemoryStream`. В него мы загрузим поступившие от клиентского сокета данные. Помимо конструктора, деструктора и обязательного для потока метода `Execute()` объявление потока `TJPEGThread` включает процедуру `GetJpeg()`, предназначенную для загрузки клиентского экранного снимка в компонент `Image1` (листинг 39.19).

Листинг 39.19. Реализация потока `TJPEGThread`

```
constructor TJPEGThread.Create(CreateSuspended: Boolean);//конструктор
begin
  inherited  Create(CreateSuspended); //создание экземпляра TJPEGThread
  MS:=TMemoryStream.Create; // создание потока в памяти
  FreeOnTerminate:=True;    //уничтожать по завершению
end;

destructor TJPEGThread.Destroy; //деструктор потока
begin
  MS.Free;
  inherited;
end;

procedure TJPEGThread.Execute; //основной метод потока
begin
  inherited;
  Synchronize(GetJpeg); //синхронизация
  Terminate; //поток больше не нужен
end;

procedure TJPEGThread.GetJpeg; //загрузка данных в Image1
var JPG:TJPEGImage;
begin
  ms.Position:=0;
  if ms.size>0 then
    try
      JPG:=TJPEGImage.Create;
      JPG.ProgressiveDisplay:=True;
      JPG.LoadFromStream(ms);
      JPG.DIBNeeded;
      frmMain.Image1.Picture.Bitmap.Assign(JPG);
    finally
      JPG.Free;
    end;
end;
```

Для понимания работы сервера нам осталось рассмотреть событие `OnAccept()`, возникающее у компонента `TcpServer1` в момент поступления заявки на обслуживание из сети от клиентского приложения (листинг 39.20).

Листинг 39.20. Обслуживание клиентского приложения

```
const BufSize=262143; //размер приемного буфера 2^18 – 1
...
procedure TfrmMain.TcpServer1Accept(Sender: TObject;
  ClientSocket: TCustomIpClient);
var Buf:array[0..BufSize] of byte;
    x:Integer;
    JPEGThread:TJPEGThread;
begin
  JPEGThread:=TJPEGThread.Create(true); //создаем поток
                                        //в приостановленном  виде
  x:=ClientSocket.ReceiveBuf(Buf, Sizeof(Buf)); //получаем данные
  JPEGThread.MS.Write(Buf,x);       // загружаем данные в поле MS потока
  JPEGThread.Resume;                //старт потока
end;
```

На рис. 39.4 предложен экранный снимок незначительно усложненной версии приложения-сервера, которое позволяет отображать изображения рабочих столов клиентских приложений в отдельных окнах.

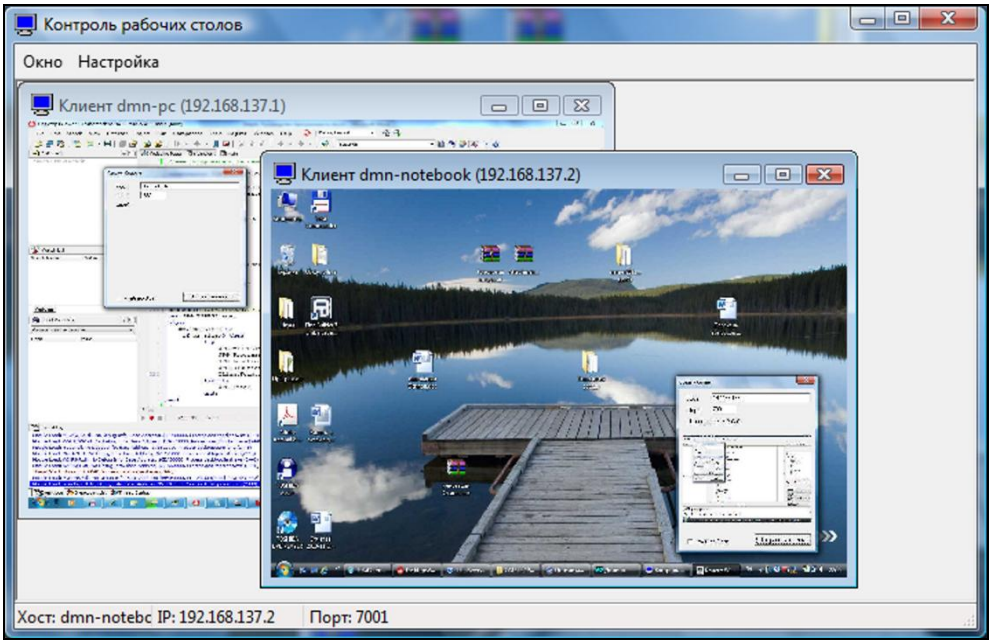


Рис. 39.4. Сбор сервером снимков рабочих столов от клиентских компьютеров

ГЛАВА 40

Сервисы Windows

В современных версиях Microsoft Windows наряду с обычными процессами трудятся незаметные обычному пользователю помощники — *сервисы* (service application) или, как их еще часто называют, *службы*. Сервис — это низкоуровневый системный процесс, выполняющий функционал по поддержке других, протекающих в операционной системе процессов. Как правило, службы не предоставляют пользователю никакого пользовательского интерфейса. Они запускаются автоматически в момент старта компьютера или вручную из консоли управления службами. Очень важен тот факт, что старт сервиса не зависит от входа пользователя в систему, поэтому службы просто незаменимы при построении различного рода работающих автономно серверов.

Для того чтобы оператор компьютера смог добраться до служб и получить возможность ими управлять, он обязан зарегистрироваться в системе с правами администратора. После этого в Панели управления Windows Vista/7 следует найти значок **Администрирование** и во вновь открывшейся папке обратиться к апплету **Службы**. В ответ на эти действия на экране компьютера появится консоль управления службами (рис. 40.1).

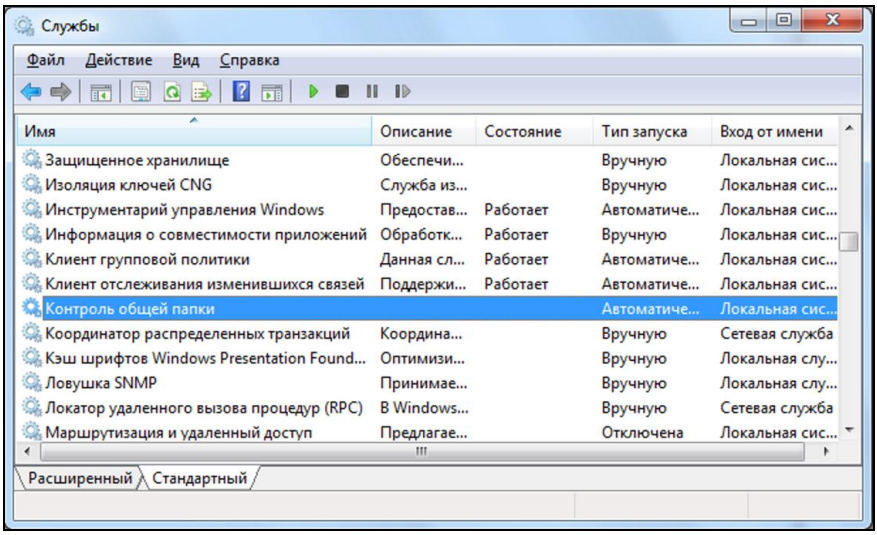


Рис. 40.1. Консоль управления службами Windows

ВНИМАНИЕ!

Не имея достаточных знаний о роли той или иной службы в жизнедеятельности локальной или удаленной станции, ни в коем случае не останавливайте ее работу! Это может негативно отразиться на работе других служб или операционной системы в целом.

Менеджер управления сервисами

В Windows работа сервисов складывается из взаимодействия трех элементов: собственно сервиса, *менеджера управления сервисами* (Service Control Manager, SCM) и *управляющего приложения* (Service Control Program, SCP).

При загрузке операционной системы Windows менеджер SCM стартует одним из первых. Сразу после собственной инициализации SCM берет в свои руки бразды контроля над службами — управляет процессом загрузки всех автоматически подключаемых служб. Во время работы операционной системы менеджер передает команды от управляющих программ (например, консоли управления сервисами) к имеющимся в его подчинении службам. К перечню основных команд относятся:

- команды инсталляции и деинсталляции служб;
- команды запуска, приостановки, возобновления выполнения или полной остановки службы;
- команды анализа состояния и изменения конфигурационных характеристик службы.

Менеджер SCM отвечает за конфигурационную базу данных служб системы, а всю информацию о зарегистрированных в операционной системе службах менеджер прячет в системном реестре Windows.

Для организации работы со службами компьютера следует получить идентификатор менеджера служб. С этой задачей превосходно справляется функция:

```
function OpenSCManager(lpMachineName, lpDatabaseName: PWideChar;
                       dwDesiredAccess: DWORD): SC_HANDLE;
```

Здесь `lpMachineName` — указатель на сетевое имя интересующего нас компьютера. Сетевое имя станции должно начинаться с двух наклонных черт, например: "\\NetComputer". Если идет речь не о сетевом компьютере, а о нашей собственной станции, то в этот параметр передаем неопределенный указатель `nil`. Второй параметр `lpDatabaseName` представляет собой указатель на строку с именем базы данных менеджера управления службами. На данный момент для инициализации этого параметра предусмотрена одна-единственная константа `SERVICES_ACTIVE_DATABASE`, и поэтому функция не обидится, если в ее второй параметр также будет направлен `nil`. Последний параметр `dwDesiredAccess` определяет наши права доступа. Наиболее грозные возможности предоставит константа `SC_MANAGER_ALL_ACCESS` — это права администратора компьютера. В случае успешного выполнения функция возвращает дескриптор менеджера служб.

После того как необходимость в услугах менеджера служб отпадает, обязательно освободите SCM. Для этого понадобится функция:

```
function CloseServiceHandle(hSCObject : THandle) : LongBool;
```

Единственный параметр метода — дескриптор менеджера SCM.

Менеджер SCM выступает точкой опоры для всех операций со службами, например для получения сведений об их состоянии. Предложенные в листинге 40.1 строки кода демонстрируют порядок решения подобных задач. В примере задействован компонент

`ListView1:TListView`. Компонент переведен в табличный режим представления данных (`ViewStyle=vsReport`), и в его свойстве `Columns` заранее подготовлены колонки (рис. 40.2) для данных о системных сервисах.

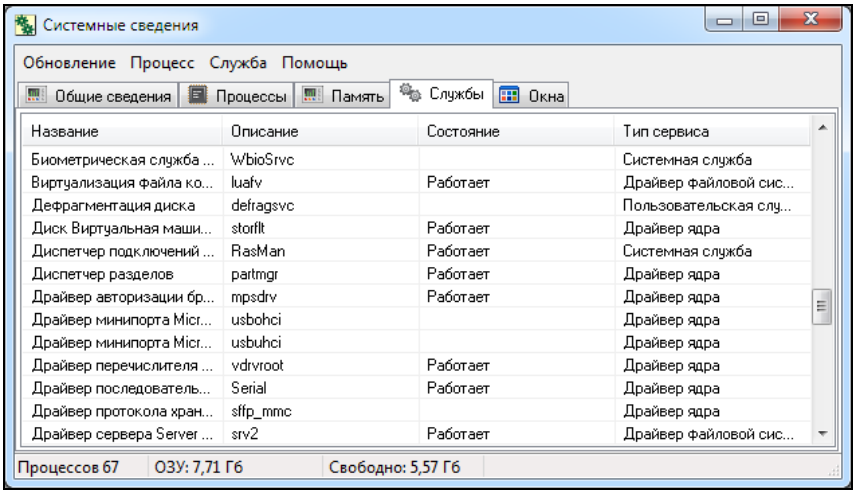


Рис. 40.2. Сбор сведений о системных службах

Листинг 40.1. Сбор сведений о доступных службах Windows

```
uses WinSvc;
...
var pServices : PEnumServiceStatus;
    BufSize,BytesNeeded,ServicesReturned,ResumeHandle : Cardinal;
    hSCM : THandle;
    LI: TListItem;
    Err : Cardinal;
    s:string;
begin
  hSCM:=OpenSCManager(nil,nil,SC_MANAGER_ALL_ACCESS);
  try
    ListView1.Items.BeginUpdate;
    ListView1.Items.Clear;
    BufSize:=SizeOf(_ENUM_SERVICE_STATUS) + 2*(255 + 1);
    GetMem(pServices, BufSize);

    ResumeHandle:=0;
    repeat
      EnumServicesStatus(hSCM, SERVICE_WIN32 OR SERVICE_DRIVER,
                         SERVICE_STATE_ALL, pServices^,BufSize,
                         BytesNeeded, ServicesReturned, ResumeHandle);
      Err:=GetLastError();
      LI:=ListView1.Items.Add;
      LI.Caption:=pServices^.lpDisplayName;
      LI.SubItems.Add(pServices^.lpServiceName);
```

```
      case pServices^.ServiceStatus.dwCurrentState of //состояние сервиса
        SERVICE_RUNNING       : LI.SubItems.Add('Работает');
        SERVICE_PAUSE_PENDING : LI.SubItems.Add('Приостановлен');
        SERVICE_STOPPED       : LI.SubItems.Add('');
      end;
    until (Err<>ERROR_MORE_DATA) OR (ResumeHandle=0);
    FreeMem (pServices, BufSize);
  finally
    ListView1.Items.EndUpdate;
    CloseServiceHandle(hSCM);
  end;
end;
```

В листинге 40.1 для сбора информации о службах была задействована функция

```
function EnumServicesStatus(hSCManager: SC_HANDLE;
                            dwServiceType, dwServiceState: DWORD;
                            var lpServices: TEnumServiceStatus;
                            cbBufSize: DWORD;
                            var pcbBytesNeeded, lpServicesReturned,
                            lpResumeHandle: DWORD): BOOL;
```

Первый параметр функции `hSCManager` должен содержать ссылку на дескриптор менеджера SCM. Аргумент `dwServiceType` конкретизирует, что именно подлежит сбору, здесь совместно могут применяться два флага — `SERVICE_DRIVER` (драйверы) и `SERVICE_WIN32` (службы). Третий параметр `dwServiceState` еще более сужает область поиска: флаг `SERVICE_ACTIVE` указывает, что нас интересуют только активные службы; `SERVICE_INACTIVE` — остановленные службы; `SERVICE_STATE_ALL` — все службы. Оставшиеся параметры предназначены для возврата результатов выполнения функции, с ними вы сможете познакомиться в справочной системе.

Управление сервисом

Для доступа к установленному в операционной системе сервису необходимо получить его дескриптор. Для этого предназначена функция

```
function OpenService (SCManager : THandle;
                      ServiceName : pChar;
                      DesiredAccess : Cardinal): THandle;
```

Здесь: `SCManager` — полученный усилиями функции `OpenSCManager()` дескриптор менеджера служб; `ServiceName` — указатель на текстовую строку с именем сервиса; `DesiredAccess` — желательные права доступа к сервису, наибольшие возможности предоставляет права администратора `SC_MANAGER_ALL_ACCESS`. При удачном стечении обстоятельств функция вернет дескриптор затребованной нами службы.

Для запуска службы из приложения SCP используют функцию

```
function StartService(Service : THandle;
                      NumServiceArgs : Cardinal;
                      var ServiceArgVectors : pChar) : LongBool;
```

Первый параметр функции определяет указатель интересующего нас сервиса, второй параметр `NumServiceArgs` — количество строк в массиве аргументов, на который указывает тре-

тий по счету параметр ServiceArgVectors. В простейшем случае (листинге 40.2) для запуска сервиса нам понадобится только его дескриптор.

Листинг 40.2. Запуск службы

```
procedure StartService(const SrvName : string);
var hSCM, hSrv : THandle;
    SStatus:_SERVICE_STATUS;
    args: pChar;
begin
  hSCM:=OpenSCManager(nil,nil,SC_MANAGER_ALL_ACCESS);
  hSrv:=OpenService(hSCM,SrvName,SC_MANAGER_ALL_ACCESS);
  if hSrv>0 then StartService(hSrv, 0, args);
  CloseServiceHandle(hSrv);
  CloseServiceHandle(hSCM);
end;
```

Обладая дескриптором сервиса, приложение вправе выдать команду на его приостановку, возобновление и завершение работы. Для этого применяется функция

```
function ControlService(Service : THandle;
                        Control : Cardinal;
                        var Service_Status : _SERVICE_STATUS) : LongBool;
```

Первый параметр метода — дескриптор сервиса. Параметр Control определяет наши требования к поведению службы: значение SERVICE_CONTROL_STOP остановит выполнение службы; SERVICE_CONTROL_PAUSE — временная остановка; SERVICE_CONTROL_CONTINUE запустит сервис на выполнение. Последний выходной параметр Service_Status ссылается на объявленную в модуле WinSvc структуру _SERVICE_STATUS, хранящую основные характеристики службы. В случае корректного завершения функция возвратит значение true.

Для определения текущего состояния службы стоит воспользоваться услугами функции

```
function QueryServiceStatus(Service : THandle;
                        var Service_Status : _SERVICE_STATUS) : LongBool;
```

На вход функции подается дескриптор исследуемой службы, а вся информация о ней возвращается в структуру Service_Status (листинг 40.3).

Листинг 40.3. Состав структуры _SERVICE_STATUS

```
type _SERVICE_STATUS = record
   dwServiceType: DWORD;      //тип исполняемого модуля сервиса
   dwCurrentState: DWORD;     //текущее состояние сервиса
   dwControlsAccepted: DWORD; //управляющие коды сервиса
   dwWin32ExitCode: DWORD;    //код ошибки старта/останова сервиса
   dwServiceSpecificExitCode: DWORD;//спецификатор ошибки
   dwCheckPoint: DWORD;//значение контрольной точки, необходимое для
                  //того, чтобы служба могла уведомлять о своей работе
   dwWaitHint: DWORD;      //время, необходимое для изменения состояния
 end;
```

Состояние службы

Для получения сведений о конфигурации службы программист может воспользоваться функцией

```
function QueryServiceConfig(Service : THandle;
                 ServiceConfig : PQueryServiceConfig; BufSize : Cardinal;
                 var BytesNeeded : Cardinal) : LongBool;
```

В функцию передается дескриптор сервиса `Service`, указатель на приемный буфер (структуру `ServiceConfig`), в который будут помещены результаты работы функции. Все поля информационной структуры `_QUERY_SERVICE_CONFIG` прокомментированы в листинге 40.4. Размер буфера определяется параметром `BufSize`. Последний выходной параметр возвращает пожелания функции по поводу размера буфера.

Листинг 40.4. Состав структуры _QUERY_SERVICE_CONFIG

```
type   _QUERY_SERVICE_CONFIG = record
    dwServiceType: DWORD;      //тип сервиса
    dwStartType: DWORD;        //тип запуска
    dwErrorControl: DWORD;     //реакция на ошибки при загрузке системы
    lpBinaryPathName: PChar;   //путь к исполняемому файлу
    lpLoadOrderGroup: PChar;   //группа загрузки
    dwTagId: Cardinal;         //номер запуска в группе
    lpDependencies: PChar;     //зависимые службы
    lpServiceStartName: PChar;//учетная запись
    lpDisplayName: PChar;      //отображаемое на экране название
  end;
```

Пример, демонстрирующий способ сбора сведений о сервисе, представлен в листинге 40.5. На вход разработанной нами процедуры следует передать дескриптор менеджера SCM и имя сервиса.

Листинг 40.5. Получение сведений о сервисе

```
procedure GetServiceInfo(SCManager :THandle; SrvName : string);
var pConfig     : PQueryServiceConfig;
    BytesNeeded : Cardinal;
    Srv         : THandle;
begin
  Srv:=OpenService(SCManager, SrvName, SC_MANAGER_ALL_ACCESS);
  {первый вызов функции для выяснения необходимого размера входного буфера}
  QueryServiceConfig(Srv, nil, 0, BytesNeeded);
  GetMem(pConfig,BytesNeeded);//инициализация входного буфера
  {второй вызов функции с получением данных}
  QueryServiceConfig(Srv, pConfig, BytesNeeded, BytesNeeded);
  ...
  //АНАЛИЗ ПОЛУЧЕННЫХ ДАННЫХ
  ...
  FreeMem(pConfig,BytesNeeded); //очистка памяти
  CloseServiceHandle(Srv);      //освобождение дескриптора сервиса
end;
```

Обратите внимание на то, что в листинге 40.5 функция `QueryServiceConfig()` вызывается дважды. Первый вызов необходим для выяснения размера входного буфера. Реальные данные о сервисе с именем `SrvName` будут возвращены во время повторного вызова.

Конфигурирование службы

Для изменения основных параметров службы предназначена функция:

```
function ChangeServiceConfig( Service : THandle; //дескриптор сервиса
                dwServiceType,          //тип сервиса
                dwStartType,            //особенности запуска
                dwErrorControl : Cardinal;//реакция на ошибки
                lpBinaryPathName,       //путь к исполняемому файлу
                lpLoadOrderGroup : pChar;//группа загрузки
                lpdwTagId : cardinal;  //номер в группе
                lpDependencies,         //зависимые службы
                lpServiceStartName,     //учетная запись
                lpPassword,             //пароль к учетной записи
                lpDisplayName : pChar  //выводимое на экран название
            ) : LongBool;
```

Отметим, что, если планируется изменить не все, а лишь некоторые из конфигурационных параметров функции в неизменяемые параметры, надо передавать константу `SERVICE_NO_CHANGE`.

Удаление службы

Вся ирония программирования в том, что создаваемый долгими бессонными ночами сервис-шедевр удаляется из системы одной строкой кода:

```
function DeleteService(Service : THandle) : LongBool;
```

Единственный параметр функции — дескриптор неактуальной службы.

Сервис в VCL, класс *TService*

В библиотеке VCL сервис представлен в виде класса `TService` и описывается в программном модуле `SvcMgr`. При выборе предка `TService` создатели Delphi воспользовались услугами модуля данных `TDataModule`, специализирующегося на хранении невизуальных компонентов. Благодаря такому решению значительно упростился процесс программирования, в буквальном смысле службу можно "потрогать" руками и даже отдать в ее распоряжение ряд компонентов.

Идентификация

Ключевое свойство, по которому будет идентифицироваться служба в системе, — ее имя:

```
property Name: TComponentName;
```

Имя сервиса нам пригодится при создании сервиса средствами Windows API (функция `CreateService()`) или при выяснении дескриптора (функция `OpenService()`).

Кроме имени, у службы может быть заголовок:

```
property DisplayName: String;
```

Заголовок будет отображаться в колонке "Имя" консоли управления службами.

Тип сервиса

Тип сервиса определяется свойством

```
property ServiceType: TServiceType;//по умолчанию stWin32
```

Различают три типа службы: `stWin32` — пользовательский или системный сервис; `stDevice` — драйвер устройства; `stFileSystem` — драйвер файловой системы.

Для совместимости со старыми разработками сохранено свойство, на сегодня уже не актуальное

```
property Interactive: Boolean; //по умолчанию false
```

Это свойство позволяло пользователю взаимодействовать со службой с помощью GUI. Поддержка интерактивности имела смысл в ситуации, когда мы наделяли сервис пользовательским интерфейсом, а это в современных версиях Windows проблематично.

> ***Внимание!***
>
> Интерактивность служб поддерживалась только до Windows XP включительно, в более поздних ОС из соображений безопасности Microsoft запретила интерактивные сервисы. Если вашему сервису обязательно нужен доступ к GUI, то наиболее простой выход из положения будет связан с разработкой отдельного процесса-помощника, который будет запускаться вместе с сервисом и взаимодействовать с ним при посредничестве любой известной вам технологии (анонимных каналов, объектов файлового отображения и т. п.).

Определение прав на управление сервисом

Запуск сервиса всегда осуществляется от имени какой-либо зарегистрированной в системе учетной записи. Если все оставить на самотек, то служба стартует от имени системной учетной записи `LocalSystem`. Если же есть специальные требования к политике безопасности, то следует поэкспериментировать со свойством

```
property ServiceStartName: String; //по умолчанию пусто
```

определяющим имя учетной записи. При запуске сервиса от имени локального пользователя формат имени должен выглядеть так: "`.\<ИМЯ>`"; для доменного — "`<ДОМЕН>\<ИМЯ>`".

Задавая учетную запись для сервиса типа `stWin32`, требуется указать пароль

```
property Password: String;
```

Но будьте осторожны, потому что пароль сохраняется в реестре, а его корректность проверяется только в момент запуска сервиса. Специальные учетные записи (`LocalSystem`, `LocalService` и `NetworkService`) пароля не имеют.

Загрузка и запуск службы

Корректность функционирования службы состоит в прямой зависимости от того, запущены или нет обеспечивающие ее сервисы и загружены ли необходимые драйверы. Если основная служба не загружена или остановлена, то это, безусловно, отразится и на выполнении зависимого от нее сервиса. Менеджер служб всегда отслеживает такие зависимости.

При определении сервиса необходимо указать список служб, от которых зависит сервис. Для этого предназначено свойство-контейнер

```
property Dependencies: TDependencies;
```

Доступ к элементу коллекции (экземпляру класса `TDependency`) осуществляется с помощью свойства

```
property Items[Index: Integer]: TDependency; default;
```

При наличии явной зависимости между разрабатываемой нами службой и другими сервисами особую роль играет строгая очередность загрузки взаимосвязанных сервисов. Для определения последовательности загрузки служб и системных драйверов в Windows введено понятие групп загрузки. При разработке службы программист указывает, в составе какой из групп следует загружать его сервис. Название группы определяется в свойстве:

```
property LoadGroup: String;
```

Список групп загрузки мы найдем в системном реестре по адресу:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder
```

Перечень включает несколько десятков наименований. При необходимости программист имеет право определить в реестре собственную группу загрузки.

Службы из каждой группы стартуют только после того, как запустятся все автоматически стартующие службы из предыдущей группы. Сервисы, которые не принадлежат ни к одной из групп загрузки, запускаются в последнюю очередь.

Для того чтобы упорядочить очередность загрузки сервиса внутри группы загрузки, его порядковый номер передается в свойство:

```
property TagID: Cardinal; //по умолчанию 0
```

Это свойство используется только при загрузке служб ядра. Если загрузочный номер неизвестен, то оставляйте в свойстве нулевое значение.

Особенности запуска службы определяются состоянием свойства:

```
property StartType: TStartType; //по умолчанию stAuto
type TStartType = (stBoot,      //запуск во время загрузки ОС
                   stSystem,    //запуск после загрузки ОС
                   stAuto,      //автоматический запуск
                   stManual,    //служба стартует вручную
                   stDisabled); //служба отключена
```

Особенности старта сервиса могут быть определены в дополнительных параметрах запуска. Для обращения к отдельному параметру запуска направьте его индекс в массив:

```
property Param[Index: Integer]: String;//только для чтения
```

Статус службы

О текущем состоянии службы нас проинформирует свойство

```
property Status: TCurrentStatus;
type  TCurrentStatus = (csStopped,        //сервис остановлен
                        csStartPending,   //сервис стартует
                        csStopPending,    //выполнение прекращается
                        csRunning,        //сервис выполняется
                        csContinuePending,//ожидание продолжения
                        csPausePending,   //ожидание приостановки
                        csPaused);        //сервис приостановлен
```

Свойство Status не только информационное, с его помощью программист имеет право управлять статусом службы из кода проекта. Любое изменение состояния службы согласуется с менеджером служб SCM. Для этого автоматически вызывается метод

```
procedure ReportStatus;
```

В теле этой процедуры заполняется структура TServiceStatus, содержащая несколько информационных полей, таких как тип, текущее состояние службы, код ошибки и т. п. Затем вся эта информация и предложение по смене состояния службы направляются менеджеру при помощи функции SetServiceStatus(). Если с нашими предложениями менеджер согласится, то служба переводится в запрашиваемое состояние.

Сбои при старте сервиса

Реакцию операционной системы на сбой в момент старта службы определяет свойство

```
property ErrorSeverity: TErrorSeverity; //по умолчанию esNormal
```

Четыре возможных варианта поведения Windows представлены в табл. 40.1.

Таблица 40.1. *Значения* `TErrorSeverity`

Значение	Поведение
esIgnore	Игнорировать ошибку. Данные об ошибке заносятся в журнал, служба продолжает выполняться
esNormal	Ошибка заносится в журнал. На экран выводится сообщение об ошибке, после чего служба продолжит свою работу
esSevere	Ошибка заносится в журнал. Выполнение сервиса продолжится только в том случае, если система сможет загрузиться в режиме "последняя удачная конфигурация"
esCritical	Критическая ошибка. Если происходит отказ этой службы, ошибка заносится в журнал и система перезагружается в режиме "последняя удачная конфигурация"

Остановка и возобновление службы

Не каждая служба может быть остановлена или приостановлена, например, такой возможностью не обладают системные драйверы. В роли индикатора того, что служба способна "взять паузу", выступает свойство

```
property AllowPause: Boolean;
```

Перед приостановкой службы вызывается событие OnPause(), а после возобновления — OnContinue().

> ***Замечание***
>
> Напомню, что проще всего приостановить или возобновить выполнение сервиса с помощью функции ControlService().

Еще одно свойство, с помощью которого проверяется, сможем ли мы остановить сервис:

```
property AllowStop: Boolean;
```

Если это свойство установлено в true, то перед остановкой службы вызывается событие OnStop(). Весь перечень событий, связанных с запуском и остановкой службы, представлен в табл. 40.2.

Таблица 40.2. *События запуска и остановки сервиса*

Событие	Описание
`property OnStart: TStartEvent; type TStartEvent = procedure(Sender: TService; var Started: Boolean) of object;`	Событием `OnStart()` сопровождается первый старт службы. В этот момент осуществляется инициализация сервиса
`property OnPause: TPauseEvent; type TPauseEvent = procedure(Sender: TService; var Paused: Boolean) of object;`	Генерируется в момент приостановки службы. Управляя параметром `Paused`, программист сможет запретить (`false`) или разрешить (`true`) приостановку службы
`property OnContinue: TContinueEvent; type TContinueEvent = procedure(Sender: TService; var Continued: Boolean) of object;`	Возобновление работы сервиса. Если мы не имеем ничего против возобновления выполнения службы, то передаем параметру `Continued` значение `true`
`property OnStop: TStopEvent; TStopEvent = procedure(Sender: TService; var Stopped: Boolean) of object;`	Событие вызывается в тот момент, когда менеджер служб завершает работу сервиса
`property OnShutdown: TServiceEvent;`	Событие генерируется в момент завершения работы операционной системы. Здесь мы можем описать действия, предшествующие принудительному разрушению сервиса

Инсталляция и деинсталляция сервиса

В роли инициатора регистрации службы может выступить внешнее приложение или владеющий этим сервисом процесс. Сразу отмечу, что это особый вид приложения, не использующий привычный для нас класс `TApplication`. Фундамент приложения-службы составляет специализированный класс `TServiceApplication`. С этим классом и процессом разработки приложений такого рода мы познакомимся несколько позднее, а пока (табл. 40.3) рассмотрим, каким образом класса `TService` реагирует на установку (и удаление) службы.

Таблица 40.3. *События установки и удаления сервиса*

Событие	Описание
`property BeforeInstall: TServiceEvent;` `type TServiceEvent = procedure (Sender: TService) of object;`	Начало инсталляции службы. Единственный параметр события `Sender` содержит ссылку на сервис
`property AfterInstall: TServiceEvent;`	Завершение инсталляции службы
`property BeforeUninstall: TServiceEvent;`	Начало деинсталляции службы
`property AfterUninstall: TServiceEvent;`	Деинсталляция службы успешно завершилась

ЗАМЕЧАНИЕ

Весь перечень установленных на компьютере служб регистрируется в системном реестре Windows. Информацию об имеющихся службах мы обнаружим в ветви `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Servicies`.

Выполнение службы, поток *TServiceThread*

В Windows каждая служба выполняется в отдельном потоке. Кроме того, для организации взаимодействия службы `TService` с менеджером служб для каждого экземпляра службы Delphi создает новый поток (экземпляр класса `TServiceThread`). Ссылка на экземпляр потока хранится в свойстве службы

```
property ServiceThread: TServiceThread;
```

Поток `TServiceThread` является прямым наследником классического потока `TThread`, его основное отличие от предка в наличии процедуры

```
procedure ProcessRequests(WaitForMessage: Boolean);
```

Благодаря `ProcessRequests()` сервис взаимодействует с менеджером служб Windows. В рамках метода, взаимодействующего со службой, поток отслеживает пять управляющих сообщений, поступающих от менеджера SCM :

- `SERVICE_CONTROL_STOP` — остановить сервис;
- `SERVICE_CONTROL_PAUSE` — приостановить сервис;
- `SERVICE_CONTROL_CONTINUE` — продолжить сервис;
- `SERVICE_CONTROL_SHUTDOWN` — уничтожение сервиса;
- `SERVICE_CONTROL_INTERROGATE` — запрос о состоянии сервиса.

Таким образом, с помощью `ProcessRequests()` служба поддерживает постоянный контакт с менеджером служб Windows. Единственный параметр `WaitForMessage` процедуры переводит ее в синхронный (`true`) или асинхронный (`false`) режим работы. При выполнении процедуры в синхронном режиме поток станет ждать управляющей команды от SCM и соответственно приостановит выполнение службы до поступления этой команды. В асинхронном режиме обращение к `ProcessRequests()` не останавливает службу. А теперь посмотрим, каким образом служба `TService` сможет воспользоваться услугами этого метода, а для этого мы познакомимся с центральным событием, связанным с выполнением сервисом своих прямых обязанностей

```
property OnExecute: TServiceEvent;
```

Это именно то событие, в рамках которого описывается основной исполняемый код службы. Если в функциональные обязанности службы входит постоянное обслуживание какого-то устройства, непрерывный сбор информации или что-то подобное, то внутри `OnExecute()` реализуют цикл, выход из которого производится по команде менеджера SCM (листинг 40.6).

Листинг 40.6. Управление взаимодействием сервиса с SCM

```
procedure TDemoService1.ServiceExecute(Sender: TService);
begin
  repeat
  //основной код службы
  //...
  ServiceThread.ProcessRequests(False);  //обращение к SCM за дальнейшими
                                         //указаниями
  until (terminated=true);
end;
```

Цикл выполняется до тех пор, пока свойство

```
property Terminated: Boolean;
```

не перейдет в состояние `true` или его не остановит менеджер служб Windows. Свойство `Terminated` уведомляет программиста о том, что выполнение службы завершается, и она должна быть уничтожена.

Ведение протокола службы

Любая профессионально разработанная служба обязана уметь общаться с журналом событий операционной системы. Благодаря записям журнала администратор компьютера получает возможность контролировать корректность работы системы и при необходимости вмешиваться в деятельность того или иного процесса или службы. Каждая строка журнала включает информацию об имени службы, типе события, времени его возникновения и т. п.

Код ошибки, возвращаемый в момент некорректной остановки или при неудачном запуске сервиса, содержится в свойстве:

```
property ErrCode: DWord;
```

Если свойство `ErrCode` не определено, то код ошибки будет взят из свойства

```
property Win32ErrCode: DWord;
```

И наконец, самый последний вариант развития событий, когда код ошибки не задан ни в одном из перечисленных свойств, в свойство `Win32ErrCode` передается константа `ERROR_SERVICE_SPECIFIC_ERROR`.

О факте старта, остановки, приостановки, возобновления сервиса менеджер служб информируется автоматически без вмешательства программиста. Продолжительность отрезка времени между моментами запуска (приостановки, остановки) службы и отправки сообщения в адрес менеджера служб компьютера определяется свойством:

```
property WaitHint: Integer; //по умолчанию 5000 миллисекунд
```

Как правило, для взаимодействия с функциями Windows API установленного по умолчанию значения в 5 секунд вполне достаточно, но если служба выполняет длительную операцию, то, вызвав `ReportStatus()`, целесообразно принудительно проинформировать менеджера о состоянии службы.

Для добавления в журнал событий сведений о жизнедеятельности сервиса применяют процедуру

```
procedure LogMessage(Message: String; EventType: DWord = 1;
                     Category: Integer = 0; ID: Integer = 0);
```

Здесь: `Message` — текстовое содержание сообщения; `EventType` — тип события (табл. 40.4); категория события и его идентификационный номер определяются значениями, переданными в параметры `Category` и `ID`.

Таблица 40.4. *Тип события* `EventType`

Константа	Описание типа события
EVENTLOG_ERROR_TYPE	Произошла ошибка
EVENTLOG_WARNING_TYPE	Предупреждение о потенциальной опасности

Таблица 40.4 *(окончание)*

Константа	Описание типа события
EVENTLOG_INFORMATION_TYPE	Обычное информационное уведомление
EVENTLOG_AUDIT_SUCCESS	Аудит завершился успешно
EVENTLOG_AUDIT_FAILURE	Ошибка аудита

Приложение-сервис *TServiceApplication*

Для инкапсуляции возможностей приложения-службы в визуальной библиотеке компонентов Delphi разработан класс `TServiceApplication`. Для создания нового сервисного приложения необходимо воспользоваться пунктом меню **File | New | Other** и на странице **New** диалогового окна **New Items** выбрать пункт **Service Application**.

В результате создается шаблон кода, в котором объявляется глобальный объект `Application` типа `TServiceApplication`. Кроме того, в проекте декларируется инкапсулирующий службу объект `Service1:TService1`. При необходимости мы можем добавить в приложение дополнительные сервисы. Для этого надо вновь воспользоваться пунктом главного меню **File | New | Other** и в окне **New Items** выбрать значок **Service**.

Приложение `TServiceApplication` лишь играет роль оболочки для входящих в его состав сервисов и поэтому практически не обладает индивидуальными свойствами и методами. Из имеющихся в нашем распоряжении свойств интерес представляет свойство

```
property ServiceCount: Integer; //только для чтения
```

информирующее программиста о количестве сервисов (экземпляров класса `TService`).

Метод

```
procedure RegisterServices(Install, Silent: Boolean);
```

позволяет зарегистрировать (`Install=true`) в системе или снять с регистрации (`Install=false`) все принадлежащие приложению сервисы. Установив параметр `Silent` в состояние `true`, мы добьемся "тихой" работы метода — на экран компьютера не будет выведено сообщение о завершении операции регистрации.

Пример

Если вам когда-нибудь приходилось администрировать файловые серверы, предназначенные для сбора документов со всего предприятия, то наверняка вы согласитесь, что одной из самых рутинных задач является сортировка поступающих из сети файлов. Теперь, когда мы получили представление о сервисах, мы сможем существенно упростить решение подобных задач.

Допустим, что у нас имеется предоставленная в общий доступ сетевая папка, в которую поступают различные документы со всех подразделений предприятия. Наша задача заключается в разработке сервиса Windows, отслеживающего все попадающие в эту папку файлы и распределяющего их по другим папкам по какому-то классификационному признаку, например по типу файла.

Создадим новое приложение-сервис (Service Application) и приступим к его конфигурированию. Для этого выберем модуль службы с именем по умолчанию `Service1` и внесем ряд изменений в его свойства:

- тип сервиса `ServiceType:=stWin32`;
- имя сервиса `Name:=FolderControl`;
- отображаемое имя `DisplayName:='Контроль общей папки'`;
- тип запуска `StartType:=stManual`;
- реакция на ошибки при запуске `ErrorSeverity:=esIgnory`.

Сохраните проект в отдельном каталоге — модуль службы назовите FolderControl_Unit.pas, а весь проект — fldrsrv.dproj.

Объявите ряд переменных, необходимых для работы сервиса (листинг 40.7).

Листинг 40.7. Глобальные переменные службы

```
var
  FolderControl: TFolderControl;
  fldrINPUT,fldrDOC,fldrPICTURE,fldrOTHER,PICTURE_EXT,DOC_EXT: string;

implementation
  uses registry; //модуль для взаимодействия с реестром
```

Переменные `fldr...` хранят пути к папкам, контролируемым сервисом, в первую очередь это общедоступная системная папка (переменная `fldrINPUT`), в которую будут поступать файлы из сети. В переменные `PICTURE_EXT` и `DOC_EXT` мы позднее занесем перечень наиболее востребованных расширений имен файлов.

В момент старта сервиса обращаемся к системному реестру и читаем из него параметры сервиса (имена каталогов и перечень расширений имен файлов). Затем обязательно проконтролируем факт существования папок и, если они по какой-то причине отсутствуют, создадим их (листинг 40.8).

Листинг 40.8. Событие старта сервиса

```
procedure TFolderControl.ServiceStart(Sender: TService; var Started:
   Boolean);
var R:TRegIniFile;
const REG_KEY='\SYSTEM\CurrentControlSet\services\FolderControl';
      REG_SECTION='PARAMETERS';
begin
  R:=TRegIniFile.Create('');
  R.RootKey:=HKEY_LOCAL_MACHINE;
  R.OpenKey(REG_KEY,true);
  fldrINPUT  :=R.ReadString(REG_SECTION,'INPUT',  'c:\inputfolder\');
  fldrDOC    :=R.ReadString(REG_SECTION,'DOC',    'c:\doc\');
  fldrPICTURE:=R.ReadString(REG_SECTION,'PICTURE','c:\picture\');
  fldrOTHER  :=R.ReadString(REG_SECTION,'OTHER',  'c:\other\');
```

```
  PICTURE_EXT:=R.ReadString(REG_SECTION,'PICTURE_EXT',
                  '.bmp .gif .tif .tiff .jpg .jpeg .png .wmf .emf');
  DOC_EXT    :=R.ReadString(REG_SECTION,'DOC_EXT',
                  '.doc .rtf .docx .docm .txt');
  R.Free;

  try
    if NOT DirectoryExists(fldrINPUT)
                        then CreateDirectory(pChar(fldrINPUT),nil);
    if NOT DirectoryExists(fldrDOC)
                        then CreateDirectory(pChar(fldrDOC),nil);
    if NOT DirectoryExists(fldrPICTURE)
                        then CreateDirectory(pChar(fldrPICTURE),nil);
    if NOT DirectoryExists(fldrOTHER)
                        then CreateDirectory(pChar(fldrOTHER),nil);
    Started:=true;
  except
    Started:=false;
  end;
end;
```

Отметим, что запрашиваемая в листинге 40.8 секция системного реестра пока отсутствует, поэтому в глобальные переменные поступают значения по умолчанию. Конечно, администратор сервера сможет заполнить реестр вручную, это гораздо лучше, чем каждый раз (при необходимости изменить имена папок или расширения имен файлов) требовать от программиста переписать сервис.

Нам осталось сделать самое важное — описать событие `OnExecute()` сервиса (листинг 40.9). В рамках этого события мы контролируем состояние папки, описываемой константой `fldrINPUT`. Структурно событие представляет собой два вложенных цикла `repeat..until`. Задача внешнего цикла — не пропустить момент поступления в папку любого файла, в этом ему поможет входящая в состав Windows API функция `FindFirstChangeNotification()`. По факту поступления файла активируется вложенный цикл, здесь с помощью функций `FindFirst()` и `FindNext()` мы собираем сведения о появившихся в каталоге файлах и распределяем их по другим папкам.

Листинг 40.9. Выполнение сервиса

```
procedure TFolderControl.ServiceExecute(Sender: TService);
var h:THandle;
    c:cardinal;
    F:TSearchRec;
    FileExt: string;
begin
  h:=FindFirstChangeNotification(pCHAR(fldrINPUT),True,
         FILE_NOTIFY_CHANGE_FILE_NAME OR FILE_NOTIFY_CHANGE_DIR_NAME);
  repeat
    c:=WaitForSingleObject(h,5000);
    if c=WAIT_OBJECT_0 then  //в папке произошли изменения
    //=================================================================
      try
        if FindFirst(fldrINPUT+'*.*',faAnyFile,F)=0 then
```

```
        repeat
          if (F.Attr and faDirectory) = 0 then
          begin
            FileExt:=lowercase(ExtractFileExt(F.Name));
            if Pos(FileExt,PICTURE_EXT)>0 then //это графический файл
            MoveFile(pCHAR(fldrINPUT+F.Name),pCHAR(fldrPICTURE+F.Name))
            else
              if Pos(FileExt,DOC_EXT)>0 then   //это документ
               MoveFile(pCHAR(fldrINPUT+F.Name),pCHAR(fldrDOC+F.Name))
              else
              MoveFile(pCHAR(fldrINPUT+F.Name),pCHAR(fldrOTHER+F.Name));
          end;
        until FindNext(F)<>0;
      finally
        FindClose(f);
      end;
    //=================================================================
    ServiceThread.ProcessRequests(False);
 until (terminated=true);
end;
```

Отметим, что листинг 40.9 представляет собой лишь самое общее решение задачи распределения файлов. Например, в нем не предусмотрены действия при совпадении имен файлов. Тем не менее для наших первых шагов на поприще разработки служб Windows мы сделали уже много. Будем полагать, что проект завершен. Откомпилируйте его. Теперь нам необходимо зарегистрировать сервис в SCM.

Регистрация сервиса в ручном режиме

Для регистрации службы в операционной системе владеющее службой приложение должно быть запущено из командной строки с ключом `/INSTALL`. Например:

```
C:\DemoService\fldrsrv.exe /install
```

Для снятия с регистрации применяют ключ `/UNINSTALL`. Процесс установки сервиса сопровождается выводом уведомляющего сообщения. Для отказа от показа окна уведомления используйте ключ `/SILENT`.

> ***ЗАМЕЧАНИЕ***
>
> Сведения о зарегистрированных сервисах хранятся в разделе системного реестра `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\<ИМЯ_СЛУЖБЫ>`.

Откройте консоль управления службами компьютера, найдите в ней наш сервис "Контроль общей папки" и запустите его на выполнение (см. рис. 40.1). Поместите в контролируемую папку любые файлы. Если вы безошибочно повторили код, то сразу получите результат — служба распределит эти файлы по соответствующим папкам.

Отмечу, что отладка службы — весьма непростое занятие по двум причинам: во-первых, службой управляет менеджер SCM, который весьма капризен по отношению внешних вмешательств; во-вторых, приложение-служба как минимум обладает двумя потоками. Если же ваше приложение-служба инкапсулирует несколько сервисов, то сложность отладки возрастает в геометрической прогрессии в соответствии с количеством интегрируемых служб.

ЗАМЕЧАНИЕ

Лучшая рекомендация по отладке служб звучит так: реализуйте весь функционал службы в обычном приложении. Выявите и устраните ошибки стандартным образом в отладчике Delphi и только затем перенесите апробированный код в приложение-службу.

Апплеты Панели управления

Даже начинающий пользователь Windows знает о существовании Панели управления (Control Panel). Панель управления предоставляет удобный доступ к глобальным настройкам операционной системы, ее системным компонентам и обслуживаемых этой ОС устройствам. Для этого оператору компьютера необходимо найти интересующий его значок и дважды по нему щелкнуть левой кнопкой мыши. В результате инициализируется специализированная динамическая библиотека, называемая *апплетом Панели управления* (Control Panel Applet). Апплет управляет единственной подпрограммой, обеспечивая пользовательский контроль параметров настройки эксплуатационных режимов отдельного системного компонента Windows.

Помимо классического способа старта апплета двойным щелчком по значку в Windows предусмотрены альтернативные методы запуска апплета — из командной строки или посредством вызова функций `WinExec()` и `ShellExec()`. Первая функция предназначена для выполнения простейших апплетов (листинг 40.10).

Листинг 40.10. Старт апплета с помощью функции `WinExec()`

```
WinExec('control.exe /name Microsoft.AddHardware', SW_NORMAL);
```

Функция `ShellExec()` востребована в случае, когда при старте апплета требуется передать дополнительные параметры (листинг 40.11).

Листинг 40.11. Старт апплета с помощью функции `ShellExec()`

```
ShellExecute(Application.Handle, 'Open', 'control.exe',
             'desk.cpl, screensaver, @screensaver', '', SW_NORMAL);
```

В листинге 40.10 `control.exe` — название приложения контейнера апплетов, а `AddHardware` — название интересующего нас апплета. Перечень основных апплетов Панели управления Windows представлен в табл. 40.5.

Таблица 40.5. *Стандартные апплеты Windows*

Апплет	Описание
`control.exe desktop`	Вывод окна **Свойства: экран**
`control.exe color`	Вывод окна **Свойства: экран** с активной страницей **Заставка**
`control.exe date/time`	Вывод окна **Свойства: дата и время**
`control.exe international`	Вывод окна **Язык и региональные стандарты**
`control.exe mouse`	Вывод окна **Свойства: мышь**
`control.exe keyboard`	Вывод окна **Свойства: клавиатура**

Таблица 40.5 *(окончание)*

Апплет	Описание
`control.exe printers`	Показ папки **Принтеры и факсы**
`control.exe fonts`	Показ папки **Шрифты**
`control.exe folders`	Вывод окна **Свойства папки**
`control.exe telephony`	Вывод окна **Телефон и модем**
`control.exe admintools`	Вывод окна **Администрирование**
`control.exe schedtasks`	Вывод окна **Назначенные задания**
`control.exe netconnections`	Показ папки **Сетевые подключения**
`control.exe userpasswords`	Вывод окна **Учетные записи пользователей**

Апплет Панели управления, класс *TAppletModule*

В библиотеке VCL функционал апплета Панели управления инкапсулирован в модуле `CtlPanel` в виде класса `TAppletModule`. В иерархии наследования классов апплет следует сразу за модулем данных `TDataModule`. Таким образом, значительно упрощается интеграция в апплет невизуальных компонентов Delphi. Свойств, методов и событий у апплета совсем немного.

Заголовок модуля определяется свойством

```
property Caption: string;
```

Связанный с апплетом значок назначается в свойстве

```
property AppletIcon: TIcon;
```

Строка с текстом комментария заносится в свойство

```
property Help: string;
```

Если же апплет инициализируется данными из ресурса, то вместо свойств `Caption`, `AppletIcon` и `Help` надо воспользоваться свойствами:

```
property ResidName: Integer; //идентификатор (номер) заголовка апплета
property ResidIcon: Integer; //идентификатор значка в ресурсе
property ResidInfo: Integer; //идентификатор информационной строки
```

Связанные с апплетом дополнительные данные ассоциируются со свойством

```
property Data: Longint;
```

Эти данные отправляются к управляемому приложению в момент вызова событий `OnInquire()` или `OnNewInquire()`.

Своих методов (кроме видоизмененных конструктора и деструктора) у класса нет, зато `TAppletModule` может похвастаться пятью событиями (табл. 40.6).

Таблица 40.6. *Обработчики событий апплета* `TAppletModule`

Событие	Описание
property OnActivate: TActivateEvent; **type** TActivateEvent = **procedure** (Sender: TObject; Data: LongInt) **of** object;	Событие генерируется в момент двойного щелчка по значку апплета. Здесь: `Sender` — ссылка на активируемый апплет; `Data` — значение, устанавливаемое в событиях `OnInquire()` или `OnNewInquire()`
property OnStartWParms: TStartWParmsEvent; **type** TStartWParmsEvent = **procedure** (Sender: TObject; Params: **string**) **of object**;	Событие возникает совместно с `OnActivate()`, оно генерируется в момент вызова апплета из `RunDLL`. Параметр `Params` содержит данные, направляемые в адрес `RunDLL`
property OnInquire: TInquireEvent; **type** TInquireEvent = **procedure** (Sender: TObject; **var** idIcon: Integer; **var** idName: Integer; **var** idInfo: Integer; **var** lData: Integer) **of object**;	Событие вызывается в момент запроса связанной с апплетом информации о: заголовке `Caption`; ассоциированном значке `Icon`; информационной строке `Help` и данными `Data`. Перечисленные данные считываются из ресурса. Необходимые идентификаторы передаются в одноименные параметры
property OnNewInquire: TNewInquireEvent; **type** TNewInquireEvent = **procedure** (Sender: TObject; **var** lData: Integer; **var** hIcon: HICON; **var** AppletName: **string**; **var** AppletInfo: **string**) **of object**;	Назначение и вызов события аналогичны `OnInquire()`. Отличие в том, что в качестве параметров передаются не идентификаторы ресурса, а реальные значения
property OnStop: TStopEvent; **type** TStopEvent = **procedure** (Sender: TObject; Data: LongInt) **of object**;	Событие генерируется в момент получения апплетом команды на завершение работы

Приложение Панели управления *TAppletApplication*

Класс `TAppletApplication` инкапсулирует приложение Панели управления Windows. Класс описан в модуле `CtlPanel` и реализован на основе `TComponent`. Приложение Панели управления может содержать один или большее число апплетов — экземпляров класса `TAppletModule`.

> ***Замечание***
>
> Строго говоря, термин "приложение Панели управления" (Control Panel Application) не совсем корректен. На самом деле это библиотека. В этом можно убедиться, открыв головной модуль проекта. Самая первая строка модуля начинается с ключевого слова `library` (библиотека).

После компиляции файл приложения Панели управления получает характерное расширение имени cpl — этого требует директива `{$E cpl}`.

Экспортируемая функция `CPlApplet()` предоставляет апплету точку входа в Панель управления компьютером.

```
function CPlApplet(hwndCPl: THandle; uMsg: DWORD;
                   lParam1, lParam2: Longint): Longint;
```

Благодаря этой функции апплет получает возможность посылать в адрес Панели управления неполный десяток сообщений. Например, в момент двойного щелчка по значку апплета отправляется сообщение `CPL_DBLCLK`.

Связь с Панелью управления поддерживается с помощью свойства

```
property ControlPanelHandle: THandle; //только для чтения
```

Здесь хранится дескриптор Панели управления или дескриптор приложения, вызвавшего функцию `CplApplet()`.

Перечень всех имеющихся в распоряжении приложения апплетов доступен благодаря свойству

```
property Modules[Index: Integer]: TAppletModule;
```

хранящему список модулей. Сколько всего модулей скрывается в апплете, мы обнаружим в свойстве:

```
property ModuleCount: Integer;
```

Экземпляр класса `TAppletApplication` создается автоматически и будет доступен программисту посредством глобальной переменной `Application`.

Практически все методы приложения предназначены для внутреннего пользования и в обычных приложениях не применяются.

Пример апплета управления сервисом Windows

В первой части главы мы разработали сервис, позволяющий распределить файлы, поступающие в сетевую папку файлового сервера. Однако у созданного нами сервиса имеются недостатки, самый главный среди них — отсутствие пользовательского интерфейса управления параметрами сервиса. Предлагаю устранить этот недостаток, а для этого нам потребуется сделать два шага:

1. Разработать приложение управления параметрами сервиса.
2. Создать апплет панели управления.

Приложение управления сервисом

Функционал, закладываемый в приложение управления сервисом, весьма прост. Приложение должно предоставлять пользователю удобный интерфейс редактирования основных параметров сервиса управления каталогами, т. е. позволять назначать контролируемые сервисом папки и редактировать расширения имена файлов. Все внесенные пользователем изменения должны передаваться в соответствующую ветвь системного реестра, из которого их сможет считать сервис во время запуска.

Больше мы не станем углубляться в подробности работы приложения (его листинги вы найдете среди примеров) и предложим вашему вниманию только фрагмент кода, демонстрирующий порядок сохранения параметров в системном реестре (листинг 40.12).

Листинг 40.12. Сохранение параметров сервиса в системном реестре

```
const REG_KEY='\SYSTEM\CurrentControlSet\services\FolderControl\parameters';
var R:TRegistry;
begin
  R:=TRegistry.Create;
    R.RootKey:=HKEY_LOCAL_MACHINE;
    R.OpenKey(REG_KEY,True);
```

```
  if R.KeyExists('INPUT') then edINPUT.Text :=R.ReadString('INPUT')
                          else edINPUT.Text:='c:\inputfolder\';
  if R.KeyExists('DOC') then edDOC.Text :=R.ReadString('DOC')
                        else edDOC.Text:='c:\doc\';
  if R.KeyExists('PICTURE') then edPICTURE.Text:=R.ReadString('PICTURE')
                            else edPICTURE.Text:='c:\picture\';
  if R.KeyExists('OTHER') then  edOTHER.Text :=R.ReadString('OTHER')
                          else edOTHER.Text:='c:\other\';

  if R.KeyExists('PICTURE_EXT') then
                    memoPICTURE_EXT.Text:=R.ReadString('PICTURE_EXT')
  else memoPICTURE_EXT.Text:='.bmp .gif .tif .tiff .jpg .jpeg .png .emf';

  if R.KeyExists('DOC_EXT') then memoDOC_EXT.Text:=R.ReadString('DOC_EXT')
                else memoDOC_EXT.Text:='.doc .rtf .docx .docm .txt';
  R.CloseKey;
  R.Free;
end;
```

Апплет Панели управления

Воспользовавшись услугами пункта меню **File | New | Other**, создадим новый проект **Control Panel Application**. По умолчанию он сразу содержит модуль апплета `AppletModule1:TAppletModule`. Выберите этот модуль и в Инспекторе объектов внесите изменения в его свойства:

- имя модуля апплета: `Name:=fldrapplet`;
- заголовок: `Caption:='Управление службой сетевой папки'`;
- на свое усмотрение подберите подходящий по смыслу значок (свойство `AppletIcon`) и подключите его к модулю апплета.

Сохраните проект в отдельной папке под любым именем (я остановился на названии fldrnet.dproj).

Как правило, при обращении к апплету на экран компьютера выводится окно, в котором пользователь осуществляет какие-то действия по настройке своего компьютера. В нашем случае это окно целесообразно создать на основе уже готовой главной формы приложения управления сервисом. Поэтому воспользуемся пунктом меню **Project | Add to project** и подключим к апплету программный модуль главной формы приложения управления сервисом.

Выбрав пункт меню **Project | View Source**, откройте головной модуль проекта апплета и откажитесь от автоматического создания экземпляра формы `frmMain` (именно так у меня называется главная форма управления сервисом) при старте приложения (листинг 40.13).

Листинг 40.13. Настройка головного модуля проекта апплета

```
library fldrnet;

uses
  CtlPanel,
  AppletModule1 in 'AppletModule1.pas' {fldrapplet: TAppletModule},
  main in 'main.pas' {frmMain};
```

```
exports CPlApplet;
{$R *.RES}
{$E cpl}

begin
  Application.Initialize;
  Application.CreateForm(Tfldrapplet, fldrapplet);
  //Application.CreateForm(TfrmMain, frmMain); // <= возьмите строку
                                               //в комментарии
  Application.Run;
end.
```

Создание формы перепоручим модулю апплета. Для этих целей лучше всего подходит событие `OnActivate()` (листинг 40.14).

Листинг 40.14. Настройка головного модуля апплета

```
uses main;
//...
procedure Tfldrapplet.AppletModuleActivate(Sender: TObject; Data: Integer);
begin
  frmMain:=TfrmMain.Create(Application);
  frmMain.ShowModal;
end;
```

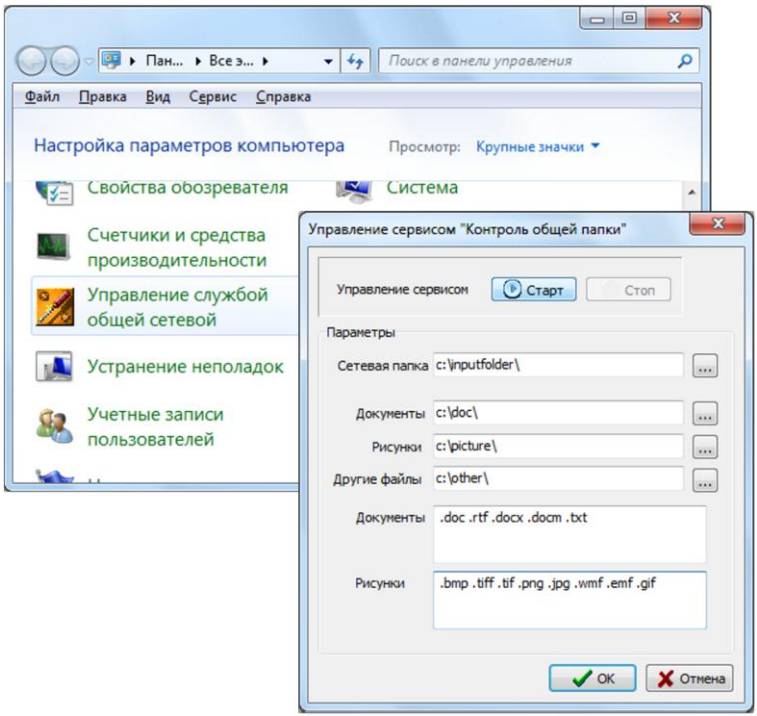


Рис. 40.3. Апплет управления сервисом

Уничтожение формы производим в момент остановки работы апплета (листинг 40.15).

Листинг 40.15. Настройка головного модуля апплета

```
procedure Tfldrapplet.AppletModuleStop(Sender: TObject; Data: Integer);
begin
  frmMain.Release;
end;
```

В результате компиляции проекта создается файл апплета fldrnet.cpl, нам лишь осталось зарегистрировать его в Панели управления компьютером. Для этого достаточно скопировать cpl-файл в системный каталог System32. После этого откройте Панель управления компьютером, и вы получите доступ к разработанному апплету (рис. 40.3).

ГЛАВА 41

Динамически подключаемые библиотеки

При освоении Windows вы, наверное, уже сталкивались с *динамически подключаемыми библиотеками* (Dynamic Link Library, DLL). Каждая из библиотек представляет собой отдельный программный модуль, содержащий код, данные или ресурсы (или все вместе) и допускающий совместное их использование несколькими приложениями. Например, библиотека kernel32.dll отвечает за управление памятью и инициализацию процессов, библиотека comdlg32.dll содержит все стандартные диалоговые окна, user32.dll поддерживает пользовательский интерфейс, gdi32.dll обслуживает графику.

Для того чтобы понять принцип работы динамических библиотек, надо знать ответ на вопрос: что происходит внутри операционной системы при запуске приложения (а если говорить еще точнее — процесса)? В 32-разрядных версиях Windows стартующий процесс в свое распоряжение получит отдельное адресное пространство размером 4 Гбайт виртуальной памяти, в 64-разрядных речь ведется об астрономическом размере в 8 Тбайт. Каждый из процессов наивно полагает, что все адресное пространство принадлежит только ему. Однако более просвещенному программисту надо понимать, что каждому выполняющемуся в Windows процессу невозможно предоставить по 2^{32}, а тем более и 2^{64} байт быстрой оперативной памяти. Поэтому для процесса распределяется участок памяти не только из оперативной памяти, но и из системного страничного файла (файла подкачки).

После успешного отображения исполняемого файла в виртуальную память операционная система разыскивает внутри этого файла все ссылки на необходимые динамически подключаемые библиотеки, и эти библиотеки отображаются в тот же процесс — где-то внутри участка адресуемой памяти процесса. Владельцем всех данных, объектов и переменных динамической библиотеки становится поток управления, вызвавший функцию DLL. Для каждого файла библиотеки создается собственный объект отображения файла. Если, в свою очередь, внутри динамической библиотеки существуют ссылки на другие файлы библиотек, то последние также отображаются в адресное пространство.

Процесс способен загружать одну и ту же динамическую библиотеку многократно, запуская дополнительные потоки. Но при этом физическое отображение DLL в адресное пространство происходит только при первой загрузке. Последующие загрузки осуществляют приращение счетчика обращений к библиотеке. Если DLL загружалась *N* раз, то образ библиотеки в адресном пространстве процесса будет храниться до тех пор, пока не произойдет *N* выгрузок. С каждой выгрузкой динамической библиотеки счетчик обращений уменьшает свое значение на 1. Нулевое значение счетчика является признаком того, что библиотека более не нужна.

При завершении работы образ каждого DLL-файла удаляется из памяти вместе с соответствующим объектом отображения. Процесс завершится до удаления образа исполняемого

файла и объекта главного процесса. С завершением главного процесса автоматически освобождается используемая память.

Создание проекта DLL

Создание проекта динамически подключаемой библиотеки начинается с обращения к традиционному пункту меню **File | New | Other**, затем на странице **New** диалогового окна **New Items** следует выбрать пункт **Dynamic-link Library**.

В ответ на наши действия Delphi создаст шаблон кода, начинающийся с ключевого слова `library` — это и есть визитная карточка проекта библиотеки DLL. Сохраните код в отдельном каталоге под именем mathlib.dproj. Как вы наверняка догадались по названию, наша библиотека будет связана с математическими расчетами. Прообраз нашей первой "тренировочной" библиотеки представлен в листинге 41.1.

Листинг 41.1. Шаблон кода библиотеки

```
library mathlib;

{ 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  System.SysUtils,  System.Classes;

{$R *.res}
begin
end.
```

Как видите, размер комментария в листинге существенно превышает объем полезного кода. На такие жертвы разработчики Delphi пошли для того, чтобы предупредить программиста о том, что если хотя бы одна из экспортируемых функций DLL требует данные строкового типа (в качестве параметра или в виде возвращаемого значения), то самым первым в операторе `uses` обязан стоять модуль `ShareMem`. Такая необходимость обусловлена особенностью хранения в памяти фирменного для Delphi строкового типа данных `String`. В свою очередь, для обслуживания программного модуля `ShareMem` требуется еще один помощник — динамическая библиотека BORLNDMM.DLL (ее вы обнаружите в каталоге BIN), которую следует распространять вместе с проектом.

> ***ЗАМЕЧАНИЕ***
>
> При желании можно отказаться от услуг библиотеки BORLNDMM.DLL, но в этом случае текстовые данные следует передавать при помощи указателя `PChar` или `ShortString`.

Объявление и экспорт функций в DLL

Для того чтобы стороннее приложение смогло воспользоваться описанными в библиотеке функциями, эти функции должны быть помечены как экспортируемые. Синтаксис объявления экспортируемых библиотекой методов имеет ряд небольших отличий от уже привычного описания. Четко выделяются следующие особенности:

- между экспортирующей функции библиотекой и выступающим в роли импортера приложением должно быть заключено *соглашение о вызовах*, оно необходимо для обеспечения корректной передачи параметров и возврата результата выполнения функции;
- объявление экспортируемой функции с ключевым словом `export` и использование секции `exports` для указания имен экспортируемых процедур и функций.

Соглашение о вызовах

Прежде чем написать первую экспортируемую функцию, программист принимает решение о способе передачи данных между модулями проекта. В Delphi различают пять соглашений о вызовах (протоколов), отличающихся друг от друга особенностями использования стека (или регистров) для передачи параметров, направлением заполнения стека (как есть, слева направо или справа налево), порядком очистки стека. Такое разнообразие протоколов объясняется достаточно просто — стремлением разработчиков Delphi к построению максимально гибкого программного продукта, совместимого с другими языками программирования (табл. 41.1).

Таблица 41.1. *Соглашения о вызовах*

Директива	Используется	Порядок	Очистка стека
`register`	Регистры, стек	Слева направо	Вызываемая процедура
`pascal`	Стек	Слева направо	Вызываемая процедура
`cdecl`	Стек	Справа налево	Вызывающая процедура
`stdcall`	Стек	Справа налево	Вызываемая процедура
`safecall`	Стек	Справа налево	Вызываемая процедура

Директива `register` считается наиболее эффективным протоколом передачи данных и устанавливается по умолчанию. Здесь по возможности для первых трех передаваемых параметров используются расширенные регистры процессора (EAX, EDX и ECX). Остальные параметры (если они есть) передаются как в директиве `pascal`. К сожалению, из-за уникальности протокола он может использоваться только с компиляторами Delphi или C++Builder.

Соглашение о вызовах `pascal` по скорости слегка уступает `register`. К недостаткам протокола можно отнести отсутствие поддержки переменного числа параметров.

Директива `cdecl` предназначена для организации вызовов в духе языка Си. Протокол допускает передачу переменного числа параметров, но в Delphi это не всегда имеет смысл.

Директивы `stdcall` и `safecall` используются для обращения к API Windows. Протокол `safecall` предназначен для объявления методов с дуальным интерфейсом. Например, это соглашение о вызовах используется для обращения к объектам автоматизации.

ВНИМАНИЕ!

Для того чтобы библиотечный модуль был совместим с применяемыми по умолчанию функциями Windows API, применяйте соглашение `stdcall`.

Пример экспорта функций

Продолжим работу с проектом библиотеки mathlib.dproj. Чтобы не перегружать заголовочный модуль библиотеки избыточным кодом, добавьте к проекту DLL новый модуль. Для этого воспользуйтесь меню **File | New | Unit — Delphi**. Сохраните созданный модуль под именем mathematic.pas.

Раз наша динамическая библиотека предназначена для решения математических задач, то первой экспортируемой функцией станет функция, суммирующая пару чисел, назовем ее `addition()`. Объявите заголовок и тело функции так, как это представлено в листинге 41.2.

Листинг 41.2. Модуль `mathematic` с функцией `addition()`

```
unit mathematic;

interface

function addition(X,Y : Real):Real; stdcall; export;

implementation
function addition(X,Y : Real):Real;
begin
  Result:=X+Y;
end;
end.
```

Теперь нам следует опубликовать функцию `addition()` в разделе экспорта головного модуля библиотеки mathlib.dproj. Для этого выберите пункт меню **Project | View Source** и незначительно доработайте основной модуль нашей библиотеки (листинг 41.3).

Листинг 41.3. Головной модуль библиотеки `mathlib` с секцией экспорта

```
library mathlib;
uses
  SysUtils, Classes,
  mathematic in 'mathematic.pas';

{$R *.res}

exports addition; //секция экспорта с именем экспортируемой функции

begin
end.
```

Пример хранения форм в библиотеке

При проектировании профессионального программного пакета, содержащего несколько приложений, совместно используемые приложениями формы программисты стараются вынести в динамические библиотеки. Такое решение позволит отказаться от бессмысленного размножения одинаковых форм в каждом проекте, снизит размер программного продукта на жестком диске и, самое главное, значительно сократит временные ресурсы, затрачиваемые на создание приложения.

Доступ приложения к хранящимся во внешнем модуле формам существенно отличается от обращения к форме из проекта. Здесь команда `Form1.Show`, к сожалению, не сработает. Для взаимодействия с хранящейся в DLL формой придется создавать специальные функции-оболочки.

Достойным развитием математической библиотеки будет создание формы-калькулятора, позволяющей нам осуществлять любые расчеты. Добавим к библиотеке модуль формы, для этого обратитесь к меню **File | New | Form — Delphi**. Переименуйте форму в `frmCalculator`, а соответствующий ей программный модуль сохраните под именем calculator.pas.

По объективным причинам мы не станем обсуждать непосредственно код калькулятора (вы его найдете среди примеров к книге), а упомянем лишь самое важное с точки зрения реализации нашей библиотеки. На форме калькулятора (рис. 41.1) помимо других элементов управления расположены:

- метка `lbRes:TLabel`, в которой отображаются результаты расчетов;
- кнопка `btnOK:TBitBtn` со свойством `Kind`, установленным в состояние `bkOK`, при нажатии кнопки форма закрывается, возвращая модальный результат `idOK`;
- кнопка `btnCancel:TBitBtn` со свойством `Kind`, установленным в состояние `bkCancel`, при нажатии кнопки форма закрывается, возвращая модальный результат `idCancel`.

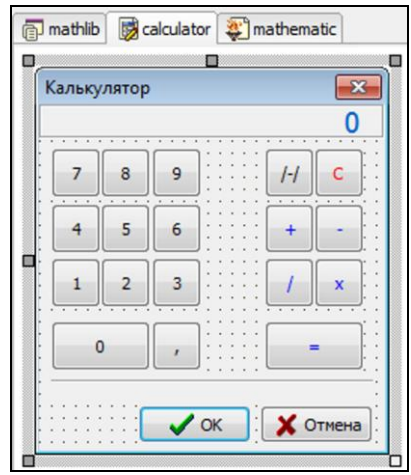


Рис. 41.1. Форма калькулятора

Опишем экспортируемую библиотекой функцию, благодаря которой к форме калькулятора будет возможно обратиться из других приложений (листинг 41.4).

Листинг 41.4. Функция, экспортируемая модулем калькулятора

```
unit calculator;
//...

function showcalculator(
             hWnd:THandle; InitValue:real=0):Real; stdcall; export;

var frmCalculator: TfrmCalculator;
```

```
implementation
{$R *.dfm}

function showcalculator(hWnd:THandle; InitValue:real=0):Real;
begin
  Application.Handle:=hWnd;
  frmCalculator:=TfrmCalculator.Create(Application);
  frmCalculator.lbRes.Caption:=FloatToStr(InitValue);
  Result:=0;
  if frmCalculator.ShowModal=mrOK then
              Result:=StrToFloat(frmCalculator.lbRes.Caption);
  frmCalculator.Release;
  Application.Handle:=0;
end;
```

Объявленная в листинге функция `showcalculator()` обладает двумя параметрами. Параметр `Wnd` необходим для передачи в создаваемую форму ссылки на дескриптор приложения, вызвавшего форму. Необязательный параметр `InitValue` содержит число, которым мы сможем проинициализировать начальное значение калькулятора. Форма калькулятора создается и выводится на экран в модальном режиме. Завершив расчеты, пользователь нажимает кнопку **OK** (`btnOK:TBitBtn`), и результаты расчетов из метки `lbRes` передаются в нашу функцию.

Для того чтобы завершить работу над библиотекой, вновь перейдите в головной модуль mathlib.dproj и добавьте в секцию экспорта название функции `showcalculator()`:

```
exports addition,showcalculator;
```

Откомпилируйте библиотеку, нажав комбинацию клавиш <Ctrl>+<F9>. В результате должен создаться файл mathlib.dll.

> ***Внимание!***
>
> Если вы работаете в коллективе разработчиков ПО, то учитывайте, что некоторые языки программирования чувствительны к регистру символов (например, Си). Поэтому, определяя имена экспортируемых функций, всегда используйте нижний (или верхний) регистр символов.

Вызов библиотеки из приложения

Чтобы динамической библиотекой смогло воспользоваться приложение, нам следует выбрать для нее подходящий каталог. Для этого подходят:

- каталог, из которого запускается основной исполняемый exe-файл;
- текущий каталог;
- системная папка ОС Windows (c:\windows\system32 или c:\windows\syswow64);
- папка Windows;
- каталоги, включенные в переменную окружения `PATH`.

Именно в этой очередности осуществляется просмотр каталогов при поиске требуемой библиотеки. Обычно наиболее востребованные библиотеки размещаются в системной папке,

там их могут найти все установленные в Windows приложения. А в нашем случае во время разработки проекта наиболее целесообразно размещать свои библиотеки в том же каталоге, где находится исполняемый файл.

Неявное подключение DLL

Для того чтобы воспользоваться услугами динамической библиотеки, приложение должно отобразить DLL в свое адресное пространство. Различают два способа решения этой задачи: *неявное* (статическое) и *явное* (динамическое) подключение библиотеки.

Неявная загрузка предназначена для тех случаев, когда услуги библиотеки необходимы на протяжении всего периода выполнения приложения. В этом случае библиотека подключается к процессу в момент его старта и отключается вместе с завершением работы. Организация неявной загрузки включает следующие этапы:

- регистрация в секции `interface` прототипов импортируемых функций и процедур с указанием соглашения о вызовах;
- объявление в разделе констант секции `implementation` имени подключаемой библиотеки и импортируемых из нее функций и процедур.

Создайте новое приложение и сохраните проект под любым именем. Разместите на его главной форме две кнопки — они возьмут на себя ответственность за вызов функций `addition()` и `showcalculator()`. Кроме того, нам понадобится несколько строк ввода `TEdit`, они позволят пользователю вводить исходные значения. Весь процесс неявного подключения разработанной нами математической библиотеки mathlib.dll нашел отражение в листинге 41.5.

Листинг 41.5. Неявное подключение библиотеки к приложению

```
unit Main;

interface

//...
//объявление прототипов импортируемых функций
function addition(X,Y:Real):Real; stdcall;
function showcalculator(Wnd:THandle; InitValue:Real=0):Real; stdcall;

var frmMain: TfrmMain;
implementation

{$R *.dfm}
const libname='mathlib'; //имя библиотеки

function addition; external libname name 'addition';
function showcalculator; external libname name 'showcalculator';

procedure TfrmMain.Button1Click(Sender: TObject);
var X,Y,Z:Real;
```

```
begin
  //сумма двух значений
  X:=StrToFloatDef(Edit1.Text,0);
  Y:=StrToFloatDef(Edit2.Text,0);
  Z:=Addition(X,Y);
  Edit3.Text:=FloatToStr(Z);
end;

procedure TfrmMain.Button2Click(Sender: TObject);
var InitValue,Z:Real;
begin
  //вызов калькулятора
  InitValue:=StrToFloatDef(Edit4.Text,0);
  Z:=showcalculator(Application.Handle,InitValue);
  Edit4.Text:=FloatToStr(Z);
end;
end.
```

Внимание!

Название библиотеки и имена экспортируемых функций чувствительны к регистру символов!

Перед первым запуском приложения следует убедиться, что исполняемый exe-модуль и файл библиотеки mathlib.dll находятся в одном каталоге. Чтобы не утруждать себя лишними операциями копирования файлов, проще всего внести незначительное изменение в поведение компилятора Delphi. Воспользуйтесь пунктом меню **Project | Options**, в появившемся окне опций проекта выделите узел верхнего уровня **Delphi Compiler** и в строке **Output directory** укажите папку, в которой размещена наша библиотека (рис 41.2). Нажав кнопку **OK**, сохраните настройки компилятора и только после этого запускайте проект на выполнение.

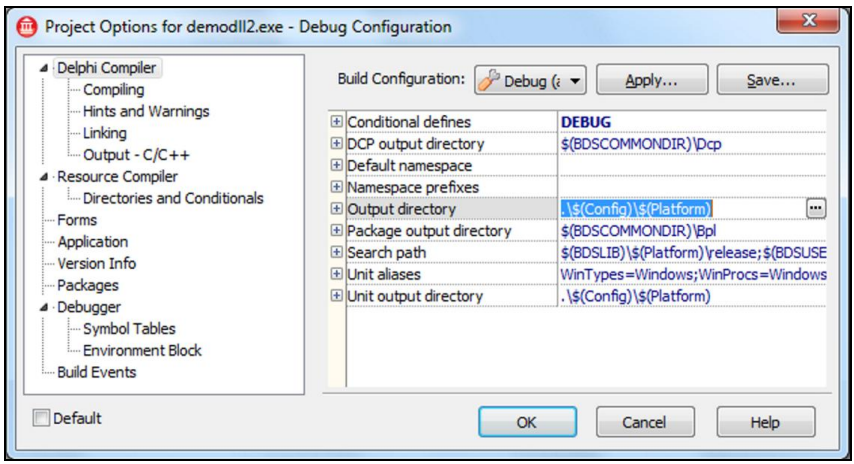


Рис. 41.2. Установка каталога для размещения откомпилированного файла

Явное подключение DLL

Явное (динамическое) подключение библиотеки к проекту целесообразно применять в тех случаях, когда обращение к библиотеке происходит нерегулярно. В такой ситуации лучше всего отображать образ библиотеки в адресное пространство процесса только на время обращения к той или иной функции и сразу выгружать библиотеку.

При организации явной загрузки программист должен сделать четыре шага:

1. Загрузить DLL.
2. Получить адрес импортируемой функции.
3. Вызвать импортируемую функцию.
4. Выгрузить библиотеку из памяти.

Три из четырех действий реализуются при помощи функций Windows API. Отображение динамической библиотеки в адресное пространство процесса осуществляет функция

```
function LoadLibrary(LibFileName: string):THandle;
```

Параметр `LibFileName` содержит имя (обычно без пути) подключаемой библиотеки.

В случае успеха функция возвратит дескриптор загрузившейся библиотеки.

Вторым этапом при использовании механизма явной загрузки библиотеки является этап получения адреса функции импортируемой из DLL. Для этого используется функция

```
function GetProcAddress(hModule: THandle; ProcName: string): Pointer;
```

Параметр `hModule` соответствует дескриптору библиотеки, полученному при помощи функций `LoadLibrary()` или `LoadLibraryEx()`. Имя функции описывается в параметре `ProcName`. Возвращаемое значение не что иное, как указатель на вызываемую функцию.

На завершающем этапе необходимо выгрузить DLL из памяти системы. Для этого используют функцию

```
function FreeLibrary(hModule: THandle ): LongBool;
```

Параметр `hModule` нам уже знаком, это дескриптор ранее загруженной библиотеки. Функция `FreeLibrary()` уменьшает на 1 значение счетчика количества пользователей загруженного модуля в адресном пространстве процесса. Если счетчик примет нулевое значение, то процесс выгружает библиотеку из памяти.

Практическая сторона вопроса явной загрузки DLL и обращения к функции суммирования отражена в листинге 41.6.

Листинг 41.6. Явное подключение библиотеки к приложению

```
procedure TfrmMain.Button1Click(Sender: TObject);
const libname='mathlib'; //имя библиотеки
type TAddition = function (X,Y:Real):Real; stdcall;
var dwError:DWord;
    Addition:TAddition;
    hLib:THandle;
    X,Y,Z:real;
begin
  hLib:=LoadLibrary(libname); //загрузка библиотеки
```

```
  if hLib=0 then
  begin
    dwError:=GetLastError();
    raise Exception.CreateFmt('Ошибка библиотеки %s, код %d',
                              [libname,dwError]);
    Exit;
  end;
  try
    Addition:=GetProcAddress(hLib,'addition'); //получаем адрес функции
    if @Addition<>nil then
    begin
      X:=StrToFloatDef(Edit1.Text,0);
      Y:=StrToFloatDef(Edit2.Text,0);
      Z:=Addition(X,Y);
      Edit3.Text:=FloatToStr(Z);
    end;
  finally
    FreeLibrary(hLib); //завершение работы с библиотекой
  end;
end;
```

ЗАМЕЧАНИЕ

На этапе подключения динамической библиотеки к проекту весьма вероятно возникновение ошибки `ERROR_DLL_NOT_FOUND = 1157`. Эта ошибка свидетельствует об отсутствии файла загружаемой библиотеки.

ГЛАВА 42

Многокомпонентная модель COM

Разработанная в корпорации Microsoft *многокомпонентная модель объектов* (Component Object Model, COM) представляет собой спецификацию, описывающую способ взаимодействия программ. Программа-сервер (сервер COM) предоставляет в наше распоряжение определенные услуги, программа-клиент (клиент COM) пользуется этими услугами. Взаимодействующие COM-объекты могут функционировать как в рамках одного и того же процесса, так и в разных процессах, в том числе протекающих на разных машинах.

Компоненты COM состоят из программного кода, реализованного в виде исполняемого exe-файла или динамической библиотеки DLL. С точки зрения стороннего наблюдателя создаваемый средствами COM программный компонент представляет собой черный ящик, обладающий следующими характеристиками:

- легкостью подключения к операционной системе Windows;
- автономностью;
- простотой интеграции в состав программного обеспечения;
- универсальным способам обращения к методам и свойствам из любого современного языка программирования. Благодаря строгой спецификации интерфейсов достигается единообразие работы клиента с любым COM-компонентом;
- прозрачностью, сервер COM, расположенный на другой сетевой станции, может рассматриваться клиентом так же, как будто он находится на вашем компьютере.

В интересах клиентских и серверных приложений COM в Windows активно трудится *библиотека COM* (COM Library), она способна:

- предоставлять в распоряжение программиста некоторый набор функций, облегчающий разработку приложений COM;
- генерировать уникальный идентификатор объекта, осуществлять поиск требуемого объекта по его идентификатору;
- производить вызов удаленных процедур тогда, когда сервер COM размещен на другом компьютере;
- управлять распределением памяти в интересах взаимодействующих процессов и контролировать освобождение памяти.

COM-объект

В основу многокомпонентной модели положено понятие COM-объекта. Физически COM-объект, как и любой другой объект, соответствующий парадигме ООП, представляет собой совокупность данных и управляющих этими данными методов. Вместе с тем структура COM-объекта существенно отличается от построения привычных для нас объектов VCL. Ключевое отличие объекта COM от объекта Delphi в способе предоставления к полям и методам объекта.

Прямого доступа к данным COM-объекта извне в принципе не существует. Каждое поле объекта скрыто в его недрах и может обслуживаться только методами COM-объекта. Методы объекта также особой коммуникабельностью не отличаются, они упакованы так глубоко, что доступны лишь через свои указатели. В свою очередь указатели на методы хранятся в специальных таблицах адресов функций — в *виртуальных таблицах* объекта (virtual tables). Описание каждой из таких таблиц называют *интерфейсом* (interface). Сколько таблиц — столько у объекта интерфейсов. Но это еще не все, непосредственное обращение к интерфейсу (указателю на виртуальную таблицу адресов методов) невозможно — вместо этого нам лишь разрешается работать с указателем на интерфейс, это единственное что доступно извне COM-объекта.

Число интерфейсов COM-объекта определяется замыслом разработчика. В границах отдельного интерфейса программисты обычно объединяют однородные по функциональному назначению методы, таким образом, чем больше задач решает объект, тем значительнее перечень его интерфейсов. Вне зависимости от того, сколько интерфейсов объявит программист, любой COM-объект поддерживает один обязательный базовый интерфейс `IUnknown`. Базовый интерфейс предназначен для информирования клиента COM об имеющихся в распоряжении объекта интерфейсах. Получив доступ к обязательно присутствующему интерфейсу `IUnknown`, программа или любой другой объект сразу может обратиться к методу `QueryInterface()` и узнать обо всех остальных имеющихся у этого объекта интерфейсах.

В рассматриваемой в этой главе технологии COM доступ клиентского приложения к методам интерфейсов COM-объекта осуществляется с помощью механизма *раннего связывания*. Суть раннего связывания заключается в том, что будущему клиентскому приложению еще на этапе компиляции предоставляется полная информация о методах и свойствах интересующих его интерфейсов. В противном случае клиент просто не поймет, каким образом ему следует обращаться к серверу. Информация обо всех тайнах интерфейса описывается в виде специального файла — библиотеки типов.

Понятие интерфейса

По своей сути интерфейс — это договор о намерениях между COM-объектом и клиентским приложением, в котором COM-объект гарантирует клиенту предоставление некоторых услуг. Услуги — это определенные в интерфейсе методы.

Весьма значимым аспектом интерфейса является то, что если COM-объект заявляет о существовании у него определенного интерфейса, то объявленный интерфейс в обязательном порядке должен быть поддержан внутренними методами объекта и возвращать какие-либо значения.

Интерфейсы могут наследоваться, но при условии, что дочерний интерфейс должен включать все методы своего предка и добавлять к ним свои. Все существующие интерфейсы наследуются от базового интерфейса `IUnknown`.

ВНИМАНИЕ!

Когда речь ведется о наследовании в COM, надо понимать, что COM поддерживает наследование интерфейса, т. е. наследование классом интерфейса базового класса. Но COM сознательно не поддерживает наследование реализации, т. е. наследование кода от базового класса.

Выделим еще один ряд правил проектирования COM-объекта:

- клиенты не могут получить услуги от COM-объекта никоим образом, кроме как с помощью интерфейса объекта;
- единожды опубликованный интерфейс никогда не может быть изменен. Необходимость дополнить существующий интерфейс новыми функциональными возможностями приводит к созданию нового интерфейса. Выполнение этого правила гарантирует, что запрашивающее конкретный интерфейс клиентское приложение получит именно то, на что оно рассчитывает;
- в рамках отдельной системы каждый отдельный интерфейс должен иметь уникальный идентификатор и название, начинающееся с символа "I".

На первый взгляд форма определения интерфейса весьма схожа с объявлением обычного класса VCL, но это совсем не так (листинг 42.1). Первое бросающееся в глаза отличие в том, что в момент декларации интерфейс должен идентифицироваться 128-битным *глобальным уникальным идентификатором* (Globally Unique Identifier, GUID). Когда GUID применяют для именования COM-интерфейсов, то идентификаторы интерфейсов обычно именуют IID (Interface ID), если GUID именует класс — говорят об идентификаторе класса CLSID (Class ID).

Листинг 42.1. Обобщенная синтаксическая конструкция объявления интерфейса

```
type имя_интерфейса = interface (интерфейс_предок)
          ['{GUID}'] //глобальный уникальный идентификатор класса
        {объявление заголовков методов и их параметров}
         function MyFunction(const IID:GUID):HResult; safecall;
end;
```

Необходимость GUID объясняется элементарным здравым смыслом — весьма высока вероятность, что текстовые названия интерфейсов различных разработчиков стали бы совпадать. В результате приложение-клиент, обращаясь к услугам какого-то объекта COM, могло бы получить абсолютно неожиданный ответ от совсем другого объекта-"однофамильца". Генерация GUID осуществляется в соответствии с алгоритмом, предложенным консорциумом Open System Foundation. Алгоритм практически исключает вероятность того, что единожды сгенерированный GUID будет когда-нибудь повторен.

ЗАМЕЧАНИЕ

Для автоматической генерации нового GUID, находясь в редакторе кода Delphi, нажмите комбинацию клавиш <Ctrl>+<Shift>+<G>.

Но на этом особенности определения интерфейса не ограничиваются. Хотя и интерфейс имеет право объявлять заголовки методов, но ни один из его методов не имеет описания — он лишь декларирует названия методов и свойств. Для того чтобы объявленные в интерфейсе методы стали работоспособными, приходится создавать специальные классы, реализующие интерфейс.

Почему интерфейсам не доверили, казалось, самое главное — реализацию? Ответ очевиден. Потому, что реализация интерфейса может изменяться вместе с развитием компонента, а интерфейс оставаться неизменным. Еще одна причина в том, что интерфейсы всегда описываются на стандартном языке определения интерфейсов IDL (благодаря этому интерфейсы COM могут быть доступны практически из любого языка программирования), а реализация интерфейса может создаваться любыми доступными для разработчика способами.

Базовый интерфейс *IUnknown*

Отправной точкой для разработки всех интерфейсов выступает интерфейс `IUnknown`. Он описан в модуле `System` и поэтому доступен практически во всех проектах VCL. Базовый интерфейс отвечает за выполнение двух задач:

- поддержку механизма подсчета ссылок на интерфейс;
- проверку наличия у объекта определенного интерфейса.

Роль интерфейса `IUnknown` в иерархии интерфейсов в чем-то сродни роли абстрактного класса `TObject` в иерархии классов VCL — и тот, и другой являются родоначальниками для всех имеющихся в природе интерфейсов (в случае `IUnknown`) или классов (если речь ведется о `TObject`). Интерфейс обладает тремя базовыми методами (листинг 42.2).

Листинг 42.2. Определение базового интерфейса `IUnknown`

```
type
    IUnknown = IInterface;
    IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function AddRef: Integer; stdcall;
    function Release: Integer; stdcall;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  end;
```

На порядок декларации входящих в коллекцию методов также наложен ряд ограничений, например, мы не вправе пользоваться классическими для Delphi директивами `virtual`, `dynamic`, `abstract` или `override`. После объявления метода программист должен указать используемое соглашение о вызовах. Так стандартное соглашение `stdcall` позволит обращаться к интерфейсам Windows, для работы с интерфейсами CORBA потребуется соглашение `safecall`. Если никакое соглашение не указано, то Delphi станет использовать соглашение `register`.

> ***Замечание***
>
> Строго говоря, в VCL базовым интерфейсом выступает не `IUnknown`, а `IInterface`. Однако с точки зрения COM именно интерфейс `IUnknown` считается опорным по отношению ко всем интерфейсам.

Каждый интерфейс снабжен внутренним счетчиком ссылок. Благодаря *механизму подсчета ссылок* (reference counting) один и тот же COM-объект умеет обслуживать несколько клиентов и, кроме того, управлять своим собственным существованием. В момент создания интерфейсного объекта его внутреннему счетчику присваивается значение 1. При каждом последующем подключении нового клиента к интерфейсу, в момент получения им указателя на интерфейс, счетчик получает приращение, а при отключении клиента — счетчик умень-

шается на единицу. В момент достижения счетчика нулевого значения объект самостоятельно выгружает себя из памяти. Управление счетчиком ведают методы `AddRef()` и `Release()`. Первый из них увеличивает значение счетчика, а второй — уменьшает.

Существенное значение отводится методу `QueryInterface()`. Благодаря ему мы запрашиваем объект на предмет наличия у него интерфейса, идентифицируемого параметром `IID`. Переменная `Obj` передается для получения ссылки на интерфейс. В случае положительного ответа функция самостоятельно обратится к методу `_AddRef()` вызываемого интерфейса и возвратит указатель на интерфейс, иначе выходной параметр `Obj` будет ссылаться "в никуда" — `nil`. У `QueryInterface()` есть еще одна примечательная черта. Когда мы с помощью интерфейса `IUnknown` обращаемся к составному объекту COM (объекту, построенному на базе многокомпонентного класса COM), то метод проверит все идентификаторы реализованных в составном объекте интерфейсов и возвратит нужный.

Реализация интерфейса

Интерфейс — это не более чем договор о намерениях, предоставляемых COM-объектом на всеобщее обозрение. Клиентские приложения, вызвав интерфейс, очень рассчитывают получить запрашиваемую услугу. Но ведь интерфейс — это просто обещание оказать услугу, а каким образом обещания воплощаются в жизнь?

Допустим, что мы планируем создать лучший на свете интерфейс `IMySuperInterface`, тогда нам не остается ничего, кроме того, как объявить его (листинг 42.3).

Листинг 42.3. Пример объявления интерфейса

```
type IMySuperInterface = interface(IUnknown)
    ['{06D87C21-96A5-405F-AFC2-BB3F4EAE19D6}']
    function VeryGoodFunction: HResult;
     //другие функции интерфейса
  end;
```

Для того чтобы интерфейс начал приносить пользу, он должен быть реализован — другими словами, заполнен программным кодом. Только выполнив код, COM-объект поможет клиенту решить его задачу. В языке Delphi интерфейс реализуется только в классе. Для этого при объявлении класса, ответственного за реализацию интерфейса, мы, кроме названия класса-предка, перечисляем поддерживаемые классом интерфейсы (листинг 42.4).

Листинг 42.4. Включение интерфейса в объявление класса

```
type TMyClass = class(TComponent, IMySuperInterface)
    function VeryGoodFunction: HResult;
    //другие функции
    destructor Destroy; override; //деструктор объекта
  end;
...
function TMyClass.VeryGoodFunction: HResult;
begin
  //код реализации
end;
```

```
destructor TMyClass.Destroy;
begin
  inherited;
{!!! объект следует защищать от разрушения методами Destroy, Free и т. п.}
end;
```

После объявления имени класса следует перечень методов, имена и параметры которых должны целиком и полностью соответствовать методам интерфейса.

При проектировании классов, реализующих интерфейс, необходимо защитить экземпляр класса от несанкционированного уничтожения. Дело в том, что механизм COM сам позаботится о разрушении COM-объекта и освобождении памяти в тот момент времени, когда объект перестанет быть востребованным. Это событие произойдет в ту же секунду, когда счетчик ссылок на предоставляемый объектом интерфейс получит нулевое значение. Именно поэтому нам пришлось внести некоторые изменения в деструктор объекта.

Объекты, реализующие интерфейс, не следует уничтожать с помощью вызова их деструкторов, по крайней мере до тех пор, пока на их интерфейс существует хотя бы одна ссылка. Если вы полагаете, что объект больше не нужен, то вместо вызова метода `Destroy()` следует просто присвоить неопределенный указатель `nil` переменной, ссылающейся на интерфейс (листинг 42.5). Такое действие автоматически приведет к вызову метода `_Release` и уменьшению счетчика ссылок на единицу.

Листинг 42.5. Отказ от использования интерфейса

```
var MyClass:TMyClass;
    I:IMySuperInterface;
begin
  MyClass:=TMyClass.Create(nil);
  I:=MyClass;
 //работа с интерфейсом
  I:=nil;
end;
```

Порядок вызова сервера COM

Прежде чем модель COM начнет функционировать, клиентское приложение должно отправить запрос на услуги COM-сервера. Каким образом клиентское приложение получит доступ к интересующему его COM-серверу? В особенности, если клиент модели COM работает в другом адресном пространстве на удаленном компьютере?

Механизм, позволяющий клиенту получать доступ к объектам, расположенным в другом адресном пространстве или на другом компьютере, называется *маршалингом (marshaling)*. Маршалинг отвечает за выполнение двух задач:

- получение указателя на интерфейс из процесса сервера и организацию доступа к нему из процесса клиента;
- передачу аргументов методов интерфейса в процесс удаленного объекта.

Еще она особенность взаимодействия приложений в COM связана с системным реестром Windows. Во время установки COM-приложения в реестр операционной системы вносятся сведения об имеющихся в наличии объектах COM. В первую очередь это однозначно опре-

деляющий класс объекта идентификатор CLSID. Идентификаторы классов хранятся в подразделах:

```
HKEY_CLASSES_ROOT\CLSID\{идентификатор COM}
HKEY_LOCAL_MACHINE\SOFTWARE\Имя_класса\CLSID\
```

Кроме того, в реестре хранится имя модуля, содержащего сервер, или, в случае, когда сервер выполняется на другой машине, — сетевой адрес.

Выяснив местонахождение сервера, клиент обращается к нему с целью получить ссылки на нужные интерфейсы. Если сервер еще не был запущен, то он стартует, и в нем создается экземпляр объекта, отвечающего за обслуживание обязательного для любого сервера COM интерфейса `IUnknown` и соответствующего ему метода `QueryInterface()`. С этого момента `IUnknown` способен рассказать клиенту обо всех возможностях своего хозяина.

Как только клиент начнет запрашивать у `IUnknown` другие интерфейсы и вызывать предоставляемые ими методы, сервер должен немедленно создать реализующие их экземпляры объектов. Для этих целей в распоряжении сервера имеется отдельный специалист — интерфейс *фабрики классов* `IClassFactory`. Для создания нового экземпляра класса фабрика пользуется шаблоном класса, или, как говорят, *фабричным образцом* (factory pattern). Фабричный образец — это особый класс, применяемый для создания экземпляра другого класса.

Интерфейс *IClassFactory* и библиотека COM

Фабрика класса — это разновидность COM-объекта, поддерживающего интерфейс `IClassFactory`. Описание ключевого интерфейса `IClassFactory` мы найдем в недрах модуля `ActiveX`, в нем определены всего два метода (листинг 42.6).

Листинг 42.6. Объявление интерфейса `IClassFactory`

```
IClassFactory = interface(IUnknown)
    ['{00000001-0000-0000-C000-000000000046}']
    function CreateInstance(const unkOuter: IUnknown;
                            const iid: TIID;
                            out obj): HResult; stdcall;
    function LockServer(fLock: BOOL): HResult; stdcall; ]
end;
```

Первый из указанных методов `CreateInstance()` используется для создания экземпляра COM-объекта. Параметр `Outer` предназначен для создания агрегатов и в обычных условиях не задействуется. Второй параметр `IID` рассчитывает получить идентификатор GUID запрашиваемого интерфейса. После успешного вызова метода в выходной параметр `obj` будет помещен указатель на интересующий нас интерфейс COM-объекта.

> ***ЗАМЕЧАНИЕ***
>
> Под агрегатом понимается совокупность объектов COM, предоставляющих клиенту некоторый набор собственных интерфейсов. Всеми объектами управляет контроллер, он владеет управляющим интерфейсом `IUnknown`.

Еще одна особенность метода `CreateInstance()` в том, что она поддерживает два алгоритма создания экземпляра объекта. Если запрашиваемый клиентом COM-объект способен одновременно обслуживать несколько клиентов, то очередная копия объекта не создается, а сис-

тема лишь увеличивает число ссылок на уже созданный ранее объект. Второй вариант поведения функции `CreateInstance()` основан на идее, что для каждого нового клиента должен создаваться новый экземпляр класса.

Второй метод фабрики `LockServer()` блокирует COM-объект, не позволяя системе выгрузить его из памяти, даже когда его не использует ни одно из клиентских приложений. Для включения блокировки в параметр `Lock` должно быть передано значение `true`. Таким образом, мы принудительно увеличиваем на единицу счетчик блокировок и, соответственно, не допускаем уничтожение объекта. Такое поведение полезно в ситуации, когда мы знаем, что хотя в данный момент объект и не нужен, но вскоре вновь появится необходимость в его услугах. Сохранив экземпляр объекта в памяти, мы избавим систему от необходимости разрушения и повторного создания объекта, тем самым сократим временные затраты на рестарт. Для уменьшения счетчика на единицу в параметр `Lock` передается `false`.

Для запуска COM-сервера в зависимости от стоящих перед клиентом задач применяется одна из трех функций библиотеки COM: `CoGetClassObject()`, `CoCreateInstance()` или `CoCreateInstanceEx()`.

Если планируется создать большое количество однотипных объектов или требуется построить объект с помощью интерфейса, отличного от `IClassFactory` (например, `IClassFactory2`), то следует задействовать функцию

```
function CoGetClassObject(const CLSID : TCLSID; dwClsContext : longint;
   ServerInfo:PCoServerInfo; const IID:TIID; var pv) : HResult; stdcall;
```

Функция не способна напрямую создать экземпляр COM-объекта. Вместо этого на свет появляется посредник — фабрика класса, имеющая право собрать нужный COM-объект. В качестве параметров функция рассчитывает получить `CLSID` класса и `IID` интерфейса фабрики классов. Параметр `dwClsContext` ограничивает контекст исполнения компонента, с которым будет работать клиент. Это может быть: `CLSCTX_INPROC_SERVER` — адресное пространство клиентского процесса (в случае DLL); `CLSCTX_LOCAL_SERVER` — адресные пространства разных процессов, исполняющиеся совместно на одном компьютере; или `CLSCTX_REMOTE_SERVER` — адресные пространства процессов, распределенных в компьютерной сети. Параметр `ServerInfo` используется в технологии DCOM, в нем задаются параметры удаленного сервера. Собрав необходимые данные, функция разыскивает в системном реестре путь к серверу COM и запускает его. Нам возвращается ссылка `pv` на фабрику. Полученный указатель на интерфейс передается функции `CreateInstance()` фабрики классов.

Если сервер создается на том же компьютере, на котором выполняется клиент, и мы не планируем создавать несколько экземпляров COM-объекта, то задействуется функция

```
function CoCreateInstance(const clsid : TCLSID; const Outer: IUnknown;
      dwClsContext : longint; const IID:TIID; var pv) : HResult; stdcall;
```

Функция воспользуется помощью рассмотренной ранее функции `CoGetClassObject()` и метода `CreateInstance()` и возвратит указатель непосредственно компонента (в то время как `CoGetClassObject` возвращает указатель для фабрики класса).

Если сервер расположен на удаленном компьютере, то вызову подлежит функция

```
function CoCreateInstanceEx(const clsid : TCLSID; const Outer: IUnknown;
         dwClsContext : longint; ServerInfo:PCoServerInfo;
         dwCount: longint; rgmqResults:PMultiQIArray) : HResult; stdcall;
```

Здесь предусмотрен параметр, хранящий имя удаленного компьютера, на котором располагается интересующий нас сервер.

Информация об удаленном сервере должна находиться в реестре клиентской машины по адресу:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\{AppID_GUID}\RemoteServerName =
<server_name>
```

Функция возвращает массив указателей `rgmqResults`, число ячеек в массиве определяется параметром `dwCount`.

Реализация фабрики класса, класс *TComObjectFactory*

Фабрика класса — это особый COM-объект, единственной целью которого является создание других объектов COM. С этой целью разработчик фабрики закладывает в нее специфические знания, необходимые для создания объекта определенного класса.

Реализующий фабричный интерфейс класс `TComObjectFactory` построен на основе `TObject`. Помимо основного для всех COM-объектов интерфейса `IUnknown` классом поддерживаются еще два, характерных для фабрики классов интерфейса: `IClassFactory` и `IClassFactory2`.

```
type TComObjectFactory = class(TObject, IUnknown, IClassFactory, IClassFactory2);
```

Интерфейс `IClassFactory` может быть использован для создания всех COM-объектов. Интерфейс `IClassFactory2` выступает логическим развитием интерфейса `IClassFactory`, он позволяет управлять созданием экземпляров COM-объектов, для работы которых необходим лицензионный ключ.

В Delphi фабрика класса `TComObjectFactory` создается автоматически COM-сервером в момент инициализации приложения-сервера. Для этого вызывается впечатляющий по числу параметров конструктор

```
constructor Create(ComServer: TComServerObject; ComClass: TComClass;
            const ClassID: TGUID; const ClassName, Description: string;
                  Instancing: TClassInstancing;
                  ThreadingModel: TThreadingModel = tmSingle);
```

Здесь `ComServer` — параметр, связывающий фабрику класса с COM-сервером. Все остальные параметры идентичны ключевым свойствам фабрики (см. табл. 42.1 и 42.2) и определяют особенности создаваемого экземпляра класса.

Обычно команда на создание экземпляра фабрики класса подается в разделе инициализации приложения-сервера (листинг 42.7).

Листинг 42.7. Раздел инициализации

```
//...
initialization
  TTypedComObjectFactory.Create(ComServer, TMyComObject,
                      Class_MyComObject, ciMultiInstance, tmApartment);
end.
```

На первый взгляд может показаться, что конструктор не возвращает никаких результатов своих трудов. На самом деле, сразу после создания фабрика класса регистрируется у менеджера фабрик — глобального объекта `ComClassManager`.

Основные свойства фабрики класса (табл. 42.1) доступны только для чтения. Они инициализируются в момент создания экземпляра фабрики.

Таблица 42.1. *Свойства фабрики класса* `TComObjectFactory`

Свойство	Описание
property ComServer: TComServerObject;	Определяет COM-сервер, владеющий этой фабрикой класса
property ComClass: TClass;	Класс COM-объекта, который будет создавать данный экземпляр фабрики класса. Параметр анализируется методом GetFactoryFromClass() менеджера COM-класса TComClassManager
property ClassID: TGUID;	Идентификатор класса CLSID экземпляра фабрики
property ClassName: **string**;	Имя COM-класса
property Description: **string**;	Текстовое описание объекта
property Instancing: TClassInstancing; **type** TClassInstancing = (ciInternal, ciSingleInstance, ciMultiInstance);	Значение ciInternal уведомляет, что сервер исполняется внутри адресного пространства вызывающего его приложения. Сервер будет недоступен вне породившего его процесса. Значения ciSingleInstance и ciMultiInstance говорят, что сервер функционирует в отдельном процессе. Разница между этими режимами в том, что ciMultiInstance позволяет одному серверу обслуживать несколько клиентов одновременно. Режим ciSingleInstance требует соблюдения правила: каждому клиенту свой COM-сервер
property ProgID: string;	Информационное свойство, хранящее имя разработчика, название класса и его версию
property ThreadingModel: TThreadingModel;	Свойство определяет особенности одновременного вызова COM-объекта несколькими клиентами (табл. 42.2). **type** TThreadingModel = (tmSingle, tmApartment, tmFree, tmBoth, tmNeutral)

Таблица 42.2. *Потоковая модель* `TThreadingModel`

Значение	Описание	Особенности реализации
tmSingle	Сервер не поддерживает многопоточное обслуживание. Одновременное обращение нескольких клиентов к COM-объекту невозможно	Запросы клиентов обрабатываются поочередно
tmApartment	Модель COM гарантирует, что в одно и то же время к объекту может обращаться только один клиент. Все клиенты работают с COM-объектом в том же потоке, в котором он и был создан	Объекты могут работать с собственными данными, но глобальные данные должны быть защищены (например, критическими секциями, мьютексами или семафорами)
tmFree	Объекты могут вызывать любое число потоков в любое время	
tmBoth	Модель Both схожа с Free, за исключением того, что все обратные вызовы выполняются в том же самом потоке	Максимальная производительность и гибкость
tmNeutral	Клиенты могут обращаться к серверу в различных потоках в одно и то же время. Модель COM сама позаботится о том, чтобы одновременно выполняемые запросы не конфликтовали	Модель доступна только при работе с COM+. Программист должен позаботиться о защите глобальных данных. Использование данной модели недопустимо при совместной работе с визуальными элементами управления

Самый главный метод фабрики

```
function CreateComObject(const Controller: IUnknown): TComObject;
```

вызывает конструктор `CreateFromFactory()` обслуживающей данный класс фабрики.

Регистрация класса создаваемого объекта производится процедурой

```
procedure RegisterClassObject;
```

Метод вызывается в момент запуска exe-файла с COM-сервером.

Регистрация (или снятие с регистрации) объекта COM осуществляет метод

```
procedure UpdateRegistry(Register: Boolean);
```

Метод заносит в реестр сведения об объекте COM и путь к исполняемому файлу.

Для обеспечения лицензионной защиты объекта от вездесущих пиратов установите в `true` свойство

```
property SupportsLicensing: Boolean;
```

С этого момента для создания COM-объекта фабрика класса будет требовать ключ. Лицензионная информация о COM-объекте находится в свойстве

```
property LicString: WideString;
```

Если вы хотите, чтобы все происходящие в COM исключительные ситуации сопровождались выводом окна с сообщением об ошибке, то установите в `true` свойство

```
property ShowErrors: Boolean;
```

Идентификационный код интерфейса, с которым связана ошибка, доступен посредством свойства:

```
property ErrorIID: TGUID;
```

Реализация COM-объекта в Delphi

В Delphi COM-объект создается на основе сотрудничества как минимум двух классов (рис. 42.1). Собственно сам COM-объект получается из вполне рядового класса `TComObject`. Это прямой наследник `TObject`, описанный в модуле `ComObj`. Как и большинство других классов VCL, наш новый знакомый обладает свойствами и методами и имеет полное право их реализовывать, т. е. заполнять программным кодом. Для того чтобы обычный класс `TComObject`, целиком и полностью исповедующий каноны ООП, смог претендовать на звание полноценного COM-объекта, ему нужен помощник. В терминах Delphi такой помощник (из-за приставки "Co", добавляемой к имени реального объекта в момент его создания) называется `CoClass`. Задача `CoClass` — создать вокруг стандартного `TComObject` такую оболочку, чтобы со стороны внешнего наблюдателя (клиента COM) наш объект выглядел точно так, как и любой другой объект COM. Благодаря `CoClass` наш объект приобретает всю атрибутику обычного COM-объекта, в том числе произвольное число интерфейсов.

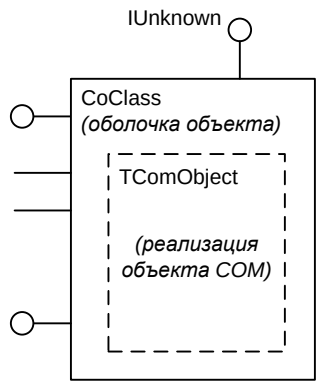


Рис. 42.1. Структура COM-объекта в Delphi

Класс *TComObject*

Основная задача TComObject — наполнение программной логикой интерфейса IUnknown и поддержка основной функциональности COM-объекта. В частности, здесь хранится идентификатор CLSID, имеется механизм построения агрегатов (составных COM-объектов) и поддерживается так необходимый для обработки исключительных ситуаций контроллеров автоматизации интерфейс ISupportErrorInfo.

```
type TComObject = class(TObject, IUnknown, ISupportErrorInfo);
```

Созданием экземпляра класса заведуют три конструктора, основной и наиболее универсальный из них:

```
constructor CreateFromFactory(Factory: TComObjectFactory;
                              const Controller: IUnknown);
```

Параметр Factory определяет фабрику класса, а параметр Controller задействуется только при формировании агрегата.

Сразу после появления на свет объект проходит инициализацию:

```
procedure Initialize;
```

Благодаря свойству

```
property Factory: TComObjectFactory; //только для чтения
```

наш объект помнит об участвующей при его рождении фабрике класса.

Если COM-объект составной, то доступ к интерфейсу IUnknown вложенного в него объекта мы обнаружим в свойстве

```
property Controller: IUnknown;
```

Если объект не составной, то свойство возвратит nil.

Число ссылок на интерфейс COM-объекта находится в свойстве

```
property RefCount: Integer; //только для чтения
```

Управлением ссылками занимается интерфейс IUnknown, поэтому внутри TComObject реализуются три метода: AddRef(), Release() и QueryInterface().

Класс *TTypedComObject*

В иерархии наследования сразу за TComObject следует класс TTypedComObject, позволяющий создавать объекты COM на основе библиотеки типов. Между предком и потомком всего одно отличие — интерфейс IProvideClassInfo.

```
type TTypedComObject = class(TComObject, IProvideClassInfo)
```

Названный выше интерфейс функционирует в интересах единственного введенного в классе метода

```
function GetClassInfo(out TypeInfo: ITypeInfo): HResult; stdcall;
```

Функция предоставляет нам информацию о типах, с этой целью она возвращает указатель на класс-надстройку CoClass в выходной параметр TypeInfo.

> ***Замечание***
>
> Библиотека типов (type library) представляет собой описание имеющихся в распоряжении объектов COM интерфейсов, методов, свойств и структур. Библиотека компилируется в

двоичный файл и может использоваться клиентским приложением, как некий эквивалент справки. Именно благодаря библиотеке типов реализуется механизм раннего связывания, позволяющий будущему клиенту COM узнать все об интерфейсах сервера еще на этапе компиляции.

Класс *TComServer*

Класс `TComServer` описан в модуле `ComServ`. Класс предназначен для создания экземпляров COM, для этого в рамках класса инкапсулировано приложение COM-сервер. Доступ к серверу реализован посредством автоматически создаваемой глобальной переменной `ComServer`. Цепочка наследования класса невелика, `TComServer` построен на основе абстрактного класса `TComServerObject`, а он в свою очередь берет начало от `TObject`.

Модуль с сервером интегрируется в проект самостоятельно сразу после добавления в проект COM-объекта. Программисту не следует задумываться над созданием экземпляра `TComServer`, сервер создается автоматически, а доступ к нему можно получить через глобальную переменную `ComServer:TComServer`.

Имя сервера устанавливается процедурой

```
procedure SetServerName(const Name: string);
```

Если сервер работает совместно с библиотекой типов, то имя сервера назначается в соответствии с установками библиотеки.

Библиотека типов загружается методом

```
procedure LoadTypeLib;
```

Процессом регистрации (и снятия с регистрации) COM-сервера в системном реестре ведает процедура

```
procedure Initialize;
```

В рамках метода `Initialize()` вызывается процедура

```
procedure UpdateRegistry(Register: Boolean);
```

Для регистрации в системном реестре всех входящих в сервер объектов в процедуру передается значение `true`. Обратный результат (стирание данных о COM-объектах в реестре) достигается при передаче `false`.

Из девяти свойств COM-сервера всего одно доступно для редактирования:

```
property UIInteractive: Boolean;
```

Это свойство определяет, должен ли COM-сервер иметь интерактивный интерфейс с пользователем. Интерактивность (состояние `true`) проявляется в информировании сервером пользователя обо всех возникающих исключительных ситуациях, если же сервер выполняется как отдельная служба (сервис), то целесообразно передавать в это свойство значение `false`.

В табл. 42.3 представлен перечень информационных свойств сервера автоматизации.

В классе `TComServer` предусмотрен один-единственный обработчик события

```
property OnLastRelease: TLastReleaseEvent;
type TLastReleaseEvent = procedure(var Shutdown: Boolean) of object;
```

Он призван реагировать на тот момент, когда последний клиент отказывается от услуг сервера. Единственный параметр метода `Shutdown` определяет: прекращать работу сервера (значение `true`) или оставить его в памяти (`false`).

***Таблица 42.3.** Информационные свойства сервера автоматизации `TComServer`*

Свойство	Описание
`property ObjectCount: Integer;`	Указывает число работающих в сервере объектов
`property IsInprocServer: Boolean;`	Значение `true` указывает, что это внутренний сервер
`property ServerFileName: string;`	Полный путь и имя файла сервера
`property HelpFileName: string;`	Имя связанного с модулем файла помощи
`property ServerKey: string;`	Значение `"LocalServer32"` указывает на то, что это исполняемый exe-файл. Значение `"InprocServer32"` — сервер реализован в dll-библиотеке
`property ServerName: string;`	Имя файла сервера
`property StartMode: TStartMode;`	`smAutomation` — сервер запускается в качестве сервера автоматизации в ответ на обращение контроллера автоматизации, `smRegServer` — сервер запущен лишь с целью регистрации в реестре, `smStandalone` — сервер запущен пользователем, `smUnregServer` — приложение удаляется из реестра
`property TypeLib: ITypeLib;`	Обеспечивает доступ к ассоциированной с сервером библиотеке типов

Пример COM-проекта

После такого большого теоретического вступления применим наши знания о COM в практической области. На начальном этапе не станем проявлять излишние амбиции и просто научим COM-сервер простейшему — осуществлять элементарные арифметические операции. Задачи клиентского приложения и того проще — суметь воспользоваться услугами COM-сервера.

Создайте новую папку, например, с названием COM_DEMO. Внутри нее еще две папки — SERVER и CLIENT, соответственно для размещения исходных файлов сервера и клиента. Свою работу мы начинаем с проектирования COM-сервера.

COM-сервер

Создаем новый обычный проект VCL Forms Application. Переименуем главную форму проекта в `frmMain` и сразу же сохраним проект будущего COM-сервера в каталоге *x*:\COM_DEMO\Server. Во время сохранения модуль, соответствующий главной форме, назовем main.pas, а сам проект — mathcomsrv.dproj.

Чтобы приложение COM получило почетное право называться сервером, оно должно обладать хотя бы одним COM-объектом. Для внедрения COM-объекта выберите пункт главного меню **File | New | Other**. В появившемся диалоговом окне **New Items** обратитесь к страничке **ActiveX**, найдите на ней значок **COM Object** и смело нажмите кнопку **OK**. Наши стара-

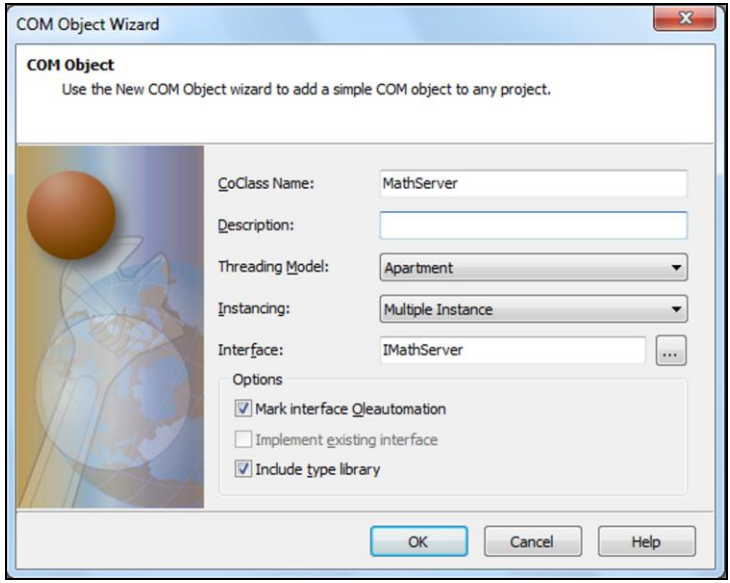


Рис. 42.2. Помощник настройки COM-объекта

ния будут вознаграждены — на экране возникнет новое диалоговое окно определения основных характеристик будущего COM-объекта — **COM Object Wizard** (рис. 42.2).

Помощник настройки COM-объекта

Первое, что потребует от нас помощник **COM Object Wizard**, — определить имя вспомогательного класса **CoClass Name**. Придумываем новое имя, раскрывающее смысл нашей задумки (я предлагаю `MathServer`), и вводим его в самой верхней строке диалогового окна. Заметьте, что, определяя имя класса, я не использую принятый для классов VCL префиксный символ "T".

Вторая строка предназначена для комментариев разработчика.

Третья строка диалога **COM Object Wizard** определяет модель потоков — особенность одновременного вызова COM-объекта несколькими клиентами. Это седьмой параметр `ThreadingModel` метода `Create()` класса `TComObjectFactory` (см. табл. 42.2). В нашем примере мы воспользуемся моделью `Free`.

В четвертой строке окна помощника указываются особенности запуска содержащего COM-объект приложения-сервера. Помощник предлагает нам выбрать один из трех вариантов поведения: **Internal**, **Single Instance** и **Multiple Instances**. Это не что иное, как шестой параметр конструктора `Create()` фабрики класса, соответствующий типу данных `TClassInstancing`. Возможные варианты были описаны ранее в табл. 42.1. В нашем примере лучше всего воспользоваться режимом **Multiple Instances**, допускающим создание неограниченного числа COM-объектов в приложении-сервере.

Заключительная строка диалогового окна **COM Object Wizard** содержит имя интерфейса, она заполняется автоматически — интерфейс получает имя `IMathServer`.

Оставьте установленным флажок **Mark interface Oleautomation**. Хотя мы и не нацелены на создание сервера автоматизации, это действие упростит регистрацию интерфейсов в системном реестре.

При включенном флажке **Include type library** помощник сгенерирует библиотеку типов, содержащую всю информацию об интерфейсах COM-объекта. Эти данные упростят нашу работу при проектировании клиентского приложения, поэтому не снимайте "галочку" с элемента управления.

После настройки всех первоначальных характеристик COM-объекта щелкните по кнопке **OK**, и помощник сформирует шаблон кода.

Шаблон кода с описанием класса

Сразу после нажатия кнопки **OK** диалогового окна **COM Object Wizard** среда Delphi примется за работу.

- На экране нашего компьютера появится окно редактора библиотеки типов с заголовком `mathcomsrv.ridl`. Для Delphi файл в формате RIDL (Restricted Interface Definition Language) относительная новинка, он появился в 2009-й версии. Предыдущие версии среды работали с бинарным файлом в формате TLB.
- К исходному коду нашего проекта добавится еще один модуль (по умолчанию он назван `Unit1.pas`) с описанием нового класса `TMathServer`.

Вновь даем команду на сохранение всего проекта. В процессе сохранения переименуем невзрачное `Unit1` в название, отражающее предназначение модуля — `Unit_TMathServer.pas`, тем самым мы оставим себе подсказку, что указанный модуль содержит реализацию класса `TMathServer`. Остальные модули не переименовывайте и сохраняйте с именами, предлагаемыми системой.

На первое время редактор библиотеки типов оставим в покое, к нему мы вернемся немного позднее, а пока уделим внимание шаблону `Unit_TMathServer` с описанием класса объекта `TMathServer` (листинг 42.8).

> ***Замечание***
>
> Директива `{$WARN SYMBOL_PLATFORM OFF}` указывает компилятору на то, что следующий за ней код может работать на всех известных Delphi платформах.

Листинг 42.8. Шаблон математического сервера

```
unit Unit_TMathServer;

{$WARN SYMBOL_PLATFORM OFF}

interface
uses Windows, ActiveX, Classes, ComObj, mathcomsrv_TLB, StdVcl;

type
  TMathServer = class(TTypedComObject, IMathServer)
  end;

implementation
uses ComServ;

initialization
  TTypedComObjectFactory.Create(ComServer, TMathServer, Class_MathServer,
    ciMultiInstance, tmFree);
end.
```

Сразу обратите внимание на строку `uses` со списком подключаемых модулей. Некоторые имена библиотек нам пока не знакомы:

- модуль `AciveX` содержит определения основных интерфейсов, например фабрики классов `IClassFactory`;
- модуль `ComObj` включает описание базовых для модели COM-классов: `TComObject`, `TComObjectFactory`, `TComServerObject` и т. п.;
- модуль `StdVcl` содержит объявления стандартных интерфейсов библиотеки VCL;
- модуль `mathcomsrv_TLB` автоматически создается средой программирования и содержит информацию о фабрике классов, интерфейсе COM-объекта и их GUID.

В начале раздела `implementation` к проекту подключается еще один модуль — `ComServ`. В этом модуле описан один-единственный класс `TComServer`, инкапсулирующий глобальный объект COM-сервер. Доступ к объекту обеспечивается через глобальную переменную `ComServer`.

В модуле декларируется описанный нами в диалоге настройки COM-объекта класс `TMathServer`, он формируется на базе `TTypedComObject` и предназначен для обслуживания "одноименного" (пока не определенного) интерфейса `IMathServer`. Раздел инициализации содержит строку, ответственную за создание фабрики класса `TTypedComObjectFactory`. Синтаксис конструктора не вполне стандартный — мы привыкли, что конструктор Delphi возвращает созданный им объект, здесь на первый взгляд такого не происходит. Вместо этого, втайне от нас, фабрика класса регистрируется у менеджера фабрик. Для этого в теле конструктора фабрики спрятан метод `ComClassManager.AddObjectFactory(self)`.

Библиотека типов

Одной из проблем, возникающих перед программистом во время изучения COM-объекта, становится поиск поясняющей документации. Переливающийся всеми цветами радуги класс может оказаться бесполезным, если мы не понимаем, как с ним обращаться. Так, при проектировании COM-приложения клиентское приложение обязано владеть полной информацией об интерфейсной части COM-сервера, иначе оно не сможет воспользоваться его услугами. В частности, клиент COM должен знать о:

- инкапсулированных в сервере COM-объектах;
- списке интерфейсов, поддерживаемых каждым из объектов;
- составе функций в каждом из интерфейсов;
- перечне аргументов функций.

Наилучшим хранилищем знаний такого рода считается *библиотека типов*. Библиотека типов не выступает обязательной частью компонента COM, но она в состоянии значительно упростить работу пользователя COM-объекта, в особенности когда идет речь о компоненте, созданном сторонним производителем на другом языке программирования. Кроме того, помимо справочной информации, на основании данных из библиотеки могут генерироваться необходимые заголовочные файлы.

Среда разработки Delphi вооружена редактором библиотеки типов (рис. 42.3). Для вызова редактора в главном меню Delphi найдите пункт **View | Type Library**. Если вернуться к нашему примеру, то после обращения к редактору в дереве объектов библиотеки типов мы увидим два объекта: интерфейс `IMathServer` и оболочку будущего COM-объекта — сокласс `MathServer`.

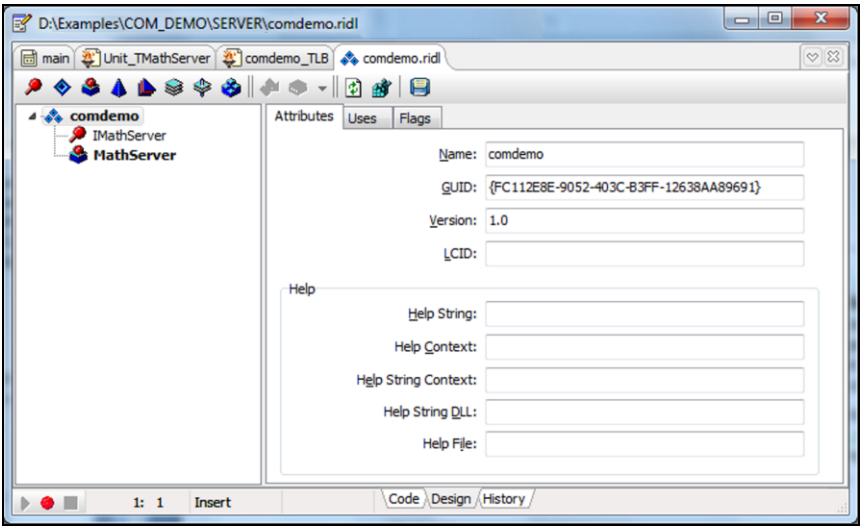


Рис. 42.3. Редактор библиотеки типов

Рассмотрим, какую помощь сможет оказать библиотека типов при создании нового метода. Как и все настоящие математики, свою работу начнем с реализации операции сложения двух чисел. Тогда прототип будущей функции будет выглядеть примерно так:

```
function Addition(X: Double; Y: Double;
                  out Outcome: OleVariant): HResult; stdcall;
```

У метода имеются два входных параметра — `X` и `Y`, они предназначены для передачи суммируемых чисел. В случае успеха метод возвратит результат сложения в выходной параметр `Outcome`. Заметьте, что в технологии COM исходящие параметры проще типизировать как `OleVariant`.

Выделите узел интерфейса `IMathServer` и нажмите кнопку **New Method**. Под узлом `IMathServer` появится дочерний элемент — прототип будущей функции. Переименуйте этот элемент в **Addition** — это название новой функции. Для нового метода системой сразу назначается новый идентификатор ID=101 (страница атрибутов метода **Attributes**). Для определения параметров перейдите к странице **Parameters**. Воспользовавшись кнопкой **Add**, добавьте три строки с параметрами функции (рис. 42.4). Результат сложения мы возвращаем в выходной параметр `OutCome`, а сам метод возвращает значение типа `HResult`.

> ***Внимание!***
>
> Все COM-методы должны возвращать значение типа `HRESULT`. В случае успешного завершения метод вернет значение `S_OK`, значение `S_FALSE` свидетельствует о том, что метод не может выполнить операцию, если интерфейс не поддерживается — `E_INTERFACE`, катастрофическая ошибка — `E_UNEXPECTED`, и т. п.

На панели управления редактора нажмите кнопку **Refresh Implementation**. Это действие заставит редактор библиотеки типов объявить метод в интерфейсе `IMathServer` и включить его в перечень методов класса `TMathServer`, отвечающего за реализацию нового метода.

> ***Замечание***
>
> Выходные параметры интерфейсных методов должны объявляться в виде указателей.

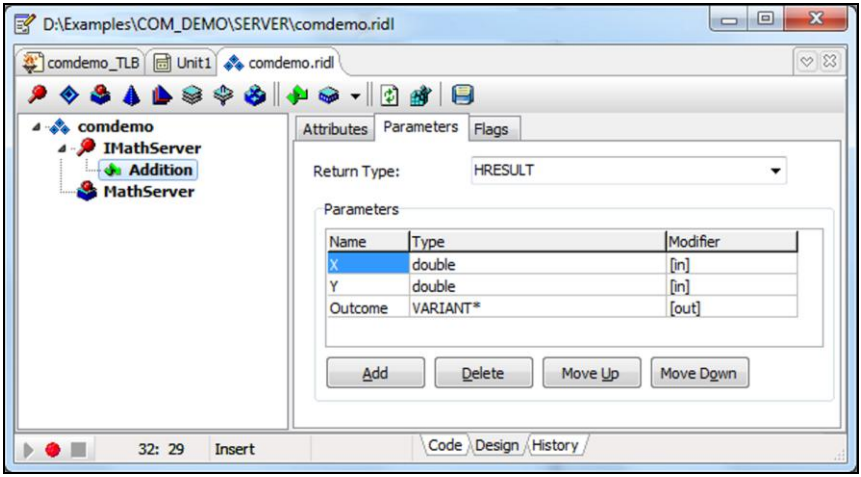


Рис. 42.4. Редактор библиотеки типов с новым методом `Addition()`

Давайте проверим, насколько хорошо потрудился редактор библиотеки и типов. Код файла mathcomsrv_TLB.pas (в нем находится описание интерфейса `IMathServer`) представлен в листинге 42.9, для большей краткости из него исключены комментарии, автоматически создаваемые Delphi.

Листинг 42.9. Результаты работы редактора библиотеки типов

```
unit mathcomsrv_TLB;
{$TYPEDADDRESS OFF}  {$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON} {$VARPROPSETTER ON} {$ALIGN 4}
interface
uses Windows, ActiveX, Classes, Graphics, OleServer, StdVCL, Variants;

const
  mathcomsrvMajorVersion = 1;
  mathcomsrvMinorVersion = 0;
  LIBID_mathcomsrv: TGUID = '{FC112E8E-9052-403C-B3FF-12638AA89691}';
  IID_IMathServer: TGUID = '{457E9417-CBC2-47C3-8DA8-4AEECADF34FB}';
  CLASS_MathServer: TGUID = '{F24AA614-54E5-4525-9B13-AF46E9B87C55}';
type
  IMathServer = interface;
  MathServer = IMathServer;
  IMathServer = interface(IUnknown)
    ['{457E9417-CBC2-47C3-8DA8-4AEECADF34FB}']
   function Addition(X: Double; Y: Double;
                     out Outcome: OleVariant): HResult; stdcall;
   end;

  CoMathServer = class
    class function Create: IMathServer;
    class function CreateRemote(const MachineName: string): IMathServer;
  end;
```

```
implementation
uses ComObj;

class function CoMathServer.Create: IMathServer;
begin
  Result := CreateComObject(CLASS_MathServer) as IMathServer;
end;
class function CoMathServer.CreateRemote(const MachineName: string): IMathServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_MathServer) as IMathServer;
end;
end.
```

В разделе `type` модуля у интерфейса `IMathServer` появилось объявление заголовка метода `Addition()`. Как и положено для интерфейсов, в модуле вы не найдете ни строчки кода, отвечающей за реализацию метода.

Сразу за описанием интерфейса следует объявление класса-оболочки `CoMathServer`. Напомню (см. рис. 42.1), что он предназначен для создания надстройки над реализующим интерфейс классом `TMathServer`. В разделе `implementation` модуля мы обнаружим два метода фабрики класса, предназначенные для запуска COM-сервера: `CreateRemote()` — на удаленном компьютере; `Create()` — на том же компьютере с приложением COM-клиента.

За реализацию интерфейса `IMathServer` в нашем проекте отвечает класс `TMathServer`. Откройте в редакторе кода листинг с описанием класса (модуль `Unit_TMathServer`). В секции защищенных объявлений `protected` появился заголовок нашей новой функции, а в секции `implementation` описана заготовка ее реализации. Дополните листинг единственной строкой `Outcome=X+Y` (листинг 42.10).

Листинг 42.10. Доработка модуля `Unit_TMathServer`

```
unit Unit_TMathServer;
{$WARN SYMBOL_PLATFORM OFF}

interface
uses  Windows, ActiveX, Classes, ComObj, mathcomsrv_TLB, StdVcl;

type  TMathServer = class(TTypedComObject, IMathServer)
  protected
    function Addition(X, Y: Double;
                      out Outcome: OleVariant): HResult; stdcall;
  end;

implementation
uses ComServ;

function TMathServer.Addition(X, Y: Double;
                              out Outcome: OleVariant): HResult;
begin
  Outcome:=X+Y; {реализация метода}
end;
```

```
initialization
  TTypedComObjectFactory.Create(ComServer, TMathServer, Class_MathServer,
    ciMultiInstance, tmFree);
end.
```

Научившись описывать методы, попробуем свои силы на определении свойств. Допустим, что нам надо объявить свойство, способное обслуживать целочисленное значение.

Для определения нового свойства в редакторе типов выделите узел с интерфейсом `IMathServer` и на панели управления программой найдите кнопку **New Property**. Щелчок по кнопке приведет к созданию двух новых одноименных узлов **Property1**. Такое "раздвоение личности" свойства объясняется тем, что один из узлов отвечает за чтение данных из свойства, второй узел решает обратную задачу — записывает в свойство данные.

Переименуйте любой из узлов нового свойства, например в **IntValue**. Тип данных, описываемых свойством, определяется на странице **Attributes** в раскрывающемся списке **Type**. Для решения нашей задачи подойдет значение по умолчанию — целое число `long` (рис. 42.5).

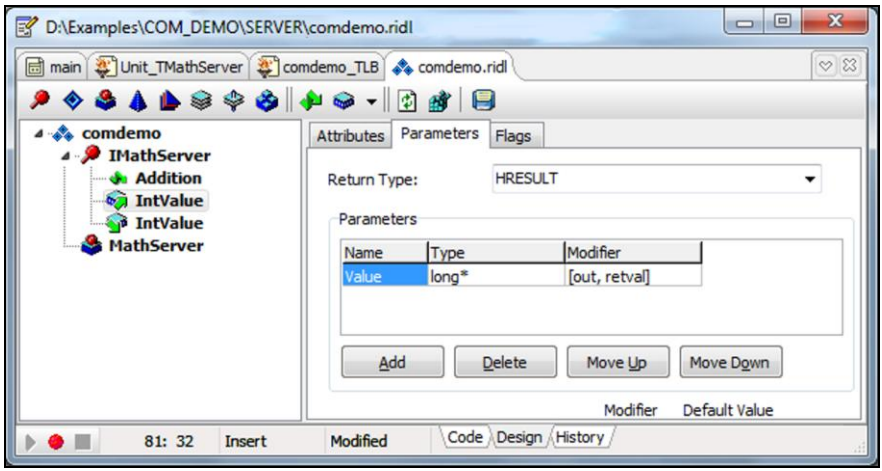


Рис. 42.5. Создание свойства

Чтение значения из свойства и запись нового значения в свойство производится посредством вызова соответствующих методов. Имена методов формируются по принципу:

- чтение из свойства: `"Get_"+<Имя_свойства>`;
- запись в свойство: `"Set_"+<Имя_свойства>`.

Страница **Parameters** определяет параметры для соответствующих методов `Get...` и `Set...`

Для того чтобы необходимые изменения попали в модули описания интерфейса и реализации, вновь на панели управления редактора библиотеки типов нажмите кнопку **Refresh Implementation**.

После этого в модуле `Unit_TMathServer` мы найдем объявление и реализацию методов чтения из свойства и записи в свойство (листинг 42.11). Хранилищем значения выступит поле `fIntValue`.

Листинг 42.11. Объявление свойства в модуле Unit_TMathServer

```
type
  TMathServer = class(TTypedComObject, IMathServer)
  protected
    fIntValue:integer;
    function Addition(X, Y: Double;
                      out Outcome: OleVariant): HResult; stdcall;
    function Get_IntValue(out Value: Integer): HResult; stdcall;
    function Set_IntValue(Value: Integer): HResult; stdcall;
  end;
```

Нам осталось заполнить методы чтения/записи кодом из листинга 42.12.

Листинг 42.12. Чтение и запись значения в свойство

```
function TMathServer.Get_IntValue(out Value: Integer): HResult;
begin
  Value:=fIntValue; //читаем свойство
end;

function TMathServer.Set_IntValue(Value: Integer): HResult;
begin
  fIntValue:=Value; //заносим данные в свойство
end;
```

> ***Внимание!***
>
> Занесенное в свойство значение доступно только подключенному к COM-серверу клиенту и хранится только на время сеанса. Поэтому свойства интерфейса не могут использоваться для обмена данными между различными клиентами.

Главная форма сервера

По большому счету, разрабатываемый нами сервер поддержки математических операций не нуждается в визуальных элементах управления. Все запрашиваемые клиентом услуги реализуются в памяти и не требуют отображения на экране. С другой стороны, наличие в проекте сервера формы может оказать хорошую услугу для администратора системы, информируя его о текущем состоянии COM-сервера. Для реализации задуманного нам потребуется помощь трех компонентов: многострочного редактора `Memo1`, метки `Label1` и таймера `Timer1`.

В листинге 42.13 представлен фрагмент кода main.pas, предлагающий вариант применения информационных свойств сервера.

Листинг 42.13. Работа с информационными свойствами сервера

```
unit main;
...
var frmMain: TfrmMain;

implementation
 uses ComServ;
{$R *.dfm}
```

```
procedure TfrmMain.FormShow(Sender: TObject);
begin
  with Memo1.Lines do
  begin
    Clear;
    Add('Имя: '+#9+ComServer.ServerName);
    Add('Тип: '+#9+ComServer.ServerKey);
    Add('Имя файла: '+#9+ComServer.ServerFileName);
    case ComServer.StartMode of
      smAutomation : Add('Режим запуска — по запросу клиента COM');
      smStandalone : Add('Режим запуска — запущен пользователем');
    end;
  end;
end;

procedure TfrmMain.Timer1Timer(Sender: TObject);
begin
  Label1.Caption:=Format('Активных клиентов %d',[ComServer.ObjectCount]);
end;
end.
```

Регистрация сервера

Запустите исполняемый файл сервера. В идеальной ситуации в момент своего первого запуска COM-сервер автоматически регистрируется сам и регистрирует свои COM-объекты в системном реестре Windows. Это операция должна осуществляться в момент инициализации приложения — вызов метода `Application.Initialize()`.

Однако требования политики безопасности современных версий Windows (в первую очередь речь идет о Windows Vista, Windows Server 2008 и Windows 7) активно препятствуют такому своеволию со стороны приложения, разработанного не в стенах Microsoft. Поэтому для гарантированной регистрации COM-сервера на компьютере лучше воспользоваться командной строкой, запустив с ее помощью сервер с ключом `/regserver` (рис. 42.6). В этом случае операционная система беспрекословно выполнит наши указания.

Для регистрации серверов, представленных в виде динамических библиотек, целесообразно воспользоваться стандартной утилитой Windows RegSvr32.exe. Регистрация COM производится в ветви реестра:

`HKEY_CLASSES_ROOT\CLSID\{`*идентификатор COM*`}`

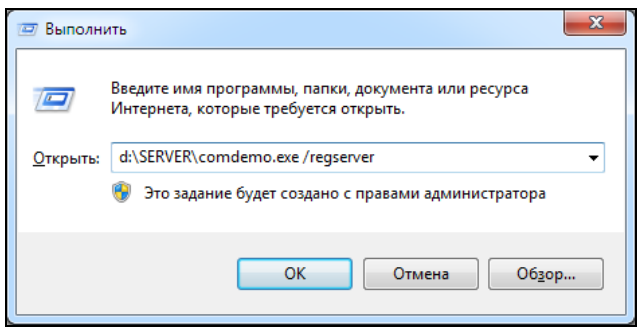


Рис. 42.6. Регистрация COM-сервера из командной строки

В названном подразделе сохраняются две наиболее важные составляющие: имя файла сервера и его CLSID. Кроме того, в подраздел **LocalServer32** передается путь к исполняемому файлу сервера, **TypeLib** — идентификатор библиотеки типов, **Version** — версия сервера (рис. 42.7).

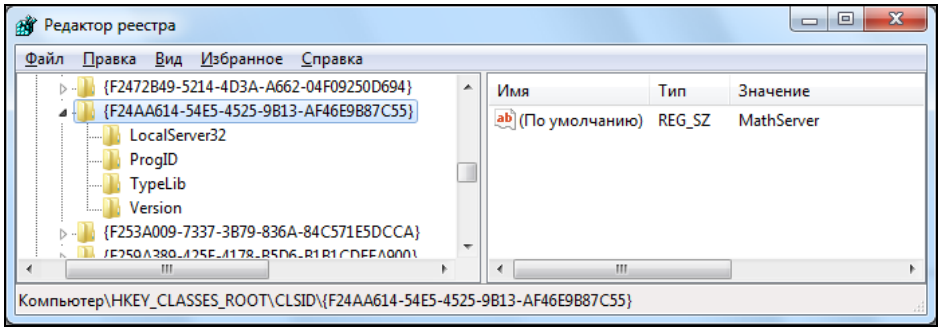


Рис. 42.7. Ветвь реестра с регистрационными данными COM-сервера

Интерфейсы регистрируются в ветви:

`HKEY_CLASSES_ROOT\Interface\{`*идентификатор интерфейса*`}`

> ***Внимание!***
> Для удаления из системного реестра всех регистрационных данных COM-сервера, его COM-объектов и интерфейсов запустите приложение-сервер из командной строки с ключом `/unregserver`.

В современных версиях Delphi для регистрации (или снятия с регистрации) COM-сервера можно воспользоваться пунктами меню **Run | ActiveX Server | Register** и **Run | ActiveX Server | Unregister**.

COM-клиент

В сравнении с COM-сервером процесс создания клиентского приложения удивительно прост. Он включает один обязательный этап — подключение к приложению описания интерфейсной части входящих в состав сервера COM-объектов. После этого нам остается лишь обращаться к нужному интерфейсу. Проверим это утверждение на практике. Для этого создадим новое приложение VCL и сохраним его в каталоге CLIENT под именем MathClient.dproj.

Импорт библиотеки типов

Для того чтобы клиентское приложение могло воспользоваться услугами COM-сервера, оно нуждается в знаниях об интерфейсной части сервера. Эта информация сохранена нами в каталоге приложения-сервера в библиотеке типов — файле mathcomsrv.ridl. В Delphi предусмотрено несколько способов импорта. В Delphi XE2 простейший из них — с помощью помощника импорта компонентов. Выбрав пункт меню **Component | Import Component...**, мы должны ответить на ряд вопросов помощника (рис. 42.8).

1. В первом окне выбираем пункт **Import a Type Library** и нажимаем кнопку **Next**.
2. Во втором окне помощника находим интересующий нас сервер. В нашем случае это `mathcomsrv`. Если в перечне доступных библиотек вы не смогли обнаружить требуемый сервер, то повторите операцию регистрации приложения-сервера.

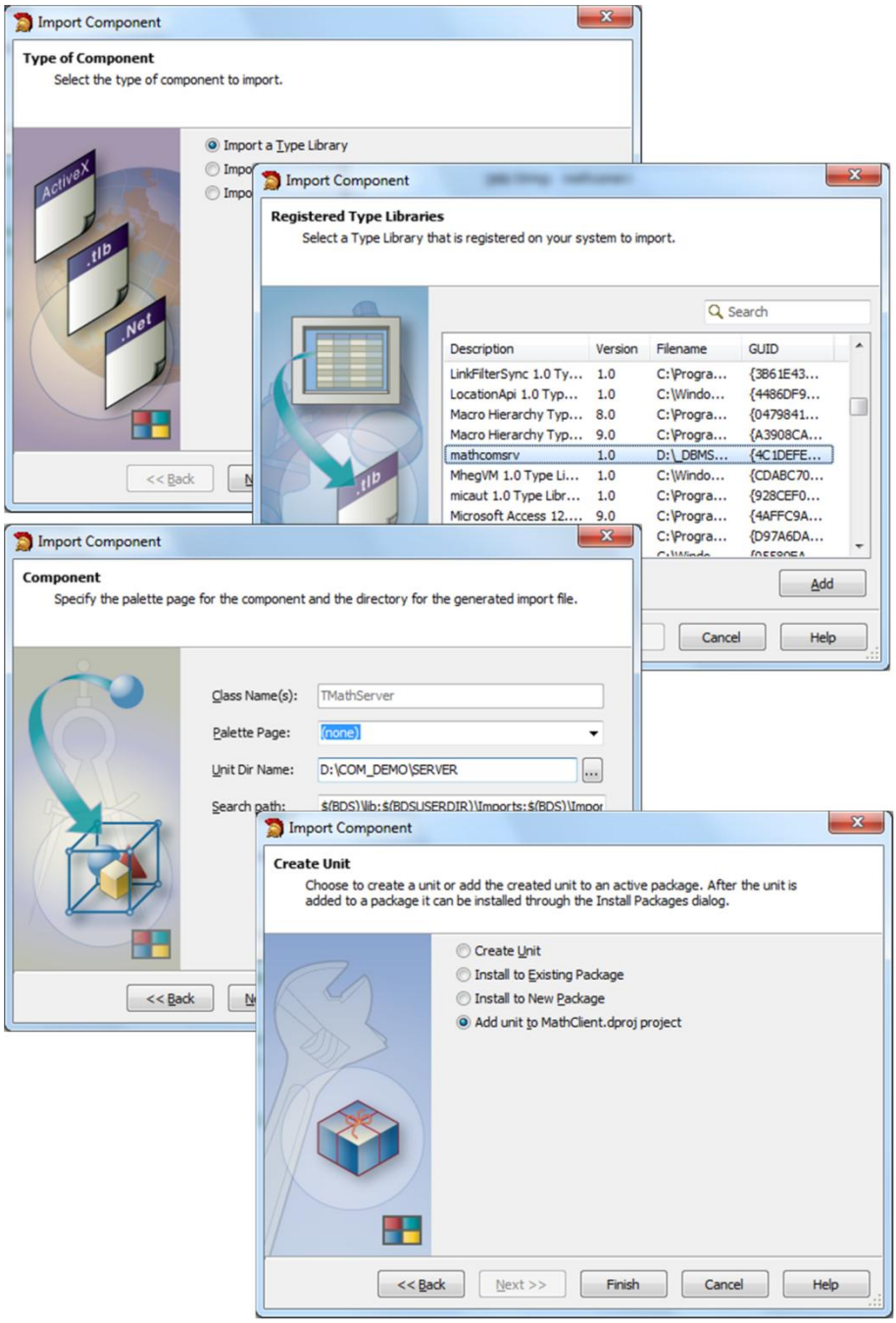


Рис. 42.8. Этапы импорта библиотеки типов

3. В третьем окне снимаем флажок **Generate Component Wrapper**. Этим действием мы проинформируем Delphi, что не собираемся генерировать новый компонент VCL. Также можно изменить путь к модулю, для этого стоит настроить элемент управления **Unit Dir Name**. Остальные элементы управления оставляем в состоянии по умолчанию.

4. В заключительном 4-м окне выделяем переключатель **Add unit to MathClient.dproj project**, тем самым уведомляем Delphi, что в результате импорта библиотека типов должна быть подключена к проекту клиента.

В результате к клиентскому приложению будет подключен уже знакомый нам по COM-серверу интерфейсный модуль `mathcomsrv_TLB`.

Обращение к COM-объекту

Обращение к методу `Addition()` из клиентского приложения предложено в листинге 42.14.

Листинг 42.14. Вызов метода сложения

```
function _Addition(X, Y: double): double;
var Outcome: OleVariant;
    I : IMathServer;
begin
  try
    I:=CoMathServer.Create;    //создаем объект и получаем его интерфейс
    I.Addition(X,Y,Outcome);   //обращаемся к методу сложения
    Result:=Outcome;           //получаем результат
    I:=nil;                    //освобождаем интерфейс
  except
    on EIntfCastError do
       raise Exception.Create('Интерфейс не поддерживается!');
  end;
end;
```

Для работы со свойством `IntValue` в интерфейсе предусмотрены два метода: `Set_IntValue()` и `Get_IntValue()`. Пример работы представлен в листинге 42.15.

Листинг 42.15. Обращение к свойству

```
var I : IMathServer;
    Value:integer;
begin
  i:=CoMathServer.Create;
  i.Set_IntValue(1);     //присвоим свойству значение
  i.Get_IntValue(value); //читаем данные из свойства
  i:=nil;
  ShowMessage(IntToStr(value));
end;
```

Напомню, что хранящиеся в свойствах интерфейса данные доступны только на время сессии клиента с сервером. После отключения клиента от сервера свойства обнуляются. Поэтому, если услуги сервера необходимы на протяжении всего времени работы клиента, подключение к интерфейсу должно производиться в момент запуска клиента, а отключение — в момент останова. Если же клиент обращается к серверу от случая к случаю, подключение и отключение надо осуществлять как можно быстрее — в рамках одной процедуры.

ГЛАВА 43

Автоматизация

Рассмотренный в предыдущей главе способ работы с COM-объектом не всегда приводит программистов в восторг. Почему? Мы уже знаем, что интерфейс предоставляет доступ к объекту не напрямую, а через таблицу адресов указателей на функции. Писать код, опираясь на указатели, весьма неудобно, в особенности разработчику, привыкшему к доброжелательному языку Delphi, который оперирует компонентами с помощью свойств и методов. С этим утверждением согласны не только начинающие программисты, но даже и ряд макроязыков, на дух не переносящих указатели. Именно по этой причине практически одновременно с COM на свет появилась надстройка над многокомпонентной моделью, получившая название *"автоматизация"* или Automation.

Безусловно, у механизма раннего связывания (напомню, что так называется способ разработки приложений COM, когда вызов методов интерфейсов производится исключительно с помощью указателей) достоинств больше, чем недостатков. Два неоспоримых преимущества классического подхода к проектированию приложений COM заключаются:

- в жестком контроле над корректностью кода клиентских приложений еще на этапе компиляции;
- в высокой скорости вызова запрашиваемых клиентом функций.

Этими достоинствами автоматизация похвастаться не может, но она более удобна при проектировании клиентов COM.

Как и все наследники идей COM, технология автоматизации нацелена на предоставление сервиса от одних приложений к другим. Все нововведения автоматизации неразрывно связаны с внедрением в технологию COM двух новых элементов:

- принципиально нового вида интерфейса `IDispatch`, умеющего вызывать методы сервера автоматизации по их имени;
- механизма *позднего связывания* (late binding). Благодаря позднему связыванию соединение между методом интерфейса объекта COM и контроллером автоматизации устанавливается не во время проектирования, а в момент выполнения приложения. Как следствие, на этапе разработки клиентского приложения отпадает необходимость применения библиотеки типов.

В итоге автоматизация стала позволять обращение к методам COM-объектов по их имени, а не только при посредничестве указателей.

> ***Примечание***
>
> Автоматизация поддерживается подавляющим большинством приложений из обоймы программного обеспечения Microsoft. В первую очередь это все приложения из состава Microsoft Office и Internet Explorer.

Интерфейс *IDispatch*

Сервер автоматизации (automation server) представляет собой объект COM, в котором реализован интерфейс динамического вызова `IDispatch`. Анонсируемый интерфейс используется клиентским приложением, а если говорить строго — *контроллером автоматизации* (automation controller), чтобы получить доступ к требуемому сервису сервера.

> ***Замечание***
>
> Основная заслуга `IDispatch` в том, что этот интерфейс позволяет клиентскому приложению обращаться к интересующему его методу COM-объекта просто по его имени.

По своей сути интерфейс `IDispatch` (листинг 43.1) очень близок к классическим интерфейсам обычного COM, т. к. является наследником базового интерфейса `IUnknown`.

Листинг 43.1. Объявление интерфейса `IDispatch`

```
IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
                         out TypeInfo): HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
                           NameCount, LocaleID: Integer;
                           DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
                    LocaleID: Integer; Flags: Word; var Params;
              VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall;
  end;
```

Функция `GetTypeInfoCount()` позволит узнать, возвращает ли COM-объект информацию о типе во время выполнения. В этом случае в параметре `Count` окажется 1.

При работе с библиотекой типов может пригодиться метод `GetTypeInfo()`. Он возвращает указатель `TypeInfo` на интерфейс `ITypeInfo` библиотеки типов.

За преобразование имени метода `Names` в идентификатор `DispID` (диспетчерский идентификатор метода) отвечает функция `GetIDsOfNames()`. Она возвращает указатель `DispIDs` на массив, содержащий идентификаторы запрашиваемых методов.

Метод `Invoke()` позволит контроллеру автоматизации обратиться к интересующему его методу сервера. Для этого контроллер передает идентификатор `DispID` требуемого метода интерфейса. В параметр `IID` направляется идентификатор интерфейса, в котором объявлен нужный метод. Необязательный параметр `LocaleID` предназначен для работы в многоязычных проектах. В перечне `Flags` (допустимые флаги `DISPATCH_METHOD`, `DISPATCH_PROPERTYGET`, `DISPATCH_PROPERTYPUT` и `DISPATCH_PROPERTYPUTREF`) конкретизируется способ вызова метода. Параметры вызываемого метода доступны с помощью указателя `Params` (он ссылается на структуру `TDispParams`).

Указатель на результат выполнения метода вызываемого интерфейса находится в параметре `VarResult`.

Последняя пара параметров метода `Invoke()` окажется полезной при возникновении ошибки. В этом случае в `ExcepInfo` помещается указатель на структуру с описанием ошибки, а параметр `cNamedArgs` уточнит индекс некорректного параметра.

Диспинтерфейсы и дуальные интерфейсы

Для того чтобы объект COM смог претендовать на высокое звание объекта автоматизации, возможностей одного интерфейса `IDispatch` ему недостаточно. В таком объекте объявляется дополнительный интерфейс, называемый *диспетчерским интерфейсом* (dispinterface) или просто *диспинтерфейсом*. Диспинтерфейсы являются наследниками `IDispatch`, их основная задача — продекларировать методы, которые будут доступны контроллерам автоматизации.

Доступ к методам диспетчерского интерфейса осуществляется при посредничестве метода `Invoke()`, который выбирает требуемый метод по его целочисленному идентификатору `dispid`. Чуть позже, в примере сервера автоматизации, мы создадим диспинтерфейс `IlbAutoDisp`, все методы и единственное свойство которого снабжены идентификаторами (листинг 43.2).

Листинг 43.2. Объявление интерфейса `IlbAutoDisp`

```
type  IlbAutoDisp = dispinterface
    ['{1925C8CD-011E-44EF-8881-37DC9A5DF9FA}']
    procedure AddLine(Line: OleVariant); dispid 201;
    procedure DeleteLine(Index: Integer); dispid 202;
    procedure SortLines; dispid 203;
    property  Count: Integer readonly dispid 204;
  end;
```

Разработчику не стоит менять идентификаторы, чтобы случайно не затронуть зарезервированные системой значения. Среда Delphi (во взаимодействии с редактором библиотеки типов) самостоятельно позаботится о присвоении диспетчерским идентификаторам необходимых значений (обычно нумерация начинается с 201).

Самым существенным недостатком диспинтерфейса считается его медлительность, ведь процесс обращения к методу по его имени более трудоемок, чем через указатель. Поэтому в проектах автоматизации активно эксплуатируется еще одна разновидность интерфейсов — *дуальные интерфейсы* (dual interfaces).

Дуальные интерфейсы наиболее универсальны, т. к. поддерживают оба способа доступа к своим методам:

- скоростной — с помощью указателей на интерфейсы;
- удобный, но "тихоходный" способ обращения к методу по его имени.

Контроллер автоматизации без применения библиотеки типов

Если вы еще никогда не сталкивались с разработкой приложений на основе технологии автоматизации, то на первых порах приемы программирования вам покажутся весьма необычными. Допустим, что нашему приложению требуется получить информацию о составе таблиц базы данных Microsoft Access. Предложенная далее процедура предоставит нам список `List` всех таблиц, принадлежащих БД, из файла с именем `DBName` (листинг 43.3).

Листинг 43.3. Объявление интерфейса IlbAutoDisp

```
uses comobj;
...
procedure TForm1.GetAccessDBTables(DBName: string; List: TStrings);
var A:OleVariant;
    i:integer;
begin
  try
    A:=CreateOleObject('Access.Application');
    A.OpenCurrentDatabase(DBName);
    for i:=0 to A.CurrentData.AllTables.Count - 1 do
                    List.Add(A.CurrentData.AllTables[i].Name);
  finally
    A:=Unassigned;
  end;
end;
```

В самом начале листинга создается объект автоматизации — приложение Microsoft Access, однако вместо идентификатора CLSID в функцию `CreateOleObject()` направляется текстовая строка `"Access.Application"`.

```
function CreateOleObject(const ClassName: string): IDispatch;
```

Результат выполнения функции передается в переменную `A`. Дальше начинается самое интересное. Вполне понятно, что у переменной, пусть даже универсальной, `OleVariant` не может быть ни свойств, ни методов. Однако буквально в следующей строке кода мы обращаемся к функции `OpenCurrentDatabase()` так, будто этот метод открытия базы данных Access всегда принадлежал переменной `A`. И, что самое удивительное, не встречаем никаких возражений со стороны Delphi! Почему так происходит?

Поняв, что переменная `A` предназначена для взаимодействия с сервером автоматизации, компилятор Delphi не превращает вызов методов в обычный бинарный код, а оставляет в коде исполняемого файла формальное описание, содержащее названия функций и их аргументов. В этом несложно убедиться, открыв откомпилированный exe-файл в любом редакторе, поддерживающем текстовое представление данных. Вы без труда найдете названия методов автоматизации (рис. 43.1). Когда приложение запускается на выполнение, сохраненные строки просто передаются интерфейсу `IDispatch`, именно он (а не компилятор Delphi) произведет синтаксический разбор команд и запустит их на выполнение.

```
00074D20: FF FF FF FF 12 00 00 00|41 00 63 00 63 00 65 00 | яяяя....A.c.c.e.
00074D30: 73 00 73 00 2E 00 41 00|70 00 70 00 6C 00 69 00 | s.s...A.p.p.l.i.
00074D40: 63 00 61 00 74 00 69 00|6F 00 6E 00 00 00 00 00 | c.a.t.i.o.n.....
00074D50: 01 01 00 08 4F 70 65 6E|43 75 72 72 65 6E 74 44 | ....OpenCurrentD
00074D60: 61 74 61 62 61 73 65 00|01 00 00 43 6F 75 6E 74 | atabase....Count
00074D70: 00 00 00 00 01 00 00 41|6C 6C 54 61 62 6C 65 73 | .......AllTables
00074D80: 00 00 00 00 01 00 00 43|75 72 72 65 6E 74 44 61 | .......CurrentDa
00074D90: 74 61 00 00 01 00 00 4E|61 6D 65 00 02 01 00 03 | ta.....Name.....
00074DA0: 41 6C 6C 54 61 62 6C 65|73 00 00 00 55 8B EC 6A | AllTables...U<mj
```

Рис. 43.1. Фрагмент бинарного кода контроллера автоматизации с поздним связыванием

Замечание

Способ работы с COM-объектом, когда связывание имен свойств и методов объекта с их кодом осуществляется не на этапе компиляции, а на этапе выполнения программы, называют *поздним связыванием* (late binding). В отличие от раннего связывания, здесь отпадает необходимость в применении библиотеки типов.

Некая "отрешенность" компилятора Delphi от процесса проверки корректности кода программы имеет простое объяснение — Delphi ничего не знает о возможностях сервера автоматизации. Ведь мы не используем библиотеку типов. Можете проверить это утверждение и добавить в предложенный ранее пример строку с любой нелепостью, скажем, `A.HelloWorld`, и Delphi без всякого зазрения совести включит это в исполняемый код. Все будет хорошо ровно до тех пор, пока мы не запустим проект на выполнение и не "поприветствуем" сервер автоматизации командой "HelloWorld". Интерфейс `IDispatch` не имеет чувства юмора, поэтому, не найдя запрашиваемого метода, с удовольствием сгенерирует исключительную ситуацию `EOleError`. Выводом из всего сказанного будет то, что работая по законам технологии автоматизации, мы должны быть абсолютно уверены в каждом символе своего кода.

Контроллер автоматизации с поддержкой библиотеки типов

Наиболее распространенной проблемой, с которой сталкиваются программисты, создающие контроллеры автоматизации, опирающиеся на сервис серверов сторонних разработчиков, является нехватка информации о возможностях сервера. Нельзя сказать, что компании-разработчики ПО специально скрывают от нас описания своих интерфейсов. Просто выходящая из-под пера служб поддержки программного обеспечения документация рассчитана на среднестатистического потребителя, которому выражение "дуальный интерфейс" сродни тарабарщине.

Безусловно, выход из положения имеется. В нашем случае (кроме обязательного поиска документации для профессионального разработчика в недрах Всемирной паутины) рекомендую воспользоваться услугами библиотеки типов, входящей в состав сервера автоматизации. Допустим, что нашему приложению требуется воспользоваться услугами одного из лучших редакторов векторной графики CorelDRAW фирмы Corel Corporation.

Обращаемся к меню **Component | Import Component...** и в окне импортирования **Import Component** устанавливаем переключатель **Import a Type Library** (рис. 43.2). Нажатие кнопки **Next** выведет на экран перечень зарегистрированных в системе библиотек типов. Разработчику предстоит выбрать интересующую его библиотеку. В следующем окне изменим только параметр **Unit Dir Name**, определяющий путь к папке, в которую импортируется файл с библиотекой. Укажите здесь путь к каталогу вашего проекта автоматизации и вновь нажмите кнопку **Next**. В заключительном окне предлагается выбрать форму импорта, в нашем случае предлагаю ограничиться пунктом **Create Unit**. Это действие укажет помощнику, что разработчик рассчитывает получить в свое распоряжение программный модуль с описанием интерфейсов сервера автоматизации.

В результате Delphi предоставит в распоряжение программиста модуль, содержащий полное описание интерфейсов и методов сервера автоматизации. Все остальное — дело техники. Мы подключаем к проекту контроллера всю библиотеку типов или только избранные фрагменты (требуемые для стоящих перед приложением задач) так, как предложено в следующем примере вызова редактора CorelDRAW (листинг 43.4).

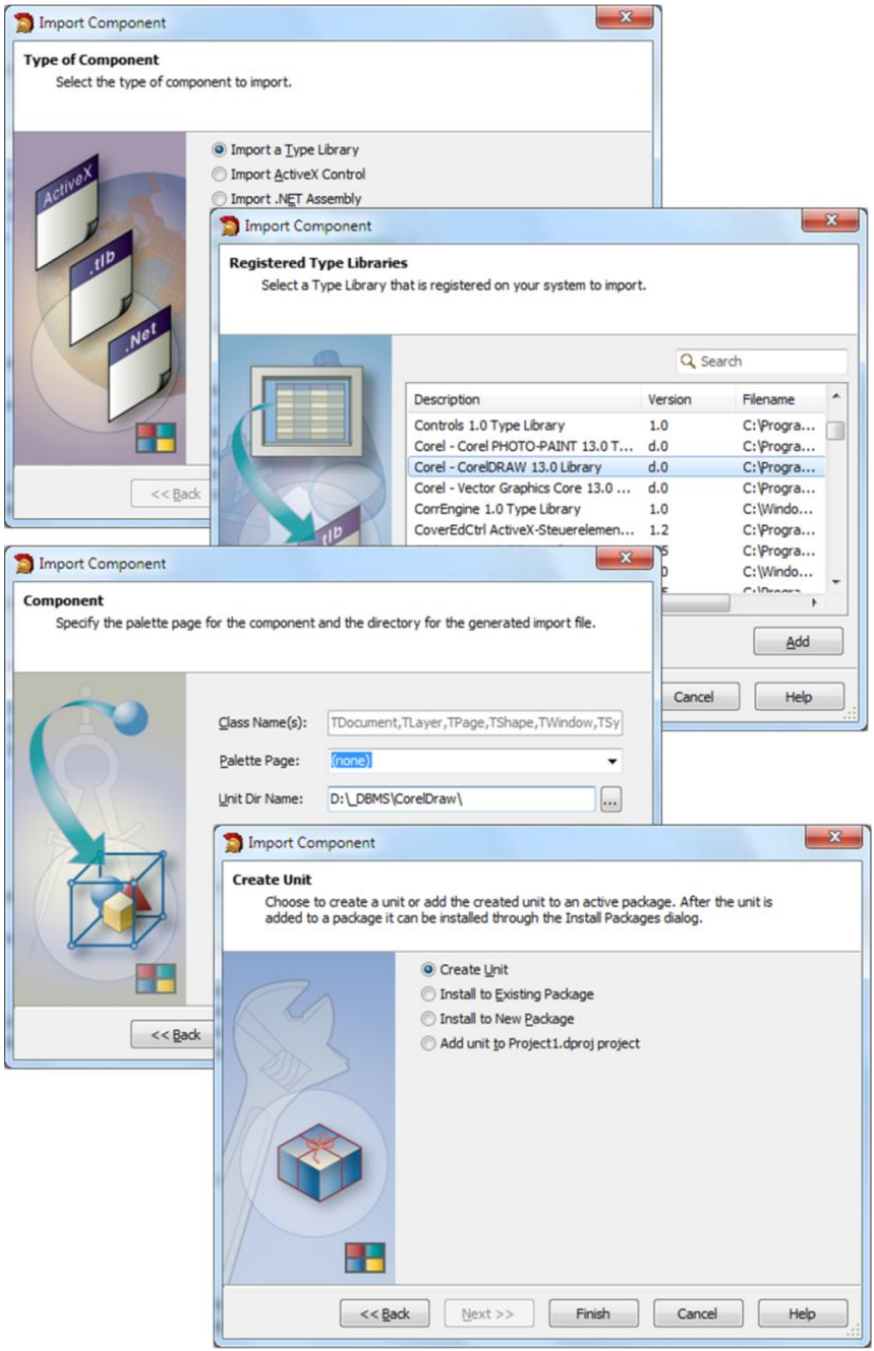


Рис. 43.2. Импорт библиотеки типов

Листинг 43.4. Пример вызова редактора CorelDRAW

```
unit Unit1;
interface

uses Windows, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  StdCtrls;

type TForm1 = class(TForm)
    Button1: TButton;
    //...
  procedure Button1Click(Sender: TObject);
   //...
  end;

const
CLASS_Application: TGUID = '{4A110001-2109-4B5D-BC63-17AE8914E384}';

type IDrawApplication = interface(IDispatch)
    ['{398C0002-8D24-11D2-89E7-0000861EBBD6}']
    //...
    function OpenDocument(const FileName: WideString;
                          CodePage: Integer): IDrawDocument; safecall;
    function CreateDocument: IDrawDocument; safecall;
    //...
end;

type  IDrawApplicationDisp = dispinterface
    ['{398C0002-8D24-11D2-89E7-0000861EBBD6}']
    //...
    property Visible: WordBool dispid 1610743810;
    //...
end;

var Form1: TForm1;
implementation
uses ComObj;
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var IDA:IDrawApplicationDisp;
begin
  {создаем объект автоматизации и получаем доступ к интерфейсу}
  IDA:=CreateComObject(CLASS_Application) as IDrawApplicationDisp;
  IDA.Visible:=True; //включаем видимость приложения
  //...
  IDA.Quit; //завершаем работу с приложением
  IDA:=nil; //отключение от сервера
end;
end.
```

На этот раз обращение к серверу автоматизации осуществляется с помощью функции CreateComObject(), в которую направляется идентификатор класса. Вместе с тем можно воспользоваться альтернативным вариантом вызова — по текстовому имени сервера:

```
IDA:=CreateOleObject('CorelDraw.Application') as IDrawApplicationDisp;
```

Оба варианта возвращают диспинтерфейс приложения — IDrawApplicationDisp.

Рассмотренные ранее способы более эффективны по сравнению с обращением к серверу автоматизации через переменную OleVariant, т. к. основаны на технологии раннего связывания. Однако если вы не намерены применять в проекте клиента библиотеку типов, то можно пойти по более простому пути (листинг 43.5).

Листинг 43.5. Альтернативный вариант вызова редактора CorelDRAW

```
var CorelDraw:OleVariant;
begin
  CorelDraw:=CreateOleObject('CorelDraw.Application');
  //...
  CorelDraw:=UnAssigned;
end;
```

ЗАМЕЧАНИЕ

Разработка программного обеспечения с поддержкой дуальных интерфейсов считается наиболее оптимальной, т. к. позволяет клиентскому процессу обращаться к объекту COM по наиболее предпочтительному для него варианту.

Сервер автоматизации, базовый класс *TAutoObject*

Основу любого сервера автоматизации составляет объявленный в модуле ComObj (или OleAuto) объект автоматизации TAutoObject. Объект автоматизации не требует явного подключения к проекту, он (точнее, наследник класса TAutoObject) присоединяется к нему автоматически после того, как программист ответит на вопросы в диалоговом окне мастера **Automation Object Wizard** (см. рис. 43.4). Вопросов не столь много, следует указать:

- имя класса-оболочки **CoClass Name**, который создается совместно с объектом автоматизации;
- предпочтительную потоковую модель **Threading Model** (о ней мы уже говорили в табл. 42.2), определяющую способ взаимодействия контроллеров и сервера автоматизации;
- особенности запуска сервера **Instancing** (см. табл. 42.1);
- поддержку интерфейса событий автоматизации **Generate Event support code**.

Позднее, при разработке демонстрационного проекта, мы задействуем, мастер создания сервера автоматизации, а пока продолжим изучение теоретических аспектов класса TAutoObject.

Ключевая особенность создаваемого на основе TAutoObject объекта автоматизации — поддержка интерфейса IDispatch (рис. 43.3). Благодаря IDispatch будущий сервер автоматиза-

ции позволит обращаться к его свойствам и методам не только с помощью указателей строго в соответствии с правилами определенными в библиотеке типов, но и в упрощенной форме — по текстовым именам. Для этих целей в классе реализуются 4 метода интерфейса `IDispatch` (листинг 43.6).

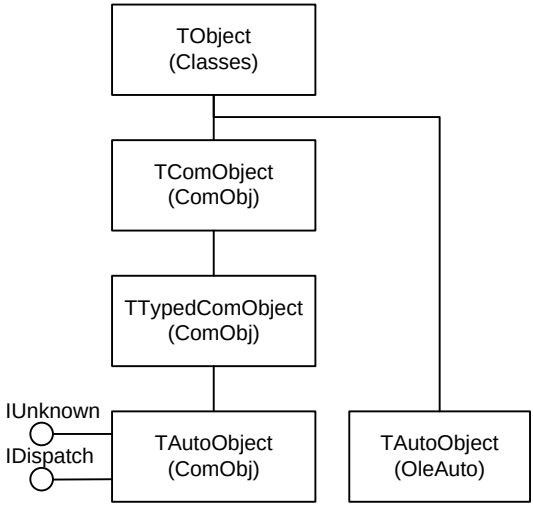


Рис. 43.3. Место класса `TAutoObject` в иерархии VCL

ЗАМЕЧАНИЕ

В VCL сосуществуют два одинаковых по названию и функциональному назначению, но разных по построению класса `TAutoObject`. Класс `TAutoObject` из модуля `ComObj` построен на основе длинной цепочки предков и самостоятельно реализует интерфейс `IDispatch`. Класс `TAutoObject` из модуля `OleAuto` вместо прямой реализации интерфейса пользуется услугами инкапсулированного в него класса `TAutoDispatch = class(IDispatch)`. Далее в этой главе мы сосредоточим свое внимание на `TAutoObject` из модуля `ComObj`.

Еще одно достижение класса — поддержка событий автоматизации. Это тот случай, когда сервер автоматизации получает возможность организации обратного вызова для передачи обслуживаемого им клиента какой-то информации.

Листинг 43.6. Объявление `TAutoObject` в модуле `ComObj`

```
type TAutoObject = class(TTypedComObject, IDispatch)
 private
    FEventSink: IUnknown;
    FAutoFactory: TAutoObjectFactory;
 protected
 {методы интерфейса IDispatch }
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
                           NameCount, LocaleID: Integer;
                           DispIDs: Pointer): HResult; virtual; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
                         out TypeInfo): HResult; virtual; stdcall;
    function GetTypeInfoCount(out Count: Integer): HResult; virtual; stdcall;
```

```
    function Invoke(DispID: Integer; const IID: TGUID;
            LocaleID: Integer; Flags: Word; var Params;
       VarResult, ExcepInfo, ArgErr: Pointer): HResult; virtual; stdcall;
 {методы поддержки событий автоматизации и фабрики класса}
    procedure EventConnect(const Sink: IUnknown; Connecting: Boolean);
    procedure EventSinkChanged(const EventSink: IUnknown); virtual;
    property  EventSink: IUnknown read FEventSink write FEventSink;
 {ссылка на фабрику классов}
    property  AutoFactory: TAutoObjectFactory read FAutoFactory;
  public
    constructor Create;
    constructor CreateAggregated(const Controller: IUnknown);
    constructor CreateFromFactory(Factory: TComObjectFactory;
                                  const Controller: IUnknown);
    procedure Initialize; override;
  end;
```

Единственный метод, опубликованный в секции `public`, отвечает за инициализацию объекта. В рамках этого метода мы опишем параметры запуска объекта.

Регистрация сервера автоматизации в таблице ROT

Вполне вероятно, что к моменту старта клиента автоматизации серверное приложение окажется уже загруженным в адресное пространство и будет выполнять задачи, поставленные другим клиентом. Для того чтобы исключить запуск еще одного экземпляра сервера и подключиться к имеющемуся в памяти экземпляру контроллеру автоматизации, целесообразно воспользоваться (объявленной в модулях `ComObj` и `OleAuto`) функцией

```
function GetActiveOleObject(const ClassName: string): IDispatch;
```

В качестве единственного параметра передается идентифицирующая объект константа строкового типа. В случае удачи (объект автоматизации действительно размещен в памяти и готов к сотрудничеству) функция вернет его интерфейс `IDispatch`.

Не исключена ситуация, что логика работы программного обеспечения станет допускать совместную работу на одном компьютере двух (или более) идентичных серверов автоматизации. К какому из серверов будет подключено клиентское приложение?

Чтобы ответить на поставленный вопрос, упомянем пару функций, объявленных в модуле `ActiveX` и имеющих прямое отношение к организации доступа к активному серверу автоматизации. Для того чтобы контроллеры автоматизации поняли, что COM-объект является активным, его целесообразно регистрировать в глобальной *таблице исполняемых объектов* (Running Object Table, ROT). Для этого опытный программист в секции инициализации сервера вызовет функцию

```
function RegisterActiveObject(unk: IUnknown; const clsid: TCLSID;
                              dwFlags: Longint;
                              out dwRegister: Longint): HResult; stdcall;
```

Здесь: `unk` — указатель на интерфейс `IUnknown` регистрируемого объекта COM; `clsid` — идентификатор класса; `dwFlags` — флаги регистрации (`ACTIVEOBJECT_STRONG` — признак, что регистрируемый COM-объект будет на постоянной основе выступать в роли активного объекта, `ACTIVEOBJECT_WEAK` — допускается смена активного объекта); функция возвращает

dwRegister — указатель на зарегистрированный объект, он нам пригодится всего один раз — при обращении к функции

```
function RevokeActiveObject(dwRegister: Longint;
                            pvReserved: Pointer): HResult; stdcall;
```

Функция обычно вызывается в момент выгрузки сервера автоматизации, ее единственная задача — исключение COM-объекта из таблицы ROT.

События автоматизации

Взаимодействие клиента и сервера в технологии COM оказалось бы неполноценным, если бы оно было только односторонним. Представьте себе, что клиент просит сервер осуществить какую-то трудоемкую операцию, например загрузить объемный файл, отсортировать строки в таблице, найти в неупорядоченном списке заданные значения или что-нибудь в подобном роде. В случае если клиент намерен взаимодействовать с сервером в синхронном режиме, то при выполнении длительных операций клиентское приложение будет вынуждено дожидаться до тех пор, пока сервер завершит выполнение поставленной задачи. Такой подход далеко не эффективен, т. к. в период ожидания клиент не сможет выполнять другие операции. В подобных ситуациях обращение к серверу автоматизации должно осуществляться в асинхронном режиме. Таким образом клиент (дав серверу задание) занимается своими делами до тех пор, пока сервер не уведомит клиента о том, что "джин" справился с поручением.

Убедившись в острой необходимости введения канала обратной связи между сервером и клиентом, обсудим способ его организации. Он таков — в момент завершения выполнения поставленной задачи у сервера COM генерируется событие автоматизации, о возникновении которого уведомляется клиент.

Мы с вами уже достаточно хорошо знакомы с постулатами COM, чтобы без особого труда догадаться, что события автоматизации (впрочем, как и все в COM) определяются с помощью интерфейсов. Но, если речь ведется об интерфейсах событий автоматизации, сервер не просто разрешает клиенту обращаться к своим интерфейсам, а можно сказать, вручает клиенту специальный *исходящий интерфейс* (outgoing interface), извещающий клиента о событиях. А теперь самое интересное. Для того чтобы клиент смог воспользоваться этим интерфейсом, он должен самостоятельно реализовать этот интерфейс. Согласитесь, что на первый взгляд это кажется необычным, ведь до сих пор реализацией интерфейсов занимался только сервер COM.

> ***ЗАМЕЧАНИЕ***
>
> Чтобы сервер автоматизации стал поддерживать события автоматизации, в мастере Automation Object Wizard следует установить флажок **Generate Event support code**.

В классе TAutoObject для передачи контроллеру автоматизации информации о событиях подготовлено специальное свойство EventSink. Названное свойство отвечает за предоставление исходящего интерфейса, с помощью которого объект автоматизации отправляет клиенту всю информацию о происходящих в нем событиях. За реализацию исходящего (из объекта автоматизации) интерфейса отвечает контроллер автоматизации. Для этого в клиентском приложении создают объект-получатель, называемый *стоком* (sink).

Для "оборудования" сервера автоматизации механизмом уведомления клиента о событиях автоматизации нам потребуется помощь нескольких интерфейсов (табл. 43.1).

Таблица 43.1. *Интерфейсы обслуживания событий автоматизации*

Интерфейс	Назначение
IConnectionPointContainer	Базовый исходящий интерфейс сервера. Предоставляет доступ к интерфейсу точки подключения IConnectionPoint и интерфейсу-перечислению IEnumConnectionPoints. Интерфейс реализуется классом TConnectionPoints
IConnectionPoint	Интерфейс точки подключения (connection point). Предоставляет контроллеру доступ к конкретному исходящему интерфейсу сервера автоматизации. Интерфейс реализуется классом TConnectionPoint
IEnumConnections	Интерфейс-перечисление, информирующий контроллер обо всех активных на данный момент соединениях
IEnumConnectionPoints	Интерфейс-перечисление. Позволяет контроллеру автоматизации опросить все поддерживаемые сервером исходящие интерфейсы

Явным признаком того, что сервер способен выступать источником событий является наличие у него исходящего интерфейса IConnectionPointContainer. Если дословно перевести название интерфейса, то получится *контейнер точек подключения*. Контейнер предоставляет программный способ доступа к точке доступа, описываемой еще одним интерфейсом — IConnectionPoint.

Контейнер точек подключения IConnectionPointContainer объявлен в модуле ActiveX и обладает всего двумя методами (листинг 43.7).

Листинг 43.7. Объявление интерфейса IConnectionPointContainer

```
type IConnectionPointContainer = interface
    ['{B196B284-BAB4-101A-B69C-00AA00341D07}']
    function EnumConnectionPoints(
             out Enum: IEnumConnectionPoints): HResult; stdcall;
    function FindConnectionPoint(const iid: TIID;
             out cp: IConnectionPoint): HResult; stdcall;
end;
```

Метод EnumConnectionPoints() позволяет клиенту COM опросить все имеющиеся у сервера исходящие интерфейсы. Второй метод FindConnectionPoint() предоставляет клиенту возможность обратиться к нужному внешнему интерфейсу, для этого он возвращает клиенту точку подключения в виде IConnectionPoint.

В среде Delphi интерфейс контейнера IConnectionPointContainer реализуется в рамках описанного в модуле AxCtrls класса TConnectionPoints. Единственное свойство класса

```
property Controller: IUnknown;
```

предоставляет доступ к базовому интерфейсу IUnknown контроллера автоматизации.

Методов у TConnectionPoints также не много. Как уже говорилось ранее, класс реализует методы интерфейса EnumConnectionPoints() и FindConnectionPoint(). Еще одним примечательным методом класса является функция

```
function CreateConnectionPoint(const IID: TGUID;
                               Kind: TConnectionKind;
                           OnConnect: TConnectEvent): TConnectionPoint;
```

отвечающая за создание точки соединения `TConnectionPoint` с клиентом автоматизации. Для этого метод инкапсулирует обращение к конструктору точки соединения. Здесь: `IID` — GUID исходящего интерфейса точки соединения; `Kind` — особенность оповещения клиента о событии автоматизации (если в параметр передано значение `ckSingle`, то данные о событии направляются только в адрес клиента, вызвавшего это событие, а значение `ckMulti` говорит о том, что оповещению подлежат все выполняющиеся клиентские приложения); `OnConnect` — событие, применяемое в методе `Adivise()` клиентского интерфейса `IConnectionPoint`.

Интерфейс точки подключения обладает пятью методами (листинг 43.8).

Листинг 43.8. Объявление интерфейса `IConnectionPoint`

```
type IConnectionPoint = interface
    ['{B196B286-BAB4-101A-B69C-00AA00341D07}']
    function GetConnectionInterface(out iid: TIID): HResult; stdcall;
    function GetConnectionPointContainer(
                out cpc: IConnectionPointContainer): HResult; stdcall;
    function Advise(const unkSink: IUnknown;
                out dwCookie: Longint): HResult; stdcall;
    function Unadvise(dwCookie: Longint): HResult; stdcall;
    function EnumConnections(
                out Enum: IEnumConnections): HResult; stdcall;
end;
```

Рассмотрим методы интерфейса точки подключения в порядке их объявления. Функция `GetConnectionInterface()` возвращает идентификатор исходящего интерфейса точки подключения. Ее коллега, функция `GetConnectionPointContainer()` возвращает управляющий интерфейс контейнера точек подключения. Наиболее интересен третий по счету метод `Adivise()`. Благодаря ему организуется взаимодействие между сервером и клиентом. Ссылка на интерфейс клиента передается через параметр `unkSink`. Название параметра не случайно, по сложившейся традиции интерфейс клиента, предназначенный для связи с сервером, называют *стоком* (sink). Метод возвращает маркер `wCookie`, который однозначно определяет это подключение. Функция `Unadvise()` сводит на нет все труды ее предшественницы, разрывая связь между клиентом и сервером. Кстати, все это происходит не без помощи маркера `wCookie`. Функция `EnumConnections()` предназначена для опроса всех активных (выполненных с помощью метода `Advise()`) подключений. Результат возвращается в виде перечисления в формате еще одного интерфейса (листинг 43.9).

Листинг 43.9. Объявление интерфейса `IEnumConnections`

```
type  IEnumConnections = interface
    ['{B196B287-BAB4-101A-B69C-00AA00341D07}']
    function Next(celt: Longint; out elt;
                  pceltFetched: PLongint): HResult; stdcall;
    function Skip(celt: Longint): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out Enum: IEnumConnections): HResult; stdcall;
end;
```

Интерфейс IEnumConnections способен поочередно обращаться ко всем доступным активным подключениям. Для этого предназначен метод Next(). В результате обращения к методу данные о подключении помещаются в выходной массив elt, в свою очередь элемент массива описывается структурой TConnectData (листинг 43.10).

Листинг 43.10. Объявление интерфейса IEnumConnections

```
type TConnectData = record
    pUnk: IUnknown; //интерфейс IUnknown строка контроллера автоматизации
    dwCookie: Longint;//маркер
end;
```

Для возврата к первому элементу перечисления вызывают метод Reset(). Метод Skip() позволяет пропустить элемент перечисления. Последний метод интерфейса Clone() создает дубликат перечисления.

Нам осталось упомянуть интерфейс-перечисление IEnumConnectionPoints. Мы его оставили на закуску потому, что перечень и назначение имеющихся в его распоряжении методов (Next, Skip, Reset и Clone) идентичен методам интерфейса IEnumConnections. Единственная разница между интерфейсами в том, что IEnumConnectionPoints позволяет перечислить все поддерживаемые сервером интерфейсы, а IEnumConnections работает избирательно, обслуживая лишь активные на данный момент интерфейсы событий.

Фабрика класса объекта автоматизации

Появление на свет объекта автоматизации тесно связано с деятельностью фабрики класса TAutoObjectFactory. Фабрика создает экземпляр класса TAutoObject и устанавливает начальные значения объекта. Основной метод фабрики — ее конструктор

```
constructor Create(ComServer: TComServerObject;
                   AutoClass: TAutoClass;
                   const ClassID: TGUID;
                   Instancing: TClassInstancing;
                   ThreadingModel: TThreadingModel = tmSingle);
```

Конструктор фабрики классов автоматизации схож с рассмотренным в предыдущей главе конструктором своего предка — фабрики TComObjectFactory. Для того чтобы вспомнить основные параметры конструктора, возвратитесь к табл. 42.1 и 42.2.

ЗАМЕЧАНИЕ

Среда разработки Delphi как всегда идет навстречу программисту и самостоятельно включает в проект сервера автоматизации строки кода, запускающие фабрику.

Пример проекта автоматизации с поддержкой событий

Если вы с успехом справились с примером обычного COM-проекта из предыдущей главы, то процесс разработки сервера и клиента автоматизации не вызовет особых затруднений. Поставим перед собой задачу разработать сервер, способный поддержать обмен текстовыми

данными между клиентами автоматизации. Для этого создайте новый каталог, например AUTOMATION_DEMO. Внутри него — еще два каталога: Server и Client соответственно для размещения исходных файлов сервера и клиента. Свою работу мы начинаем с проектирования COM-сервера.

Сервер автоматизации

Создайте новый проект на основе VCL. Переименуйте главную форму проекта в `frmMainServer` и сохраните проект будущего COM-сервера в каталоге *x*:\AUTOMATION_DEMO\Server. При этом модуль, соответствующий главной форме, назовите MainServer.pas, а головной файл проекта — automsrv.dproj.

Разместите на главной форме сервера единственный компонент — список `ListBox1`.

Для того чтобы обычный проект превратился в сервер автоматизации, подключим к нему объект автоматизации. Для этого находим пункт меню **File | New | Other** и в появившемся окне выбираем значок **Automation Object**.

Следующим шагом станет заполнение полей мастера объекта автоматизации **Automation Object Wizard** (рис. 43.4). В первую очередь требуется ввести имя класса-оболочки `CoClass`, я предлагаю остановиться на `lbAuto`. После ввода названия класса мастер самостоятельно (добавив к имени `CoClass` префикс "I") предложит название для интерфейса. Модель потоков **Threading Model** и особенности запуска сервера **Instancing** оставляем со значениями по умолчанию. В заключении следует поставить флажок **Generate Event support code** — это действие укажет мастеру на то, что проектируемый сервер намерен поддерживать события автоматизации.

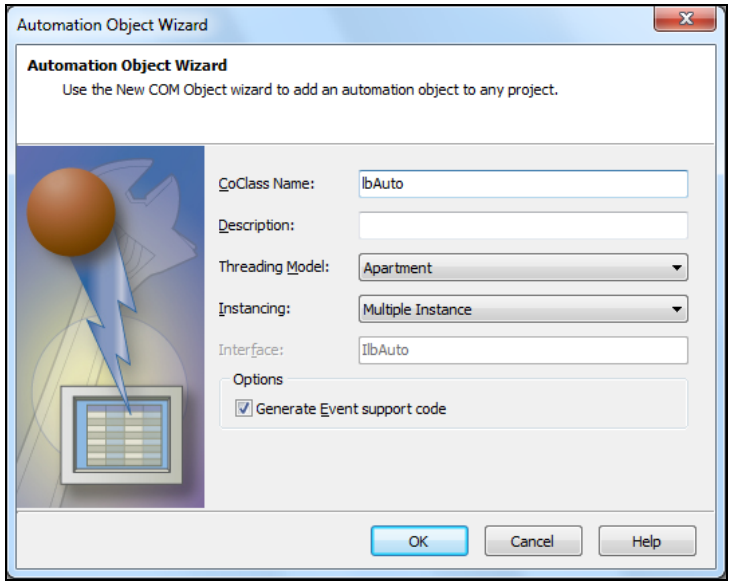


Рис. 43.4. Помощник настройки объекта автоматизации

Нажатие кнопки **OK** в окне мастера приведет к созданию нескольких программных модулей:

- automsrv.ridl — файл, предназначенный для хранения библиотеки типов. Формат появился в последних версиях Delphi и призван заменить файл в формате TLB;

- automsrv_TLB.pas — представление библиотеки типов в классическом для Delphi формате;
- unit1.pas — описание класса `TlbAuto`, он отвечает за реализацию интерфейса сервера автоматизации. При сохранении проекта переименуйте этот файл в lbAuto_Unit.pas.

Изучим созданные Delphi файлы. Рассмотрение начнем с `automsrv_TLB`. Модуль представляет библиотеку типов, доступ к которой вы найдете среди элементов управления в окне редактора библиотеки типов (листинг 43.11).

Листинг 43.11. Исходный код библиотеки типов `automsrv_TLB`

```
unit automsrv_TLB;
{$TYPEDADDRESS OFF}    {$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}   {$VARPROPSETTER ON}        {$ALIGN 4}
interface
uses Windows, ActiveX, Classes, Graphics, OleServer, StdVCL, Variants;

const
  automsrvMajorVersion = 1;  automsrvMinorVersion = 0;
  LIBID_automsrv: TGUID = '{276611A6-0CF5-4BDB-8961-D4AE6992FEB2}';
  IID_IlbAuto: TGUID = '{1925C8CD-011E-44EF-8881-37DC9A5DF9FA}';
  DIID_IlbAutoEvents: TGUID = '{2A3757AF-2C0F-444A-8A2C-E30C6AFE632A}';
  CLASS_lbAuto: TGUID = '{D513D936-A323-4AFC-A7EE-31964D220F94}';
type
  IlbAuto = interface;
  IlbAutoDisp = dispinterface;
  IlbAutoEvents = dispinterface;
  lbAuto = IlbAuto;
  IlbAuto = interface(IDispatch)
    ['{1925C8CD-011E-44EF-8881-37DC9A5DF9FA}']
  end;
  IlbAutoDisp = dispinterface
    ['{1925C8CD-011E-44EF-8881-37DC9A5DF9FA}']
  end;
  IlbAutoEvents = dispinterface
    ['{2A3757AF-2C0F-444A-8A2C-E30C6AFE632A}']
  end;
  ColbAuto = class
    class function Create: IlbAuto;
    class function CreateRemote(const MachineName: string): IlbAuto;
  end;
implementation
uses ComObj;

class function ColbAuto.Create: IlbAuto;
begin
  Result := CreateComObject(CLASS_lbAuto) as IlbAuto;
end;
```

```
class function ColbAuto.CreateRemote(const MachineName: string): IlbAuto;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_lbAuto) as IlbAuto;
end;
end.
```

В интересах нашего проекта сервера в библиотеке типов описаны три интерфейса:

- основной интерфейс `IlbAuto`, сформированный на основе `IDispatch`;
- диспинтерфейс `IlbAutoDisp`. Ключевое слово `dispinterface` говорит о том, что методы данного интерфейса могут вызываться не только по указателю (раннее связывание), но и по имени (позднее связывание);
- исходящий диспинтерфейс `IlbAutoEvents` предназначен для поддержки событий автоматизации.

Кроме того, в модуле объявляется сокласс `colbAuto`, он берет на себя обязанности по созданию оболочки, отделяющей интерфейсы объекта от их реализации. Названный класс обслуживается двумя функциями класса, позволяющими создавать объект локально `Create()` либо на удаленной станции `CreateRemote()`.

Теперь откройте модуль `lbAuto_Unit` и внесите в него несколько доработок (листинг 43.12).

Листинг 43.12. Доработка реализации интерфейса в модуле `lbAuto_unit`

```
unit lbAuto_unit;
{$WARN SYMBOL_PLATFORM OFF}

interface
uses ComObj, ActiveX, AxCtrls, Classes,automsrv_TLB, StdVcl;

type
  TlbAuto = class(TAutoObject, IConnectionPointContainer, IlbAuto)
  private
    FConnectionPoints: TConnectionPoints;
    FConnectionPoint: TConnectionPoint;
    FEvents: IlbAutoEvents;
    {Изменение 1 — поле регистрации активного объекта}
    fObjRegHandle:integer;
  public
    procedure Initialize; override;
   {Изменение 2 — объявляем деструктор}
    destructor Destroy; override;
  protected
    property ConnectionPoints: TConnectionPoints
          read FConnectionPoints implements IConnectionPointContainer;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
  end;

implementation
uses ComServ;
```

```
procedure TlbAuto.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IlbAutoEvents;
end;

procedure TlbAuto.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  {Изменение 3 — регистрируем активный объект в таблице ROT}
  RegisterActiveObject(Self as IUnknown, CLASS_lbAuto,
                       ACTIVEOBJECT_WEAK, fObjRegHandle);
  if AutoFactory.EventTypeInfo <> nil then
  begin
  {если о событиях следует уведомлять только отдельного клиента
   — ckSingle, если всех — ckMulti}
   FConnectionPoint := FConnectionPoints.CreateConnectionPoint(
      AutoFactory.EventIID, ckMulti, EventConnect);
  end
  else FConnectionPoint := nil;
end;

destructor TlbAuto.Destroy; {Изменение 4 — доработанный деструктор}
begin
  inherited; {Отменяем регистрацию активного объекта автоматизации}
  RevokeActiveObject(fObjRegHandle,nil);
end;
...
```

Модуль `AutoMemo_Unit` — это вотчина класса-оболочки `TlbAuto`, отвечающего за реализацию интерфейсов сервера автоматизации. Как и полагается для объектов автоматизации, класс создан на основе `TAutoObject`. Так как мы намерены проектировать сервер, поддерживающий события, то, кроме интерфейса автоматизации `IlbAuto`, экземпляр класса `TAutoObject` учитывает пожелания исходящего интерфейса контейнера точек подключения `IConnectionPointContainer`.

Для обслуживания событий в секции частных объявлений `private` включены 4 поля:

1. Поле `FConnectionPoints` предназначено для обслуживания контейнера точек соединения, предоставляемого интерфейсом `IConnectionPointConteiner`. Благодаря `IConnectionPointConteiner` клиенты автоматизации смогут опросить все исходящие интерфейсы сервера.
2. Поле `FConnectionPoint` на стороне сервера реализует интерфейс `IConnectionPoint`, тем самым предоставляя клиенту затребованный им интерфейс.
3. Поле `FEvents` отвечает за реализацию объявленного нами интерфейса событий `IlbAutoEvents`.
4. Поле `fObjRegHandle` хранит указатель на активный COM-объект (который мы получим в результате регистрации COM-объекта в глобальной таблице ROT с помощью функции `RegisterActiveObject()`).

Особого интереса заслуживает процедура инициализации класса `Initialize()`. Создав контейнер точек подключения `FConnectionPoints` и экземпляр точки подключения `FConnectionPoint`, мы проявляем заботу о регистрации COM-объекта в системной таблице исполняющихся объектов ROT. Благодаря этому действию (если по какой-то причине будет запущено несколько экземпляров серверов автоматизации) клиентские приложения поймут, что роль активного сервера исполняет приложение, которое было загружено в адресное пространство последним.

В момент прекращения работы сервера вызывается деструктор, в рамках деструктора мы вызываем функцию `RevokeActiveObject()` и отменяем регистрацию COM-объекта в таблице ROT.

Приступаем к проектированию методов, свойств и событий интерфейса сервера автоматизации. Для вызова редактора библиотеки типов выбираем пункт меню **View | Type Library**. Выберите узел интерфейса `IlbAuto` и (нажимая кнопку **New Method**) создайте три метода (рис. 43.5).

- Метод добавления новой текстовой строки `AddLine()`. Этот метод будет обладать одним параметром `Line:OleVariant`, благодаря которому клиентское приложение сможет заносить в список строку.
- Метод удаления текстовой строки по ее индексу `DeleteLine()`. В данном случае нам также достаточно одного параметра `Index:long`, который предназначен для определения индекса удаляемой строки.
- Метод сортировки текста в списке `SortLines()`. Для этого метода параметры не нужны.

Кроме того, сервер автоматизации нуждается в свойстве, уведомляющем клиента о количестве элементов в списке `ListBox1`. С этой целью на панели управления редактором библиотеки типов нажмите кнопку **New Property | Read Only**. Появившееся свойство переименуйте в `Count` и укажите тип свойства **Long** и **out**.

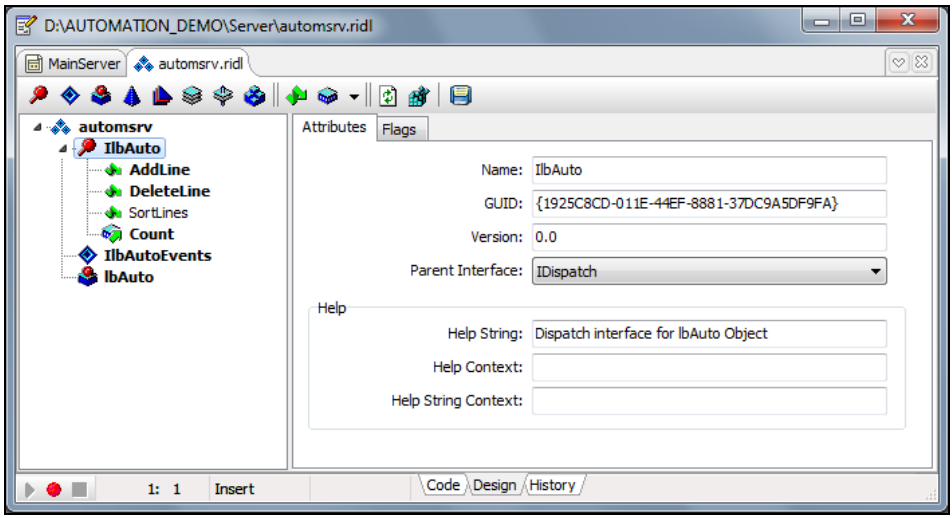


Рис. 43.5. Определение методов и свойств сервера автоматизации

Нажмите кнопку **Refresh Implementation**, это действие приведет к обновлению библиотеки типов. Открыв файл automsrv_TLB.pas, вы увидите, что в объявление интерфейса `IlbAuto` и

диспинтерфейса IlbAutoDisp включились процедуры добавления, удаления и сортировки строк (листинг 43.13). Кроме того, оба интерфейса приобрели доступное только для чтения свойство Count. Для управления свойством Count интерфейс IlbAuto вооружен функцией Get_Count().

Листинг 43.13. Изменения в составе интерфейсов IlbAuto и IlbAutoDisp

```
type IlbAuto = interface(IDispatch)
    ['{1925C8CD-011E-44EF-8881-37DC9A5DF9FA}']
    procedure AddLine(Line: OleVariant); safecall;
    procedure DeleteLine(Index: Integer); safecall;
    procedure SortLines; safecall;
    function Get_Count: Integer; safecall;
    property Count: Integer read Get_Count;
  end;
  IlbAutoDisp = dispinterface
    ['{1925C8CD-011E-44EF-8881-37DC9A5DF9FA}']
    procedure AddLine(Line: OleVariant); dispid 201;
    procedure DeleteLine(Index: Integer); dispid 202;
    procedure SortLines; dispid 203;
    property Count: Integer readonly dispid 204;
  end;
```

Обратите внимание на то, что все методы и свойства в диспинтерфейсе IlbAutoDisp снабжены уникальными идентификаторами dispid, именно благодаря ним обеспечивается вызов методов клиентом автоматизации.

Перейдем к модулю lbAuto_Unit, в котором объявлен класс TlbAuto, отвечающий за реализацию интерфейсов. Обратившись к секции защищенных объявлений класса, мы вновь встретим наши процедуры и функции (листинг 43.14).

Листинг 43.14. Секция protected модуля lbAuto_Unit

```
protected
    procedure AddLine(Line: OleVariant); safecall;
    procedure DeleteLine(Index: Integer); safecall;
    procedure SortLines; safecall;
    function  Get_Count: Integer; safecall;
```

Как видите, Delphi изрядно потрудилась и создала все необходимые для нас заготовки кода.

Предлагаю вновь перейти к редактору библиотеки типов, но на этот раз с целью создать обработчики событий. Для этого выберите узел интерфейса событий IlbAutoEvents и трижды щелкните по кнопке **New Method** (рис. 43.6).

- Первое событие OnChange() специализируется на извещении клиентского приложения о том, что содержимое списка ListBox1 претерпело изменения. Это событие может произойти в случае добавления новой строки в список или при удалении строки. Для того чтобы клиент автоматизации смог узнать, сколько строк осталось в списке, мы снабдим

метод выходным параметром `Count` и типизируем его как указатель на целое число `long*`.

- Событие `OnBeforeSort()` вызывается перед началом сортировки строк в списке `ListBox1`. Параметры отсутствуют.
- Событие `OnAfterSort()` уведомляет клиента автоматизации о завершении сортировки. Без параметров.

Для того чтобы все внесенные нами изменения в интерфейсы сервера автоматизации отразились в соответствующих разделах модулей проекта, вновь воспользуйтесь кнопкой **Refresh Implementation**.

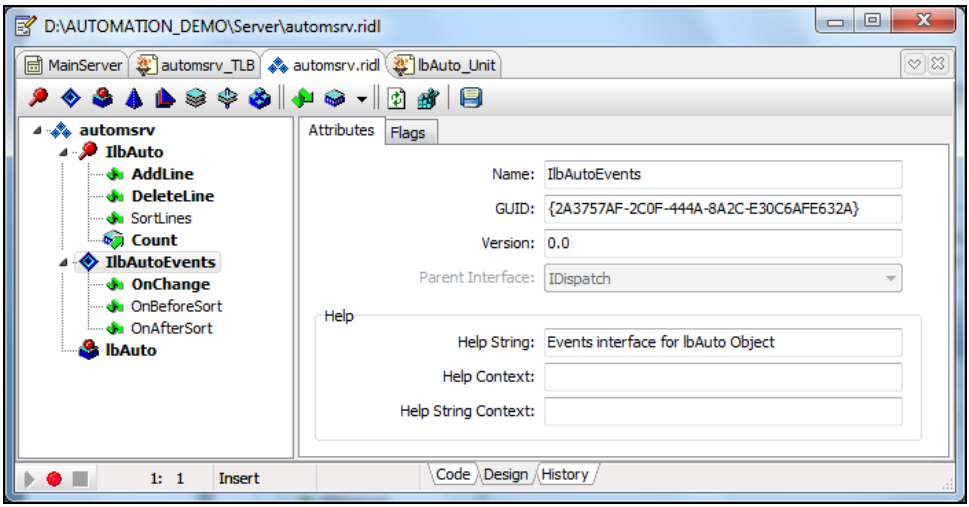


Рис. 43.6. Определение событий сервера автоматизации в редакторе библиотеки типов

В первую очередь проверим, каким изменениям подвергся модуль `automsrv_TLB`. В секции объявления диспинтерфейса событий `IlbAutoEvents` появились заголовки всех трех событий (листинг 43.15).

Листинг 43.15. Заголовки методов диспинтерфейса событий `IlbAutoEvents`

```
type IlbAutoEvents = dispinterface
    ['{2A3757AF-2C0F-444A-8A2C-E30C6AFE632A}']
    procedure OnChange(out Count: Integer); dispid 201;
    procedure OnBeforeSort; dispid 202;
    procedure OnAfterSort; dispid 203;
  end;
```

Каждое из событий снабжено идентификатором `dispid ###`. Эти значения нам очень пригодятся при проектировании клиентского проекта.

Перейдем в модуль lbAuto_Unit.pas и заполним программной логикой методы управления списком (листинг 45.16). После добавления в список `ListBox1` новой строки мы генерируем событие `OnChange()`.

Листинг 43.16. Реализация методов управления списком

```
procedure TlbAuto.AddLine(Line: OleVariant); {добавляем строку}
var X:Integer;
begin
  frmMainServer.ListBox1.Items.Add(Line);   {добавляем строку}
  X:=frmMainServer.ListBox1.Count;          {узнаем число строк в списке}
  if FEvents<>nil then FEvents.OnChange(X); {генерируем OnChange}
end;

procedure TlbAuto.DeleteLine(Index: Integer); {удаляем строку}
var X:Integer;
begin
  frmMainServer.ListBox1.Items.Delete(Index); {удаляем строку}
  X:=frmMainServer.ListBox1.Count;
  if FEvents<>nil then FEvents.OnChange(X);   {генерируем OnChange}
end;

procedure TlbAuto.SortLines; {сортируем строки}
begin
  if FEvents<>nil then FEvents.OnBeforeSort;{генерируем OnBeforeSort}
  frmMainServer.ListBox1.Sorted:=true;      {сортируем строки}
  if FEvents<>nil then FEvents.OnAfterSort; {генерируем OnAfterSort}
end;

function TlbAuto.Get_Count: Integer;{число строк в списке}
begin
  Result:=frmMainServer.ListBox1.Count;
end;
```

Проект сервера автоматизации в целом закончен. Наш проект способен не просто предоставлять услуги автоматизации, но и извещать клиентское приложение о ряде событий.

Клиент автоматизации

Если вы планируете разработать клиентское приложение, поддерживающее события автоматизации, то придется хорошо потрудиться. Сложность подобных проектов обуславливается необходимостью реализации стока событий, по которому сервер сможет уведомлять клиента о своих действиях.

Создайте новый проект VCL. Переименуйте главную форму проекта в `frmMainClient`, а соответствующий ей модуль с кодом в `MainClient.pas`. Для того чтобы клиент получил представление о методах и свойствах сервера, к проекту следует сразу подключить библиотеку типов `automsrv_TLB` и сослаться на нее в строке `Uses`. Сохраните проект в каталоге *x*:\AUTOMATION_DEMO\Client под любым именем.

В секции частных объявлений главной формы проекта определяем поле `fServer` со ссылкой на интерфейс сервера `IlbAuto` и поле `fCookie`, которое позднее пригодится нам при подключении к стоку событий (листинг 43.17).

Листинг 43.17. Объявление полей в модуле `MainClient` главной формы

```
type
  TfrmMainClient = class(TForm)
...
  private
    fServer:IlbAuto;
    fCookie:Longint;
    Procedure OnChange(Count:Integer); {реакция на событие OnChange}
    Procedure OnBeforeSort; {реакция на событие OnBeforeSort}
    Procedure OnAfterSort; {реакция на событие OnAfterSort}
  end;
```

Кроме того, в секции частных объявлений следует описать заголовки трех процедур, которые будет вызывать клиентское приложение в момент возникновения того или иного события на сервере автоматизации (см. листинг 43.17). Мы пока не станем заполнять объявленные процедуры кодом, вернемся к этому позднее.

Настал черед наиболее ответственного этапа — этапа разработки класса стока событий `TEventSink`. К стоку будет обращаться сервер автоматизации, чтобы уведомить клиента автоматизации о возникающих событиях. Для того чтобы класс смог обслуживать пожелания клиента автоматизации, нам следует обеспечить поддержку интерфейсов `IUnknown` и `IDispatch` (листинг 43.18).

Листинг 43.18. Описание класса стока событий

```
type TEventSink=class(TObject, IUnknown, IDispatch)
   private
    fController : TfrmMainClient; {ссылка на форму клиента автоматизации}

    {методы интерфейса IUnknown}
    function AddRef: Integer; stdcall;
    function Release: Integer; stdcall;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;

    {методы интерфейса IDispatch}
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
                         out TypeInfo): HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
                           NameCount, LocaleID: Integer;
                           DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
                    Flags: Word; var Params; VarResult, ExcepInfo,
                    ArgErr: Pointer): HResult; stdcall;

   public
     Constructor Create(Controller : TfrmMainClient);
 end;
```

К счастью, нам потребуется "поколдовать" не над всеми семью методами интерфейсов IUnknown и IDispatch, для нашей задумки достаточно реализовать только два из них. В первую очередь обратимся к методу QueryInterface() интерфейса IUnknown. Благодаря QueryInterface() мы сможем запрашивать COM-объект на предмет существования у него интерфейса с идентификатором IID, в случае успеха ссылка на интерфейс окажется в выходном параметре Obj (листинг 43.19).

Листинг 43.19. Реализация метода QueryInterface()

```
function TEventSink.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  if GetInterface(IID,Obj) then Result:=S_OK
  else
    if IsEqualIID (IID,IlbAutoEvents) then
       Result:=QueryInterface(IDispatch, Obj)
    else Result:=E_NOINTERFACE;
end;
```

В первых строках кода вызывается метод, объявленный еще у прадедушки всех классов VCL TObject:

```
function GetInterface(const IID: TGUID; out Obj): Boolean;
```

Функция возвращает в выходной параметр Obj интерфейс, обозначенный идентификатором IID. Если попытка решить проблему в лоб не возымела успеха, то мы запрашиваем интерфейс IDispatch, рекурсивно вызывая этот же метод. Таким образом, метод настроен только на возвращение интерфейсов IUnknown, IDispatch и нашего интерфейса обработки событий IlbAutoEvents. Во всех остальных случаях в результат будет помещено значение E_NOINTERFACE, говорящее об отсутствии подходящего интерфейса.

Еще один метод, который нам необходимо реализовать, принадлежит интерфейсу IDispatch. Благодаря методу Invoke() клиент автоматизации начинает реагировать на события сервера (листинг 43.20).

Листинг 43.20. Реализация метода Invoke()

```
function TEventSink.Invoke(DispID: Integer; const IID: TGUID;
                           LocaleID: Integer;  Flags: Word; var Params;
                         VarResult, ExcepInfo, ArgErr: Pointer): HResult;
var param:OleVariant;
begin
  Result:=S_OK;
  case DispId of
   201 : begin
         param:=OleVariant(TDispParams(Params).rgvarg^[0]);
         fController.OnChange(param);//procedure OnChange; dispid 201;
         end;
   202 : fController.OnBeforeSort; //procedure OnBeforeSort; dispid 202;
   203 : fController.OnAfterSort;  //procedure OnAfterSort; dispid 203;
  end;
end;
```

Сердце процедуры — селектор `case`, обслуживающий числовой идентификатор `DispId`. Благодаря тому, что к клиенту подключен файл библиотеки типов, ему известны значения этих идентификаторов. Так значение 201 соответствует событию `OnChange()`, 202 — `OnBeforeSort()` и 203 — `OnAfterSort()`. Так как событие `OnChange()` обладает аргументом, возвращающим число строк в списке, мы считываем его значение из параметра `Params`.

Работа со стоком событий `TEventSink` завершена, теперь в модуле главной формы следует объявить глобальную переменную соответствующего типа (листинг 43.21).

Листинг 43.21. Объявление глобальной переменной стока

```
var frmMainClient: TfrmMainClient;
    fEventSink:    TEventSink;
```

Займемся подключением стока к проекту клиента автоматизации. Для этого потребуется помощь события `OnCreate()` главной формы проекта (листинг 43.22).

Листинг 43.22. Подключение стока к клиенту автоматизации

```
procedure TfrmMainClient.FormCreate(Sender: TObject);
begin
  fServer:=ColbAuto.Create;
  fEventSink:=TEventSink.Create(frmMainClient);
  InterfaceConnect(fServer, IlbAutoEvents, fEventSink, fCookie);
end;
```

В коде мы создаем экземпляр `fServer` и `fEventSink` и соединяем сток событий с исходящим интерфейсом сервера.

Чтобы после завершения работы со стоком событий клиент не ушел по-английски (не попрощавшись), отключаемся от стока в момент закрытия главной формы проекта (листинг 43.23).

Листинг 43.23. Отключение от стока при завершении работы

```
procedure TfrmMainClient.FormClose(Sender: TObject;
                                   var Action: TCloseAction);
begin
  InterfaceDisConnect(fEventSink,IlbAutoEvents,fCookie);
end;
```

Наступает заключительная фаза работы над клиентом автоматизации. Расположим на форме следующие компоненты.

- Четыре кнопки `TButton`. Первая кнопка `btnAddLine` отвечает за вызов метода сервера автоматизации `AddLine()`, добавляющего новую строку в список. Кнопка `btnDeleteLine` предназначена для удаления строки из списка. Кнопка `btnSort` отправляет команду на сортировку строк. Кнопка `btnCount` опрашивает единственное свойство сервера, хранящее число строк в списке.
- Две строки ввода `TEdit`. Компонент `Edit1` предназначен для ввода новой строки. Компонент `Edit2` содержит индекс удаляемой строки.

- Метка TLabel с именем lbCount станет отображать число доступных строк.
- Многострочный редактор TMemo с названием по умолчанию Memo1. Этот компонент станет специализироваться на ведении протокола событий.

Листинг 43.24 содержит описание четырех обработчиков событий, вызываемых по щелчку по кнопкам btnAddLine, btnDeleteLine, btnSort и btnCount.

Листинг 43.24. Запросы услуг на сервере

```
procedure TfrmMainClient.btnAddLineClick(Sender: TObject);
begin {отправка серверу новой текстовой строки}
  if TRIM(Edit1.Text)<>'' then fServer.AddLine(Edit1.Text);
  Edit1.Clear;
end;

procedure TfrmMainClient.btnDeleteLineClick(Sender: TObject);
var Index:Integer; {удаление строки из списка}
begin
  Index:=StrToIntDef(Edit2.Text,-1);
  if (Index>-1) and (Index<fServer.Count) then fServer.DeleteLine(Index)
  else Application.MessageBox('Ошибка удаления!','Ошибка',MB_OK);
end;

procedure TfrmMainClient.btnSortClick(Sender: TObject);
begin {вызов процедуры сортировки строк}
  fServer.SortLines;
end;

procedure TfrmMainClient.btnCountClick(Sender: TObject);
var X:Integer; {узнаем число строк}
begin
  X:=fServer.Count;
  lbCount.Caption:=IntToStr(X);
end;
```

У нас остались незаполненными процедуры с реакцией клиента автоматизации на события сервера. Например, при поступлении извещения о том, что на сервере произошло событие OnChange(), клиент автоматизации обновит свои данные о количестве строк в списке сервера (листинг 43.25).

Листинг 43.25. Ответная реакция клиента на изменение числа строк на сервере

```
procedure TfrmMainClient.OnChange(Count :Integer);
begin
  frmMainClient.lbCount.Caption:=IntToStr(Count);
end;
```

Перед началом сортировки списка на сервере имеет смысл деактивировать все кнопки клиента, чтобы он не смог помешать процессу упорядочивания строк (листинг 43.26).

Листинг 43.26. Ответная реакция на начало и завершение сортировки на сервере

```
procedure TfrmMainClient.OnBeforeSort;
var i:integer; {перед сортировкой отключаем все кнопки TButton}
begin
  for i := 0 to frmMainClient.ControlCount - 1 do
    if (frmMainClient.Controls[i] is TButton) then
      with (frmMainClient.Controls[i] as TButton) do Enabled:=False;
end;
//по завершению сортировки - активируем все экземпляры кнопок
procedure TfrmMainClient.OnAfterSort;
var i:integer;
begin
  {После сортировки включаем все кнопки TButton}
  for i:= 0 to frmMainClient.ControlCount - 1 do
    if (frmMainClient.Controls[i] is TButton) then
        with (frmMainClient.Controls[i] as TButton) do Enabled:=True;
end;
```

И еще один штрих. Немного доработаем метод `Invoke()`. Для того чтобы компонент `Memo1` смог протоколировать поступающие в адрес клиента уведомления о событиях, добавим в него единственную строку кода (листинг 43.27).

Листинг 43.27. Доработка метода `Invoke()`

```
function TEventSink.Invoke(DispID: Integer; const IID: TGUID;
                           LocaleID: Integer;  Flags: Word; var Params;
                         VarResult, ExcepInfo, ArgErr: Pointer): HResult;
var param:OleVariant;
begin
  Result:=S_OK;
{********************** Новая строка кода ******************}
  fController.Memo1.Lines.Add(Format('DispID=%d',[DispId]));
{************************************************************}
  case DispId of
   201: ...
```

Проект завершен. В нем, помимо выполнения стандартных задач по обслуживанию команд контроллера автоматизации, сервер приобрел уникальную возможность — уведомлять клиента о происходящих событиях.

ГЛАВА 44

Интерфейс *IShellFolder*

Одной из важных составных частей прикладного программного интерфейса Windows выступает оболочка Shell операционной системы Windows. Все объекты оболочки Windows построены в строгом соответствии с требованиями COM, а доступ к ним осуществляется посредством интерфейсов COM. Благодаря такой реализации мы получаем возможность не только манипулировать этими объектами, но и расширять возможности Shell Windows. В пакете API, обслуживающем Shell, сосредоточена коллекция методов и несколько десятков объектов COM и их интерфейсов.

Если вам не терпится увидеть Shell в действии, то просто откройте стандартный Проводник (explorer.exe). Проводник, как программное воплощение оболочки, интегрирует ряд физических и виртуальных объектов операционной системы, входящих в *пространство имен оболочки* (shell namespace), в единую иерархическую структуру. В перечень физических объектов shell namespace входит все, что можно "потрогать руками": файлы, каталоги, дисковые накопители. С этими объектами мы хорошо знакомы и много о них говорили на протяжении всей книги. Но наряду с ними в оболочке представлены виртуальные папки и объекты: "Мой компьютер", "Принтеры", "Панель управления", "Корзина". Также оболочка способна представить объекты, физически принадлежащие другим компьютерам: сетевые ресурсы, общие принтеры. Это тоже элементы пространства имен оболочки, с которыми мы очень часто сталкивались как пользователи Windows, но пока не тревожили их покой в роли программистов.

> ***Замечание***
>
> В арсенале оболочки операционной системы Windows Shell насчитывается более сотни COM-объектов и их интерфейсов. Среди них: интерфейс активного рабочего стола `IActiveDesktop`, интерфейсы контекстного и всплывающего меню (`IContextMenu`, `IMenuPopup`), интерфейс инсталляции и деинсталляции приложений `IShellApp`, интерфейс ярлыка `IShellLink` и даже интерфейс записи данных на CD- и DVD-диски `ICDBurn`.

В рамках этой главы мы познакомимся с одним из базовых интерфейсов оболочки — интерфейсом `IShellFolder`. Наш новый знакомый специализируется на обслуживании папок Windows.

> ***Внимание!***
>
> Для доступа к интерфейсу `IShellFolder` к проекту Delphi требуется подключить модули `ShellApi` и `ShlObj`.

Идентификация объекта Shell

Каждый объект пространства имен оболочки имеет два описания. С одной стороны это привычное представление в виде тестового пути и имени файла. С другой стороны объект характеризуется уникальным идентификатором — PIDL. Хотя первый способ является классическим, но он не выступает в роли опорного для Shell. Основу идентификации объекта пространства имен составляет *указатель на список идентификаторов* (pointer to an item identifier list, PIDL).

В языке Delphi список идентификаторов представлен в модуле `ShlObj` в формате структуры `TItemIDList` (листинг 44.1).

Листинг 44.1. Структура `TItemIDList`

```
type  PItemIDList = ^TItemIDList;
  _ITEMIDLIST = record
     mkid: TSHItemID;
   end;
```

Структура `TItemIDList` содержит поле `TSHItemID` (листинг 44.2), это и есть идентификатор объекта Windows Shell.

Листинг 44.2. Структура `TSHItemID`

```
type PSHItemID = ^TSHItemID;
  _SHITEMID = record
    cb: Word;                   //размер идентификатора
    abID: array[0..0] of Byte; //собственно идентификатор
  end;
```

Массив байтов `abID` распределяется автоматически, содержимое массива уникально по отношению ко всем элементам папки, которой и принадлежит описываемый этой записью объект. Непосредственно с полями структуры PIDL нам работать вряд ли придется, т. к. все методы оболочки предпочитают получать эту последовательность, называемую Item ID Lists, целиком. Список идентификаторов может быть как абсолютным (т. е. определяющим положение объекта относительно корневой папки), так и относительным — определяющим положение объекта относительно какой-либо конкретной папки.

Диалоговое окно получения PIDL

Применение PIDL особо актуально в том случае, если объект не является частью файловой системы и соответственно не может похвастаться наличием файлового имени. Для получения PIDL объекта можно воспользоваться диалоговым окном, вызываемым функцией

```
function SHBrowseForFolder(var lpbi: TBrowseInfo): PItemIDList;
```

Здесь `lpbi` представляет собой ссылку на структуру `TBrowseInfo` (листинг 44.3).

Перечень допустимых флагов `ulFlags` представлен в табл. 44.1.

Листинг 44.3. Структура `TBrowseInfo`

```
type TBrowseInfo = record
  hwndOwner : HWND;          {дескриптор родительского окна}
  pidlRoot : PItemIDList;    {PIDL корневого элемента}
  pszDisplayName : PChar;    {адрес буфера, в который будет помещаться
                              название выбранной пользователем папки}
  lpszTitle : PChar;         {адрес буфера, содержащего заголовок диалога}
  ulFlags : UINT;            {флаг опций диалогового окна (табл. 44.1)}
  lpfn : Pointer;      {адрес функции обратного вызова приложения,
                        вызываемого при показе диалога, допускается nil}
  lParam : Word;        {значение для передачи в функцию обратного вызова}
  ImageIndex : INTEGER;{индекс значка выбранной в диалоге папки}
end;
```

Таблица 44.1. *Возможные значения параметра* `ulFlags` *опций диалогового окна*

Флаг	Описание
BIF_BROWSEFORCOMPUTER	Допускается только выбор компьютеров, в противном случае кнопка **OK** будет недоступна
BIF_BROWSEFORPRINTER	Допускается только выбор принтеров
BIF_BROWSEINCLUDEFILES	Помимо папок в диалоге отображаются и файлы
BIF_BROWSEINCLUDEURLS	Работает при установленных флагах BIF_USENEWUI и BIF_BROWSEINCLUDEFILES, диалог способен отображать адреса URL
BIF_DONTGOBELOWDOMAIN	Не включает сетевые папки за пределами текущего домена
BIF_EDITBOX	Интегрирует в диалоговое окно строку ввода, в которой пользователь может ввести имя элемента
BIF_NEWDIALOGSTYLE	Выводит диалоговое окно в современном интерфейсе, допускает изменение размеров окна, поддерживает операции удаления, создания новой папки и т. п.
BIF_RETURNFSANCESTORS	Возвращает только элементы файловой системы
BIF_RETURNONLYFSDIRS	Возвращает только файловые каталоги, если пользователь выбирает виртуальную папку, то кнопка **OK** становится недоступной
BIF_SHAREABLE	Работает совместно с флагом BIF_USENEWUI, диалог отображает выделенные в общее пользование ресурсы на удаленных системах
BIF_STATUSTEXT	Отображает строку статуса, в нее может быть помещена информация функцией обратного вызова приложения, которому принадлежит диалоговое окно
BIF_USENEWUI	Выводит диалог с современным интерфейсом, эквивалент флагов BIF_EDITBOX и BIF_NEWDIALOGSTYLE
BIF_VALIDATE	Работает совместно с флагом BIF_EDITBOX, предназначен для проверки корректности ввода имени элемента в строку ввода

Задачу преобразования полученного идентификатора PIDL в путь и имя файла решает другая функция:

```
function SHGetPathFromIDList(pidl : PItemIDList;
                             var pszPath : PChar): boolean;
```

В листинге 44.4 предложен вариант использования только что изученных функций. Разработанная нами функция SelectPIDL() вызывает диалоговое окно, содержащее элементы файловой структуры компьютера. Указатель на список идентификаторов PIDL выбранной пользователем папки возвращается в качестве результата выполнения метода, путь передается в параметр-переменную Path.

Листинг 44.4. Вызов диалога доступа к файловой структуре

```
function SelectPIDL(var Path : string): PItemIDList;
var PIDL:PItemIDList;
    BrowseInfo:TBrowseInfo;
    Buf : Array[0..MAX_PATH-1] of Char;
begin
  with BrowseInfo do
  begin
    hwndOwner     :=Application.MainForm.Handle;
    pidlRoot      :=nil;
    pszDisplayName:=Buf;
    lpszTitle     :='Выбор папки';
    ulFlags       :=BIF_RETURNFSANCESTORS OR BIF_USENEWUI;
    lpfn          :=nil;
  end;
  PIDL:=SHBrowseForFolder(BrowseInfo);
  SHGetPathFromIDList(PIDL, Buf);
  Path:=StrPas(Buf);
  Result:=PIDL;
end;
```

В результате обращения к функции из листинга 44.4 на экране отобразится диалог выбора папки.

Получение пути к системным папкам

Для разработчика большой интерес представляет функция

```
function SHGetSpecialFolderPath(hwndOwner : HWND; lpszPath : PChar;
          nFolder : Integer; fCreate : boolean) : boolean;
```

Функция предоставит нам информацию о файловом пути к специальным папкам Windows. Эта функция (в отличие от рассмотренной ранее SHBrowseForFolder()) обходится без диалогового окна.

Здесь: hwndOwner — дескриптор окна, которое будет считаться владельцем всех показываемых сообщений и диалоговых окон; lpszPath — указатель на буфер, в который функция будет возвращать путь к папке; nFolder — числовой идентификатор папки (большинство ключевых констант приведено в *приложении 8*); fCreate — определяет поведение метода при отсутствии папки, если мы передаем значение true, то папка (если это возможно) будет создана. При успешном выполнении метод возвратит true.

В качестве примера использования SHGetSpecialFolderPath() изучите код из листинга 44.5. В листинге сведения о путях ко всем доступным системным папкам заносятся в компонент ListBox1:TListBox.

Листинг 44.5. Построение списка системных путей

```
var Path   : array[0..MAX_PATH-1] of Char;
    I      : word;
    Result : boolean;
    S      : string;
begin
  for I:=$0000 to $003d do
  begin
    Result:=SHGetSpecialFolderPath(Application.MainForm.Handle,
                                   Path, i, false);
    if Result then begin
       s:=Format('%.4x %s',[I,StrPas(Path)]);
       ListBox1.AddItem(s, nil);
    end;
  end;
end;
```

Интерфейс *IShellFolder*

Вне зависимости от того, является ли папка физическим каталогом на жестком диске или представляет собой виртуальный элемент, она описывается в рамках единого интерфейса, именуемого `IShellFolder`.

Изучение интерфейса начнем с ответа на вопрос: каким образом можно получить указатель на интерфейс из программного кода?

Открыв Проводник, мы увидим, что корневой папкой (порождающей все дерево папок) пространства имен назначен рабочий стол. Поэтому в Windows реализована функция, специализирующаяся на получении интерфейса именно рабочего стола:

```
function SHGetDesktopFolder(var pSF: IShellFolder) : HRESULT;
```

Полученный указатель на интерфейс передается в единственный параметр метода — `pSF`. Результат выполнения метода сигнализирует о том, успешно или не совсем (значение меньше нуля) завершился его вызов.

Для обработки результатов типа `HRESULT`, возвращаемых методами объектов COM, в модуле `ComObj` предусмотрена специальная процедура:

```
procedure OleCheck(ECode: HResult);
```

Задача процедуры `OleCheck()` — генерация исключительной ситуации `EOleSysError`, если метод завершился некорректно (листинг 44.6).

Листинг 44.6. Получение интерфейса `IShellFolder` рабочего стола

```
function DesktopShellFolder: IShellFolder;
begin
  OleCheck(SHGetDesktopFolder(Result));
end;
```

Научившись получать доступ к интерфейсу рабочего стола, приступим к рассмотрению методов интерфейса.

Получение PIDL из файлового пути

Вместо понятного человеку представленного обычной текстовой строкой пути к файлу, привередливая оболочка Windows предпочитает работать с идентификатором PIDL. Роль переводчика с человеческого языка на язык, понятный Windows Shell, играет метод

```
function ParseDisplayName(hwndOwner: HWND; pbcReserved: Pointer;
              lpszDisplayName: pOLESTR; out pchEaten: ULONG;
              out ppidl: PItemIDList; var dwAttributes: ULONG): HResult;
```

Первый параметр метода `hwndOwner` представляет собой дескриптор родительского окна, которому принадлежит вызов этого метода, в дескриптор допускается помещать нулевое значение. Параметр `pbcReserved` — резервный и поэтому достоин неопределенного указателя `nil`. В основной параметр `lpszDisplayName` мы передаем указатель на строку с полным путем к каталогу или файлу (листинг 44.7). Необязательный параметр `pchEaten` возвратит в переменную число проанализированных методом символов. Результаты перевода файлового пути в так любимый Windows Shell идентификатор мы обнаружим в параметре `ppidl`. Последний параметр представляет собой набор флагов, определяющих атрибуты файловых объектов. Параметр работает как на вход (конкретизируя, какие именно объекты нас интересуют), так и на выход, информируя о характеристиках объектов. Атрибутов почти три десятка, поэтому мы не сможем привести их на страницах книги, в простейшем случае в параметр допустимо передавать нулевое значение.

Получение интерфейса дочерней папки

Мы уже умеем получать интерфейс `IShellFolder` самой главной папки Windows — рабочего стола. Теперь стоит разобраться, как можно получить интерфейс дочерней папки (принадлежащей тому объекту, чьим интерфейсом мы уже владеем). Для этих целей в состав методов интерфейса включена функция

```
function BindToObject(PIDL : PItemIDList; pbc : IBindCtx;
                      const riid : TGUID; Out ppvOut : Untyped) : HRESULT;
```

Обсудим параметры метода. Первый параметр `PIDL` — идентификатор папки, интерфейс которой нам так необходим. Во втором параметре `pbc` передается адрес интерфейса `IBindCtx`, определяющий связанные с папкой контекстные данные. Однако не все папки поддерживают эти данные, поэтому параметр не обязателен. Вместо адреса интерфейса допускается передача пустышки `nil`. Третий аргумент `riid` определяет, какого рода интерфейс мы рассчитываем получить. Для всех наших примеров следует пользоваться константой `IID_IShellFolder`, но надо знать, что это не единственный вариант развития событий. В модуле `ShlObj` объявлено несколько констант интерфейсов (`IID_IShellFolder`, `IID_IShellFolder2`, `IID_IStream` и т. д.). Результат выполнения метода в виде указателя на интерфейс передается в последний параметр-переменную `ppvOut`.

Листинг 44.7 демонстрирует совместную работу методов `ParseDisplayName()` и `BindToObject()`. Задача функции `FilePathToInterface()` — преобразовать символьный файловый путь `Path` в `PIDL` объекта и получить интерфейс `IShellFolder` этого объекта.

Листинг 44.7. Получение интерфейса `IShellFolder` из файлового пути

```
function FilePathToInterface(const Path : string; var PIDL: PItemIDList;
                             var ShellFolder:IShellFolder):boolean;
var  Desktop:IShellFolder;
     Eaten,Attributes: ULONG;
```

```
begin
try
  OleCheck(SHGetDesktopFolder(Desktop)); {рабочий стол}
  OleCheck(Desktop.ParseDisplayName      {получаем PIDL}
                  (0, nil, pChar(Path), Eaten, pidl, Attributes));
  OleCheck(Desktop.BindToObject          {получаем PIDL}
                  (pidl,nil,IID_IShellFolder, Pointer(ShellFolder)));
  Result:=True; //функция выполнена корректно
except
  Result:=false; //в работе функции был сбой
end;
end;
```

Получение названия объекта по PIDL

Каким бы не был хорошим способ идентификации файлового объекта с помощью идентификатора PIDL, нам с вами привычнее работать с текстовым названием. Поэтому в перечне методов интерфейса `IShellFolder` предусмотрена функция

```
function GetDisplayNameOf(pidl: PItemIDList;
                          uFlags: DWORD; var lpName: TStrRet): HResult;
```

Здесь: `pidl` — идентификатор интересующего нас объекта; параметром `uFlags` определяется полнота возвращаемой информации, например, для получения полного пути подойдет сочетание флагов `SHGDN_NORMAL` и `HGDN_FORPARSING`, а флаг `SHGDN_INFOLDER` возвратит имя объекта относительно родительской папки. Результаты выполнения метода окажутся в параметре `lpName`, представляющем собой запись типа `TStrRet` (листинг 44.8).

Листинг 44.8. Состав полей записи `TStrRet`

```
type PSTRRet = ^TStrRet;
    _STRRET = record
     uType: UINT;          //значение, определяющее тип строки
   case Integer of
     0: (pOleStr: LPWSTR);//указатель на строку OLE
     1: (pStr: LPSTR);    //указатель на строку, заканчивающуюся 0
     2: (uOffset: UINT);  //размер отступа в байтах в поле TSHItemID,
                          //где хранится название объекта в формате ANSI
     3: (cStr: array[0..MAX_PATH-1] of Char); //буфер с данными
end;
```

Разработаем собственную функцию `GetDisplayName()`, позволяющую получить текстовое описание файла или папки. На вход функции поступают: указатель на интерфейс `IShellFolder`, идентификатор `PIDL` интересующего нас объекта и флаги `Flags`, определяющие формат возвращаемого текста (листинг 44.9).

Листинг 44.9. Получение названия объекта

```
function GetDisplayName(Parentfolder: IShellFolder; PIDL: PItemIDList;
                        Flags: DWORD): string;
var  StrRet: TStrRet;
     P: PChar;
```

```
begin
  Result := '';
  FillChar(StrRet, SizeOf(StrRet), 0);
  ParentFolder.GetDisplayNameOf(PIDL, Flags, StrRet);
 case StrRet.uType of
  STRRET_CSTR: SetString(Result, StrRet.cStr, lStrLenA(StrRet.cStr));
  STRRET_OFFSET: begin
           P := @PIDL.mkid.abID[StrRet.uOffset - SizeOf(PIDL.mkid.cb)];
           SetString(Result, P, PIDL.mkid.cb - StrRet.uOffset);
                end;
  STRRET_WSTR: if Assigned(StrRet.pOleStr) then Result := StrRet.pOleStr
               else Result := '';
 end;
//если в Result мы получили не текстовое описание, а GUID объекта
//(он содержит префикс '::{'), надо сделать еще одно преобразование,
//но на этот раз с флагом SHGDN_NORMAL
 if (Pos('::{', Result) = 1) then
    Result := GetDisplayName(ParentFolder, PIDL, SHGDN_NORMAL);
end;
```

Изменение названия объекта

Чтобы изменить имя файлового объекта либо дочерней папки, обратитесь к методу

```
function SetNameOf(hwndOwner: HWND; pidl: PItemIDList; lpszName: POLEStr;
                   uFlags: DWORD; var ppidlOut: PItemIDList): HResult;
```

Здесь: `hwndOwner` — дескриптор окна, из которого был произведен вызов метода; `pidl` — идентификатор обслуживаемого файлового объекта; `lpszName` — указатель на строку, содержащую новое название; `uFlags` — флаги, например флаг `SHGDN_NORMAL` указывает, что идет речь о полном имени, `SHGDN_INFOLDER` — относительное имя. Результат работы метода возвращается в последнем параметре в виде структуры `ppidlOut`.

Сбор дочерних объектов папки, интерфейс *IEnumIDList*

Зачастую программе требуется не просто обратиться к конкретному объекту файловой системы, а получить список дочерних каталогов и файлов, принадлежащих родительской папке. В этом случае лучшим помощником окажется метод

```
function EnumObjects(Wnd : HWND; grfFlags : DWORD;
                     out EnumIDList : IEnumIDList) : HRESULT;
```

Первый параметр метода рассчитывает получить дескриптор окна, из которого был вызван интерфейс. Второй аргумент представляет собой набор флагов, определяющих, какие именно объекты представляют для нас интерес. Упомянем основные из них: `SHCONTF_FOLDERS` — включать элементы папки; `SHCONTF_NONFOLDERS` — включать элементы, не являющиеся папками; `SHCONTF_INCLUDEHIDDEN` — включать скрытые элементы.

Метод возвращает указатель на специализированный интерфейс `IEnumIDList`, позволяющий организовать перебор дочерних объектов. Эта операция осуществляется за счет четырех методов (табл. 44.2).

Таблица 44.2. *Методы интерфейса* `IEnumIDList`

Метод	Описание
`function Reset : HRESULT;`	Метод перемещается к первому элементу дочерних объектов
`function Next(celt : Cardinal;out rgelt : PItemIDList, var CeltFetched : Cardinal) : HRESULT;`	Метод осуществляет последовательный перебор объектов, представленных интерфейсом `IEnumIDList`. Возвратит определенное количество элементов множества и известит о том, в каком месте множества мы находимся. Возвращаемые элементы окажутся в параметре `rgelt`. Это указатель на массив `PItemIDList`. Какими порциями будет осуществляться передача элементов в результирующий массив, определяет самый первый параметр функции — `celt`. Для поэлементного перебора устанавливайте его в 1. Реальное количество считанных элементов выясняется в последнем параметре `CeltFetched`. Пока метод возвращает значение `S_OK`, это служит признаком, что конец перечня еще не достигнут. Значение `S_FALSE` свидетельствует о том, что перебор элементов завершен
`function Skip(celt : cardinal) : HRESULT;`	Метод пропускает `celt` элементов массива
`function Clone(ppenum : IEnumIDList) : HRESULT;`	Клонирует множество и возвращает указатель на клон в единственном параметре функции

Процесс сбора сведений обо всех дочерних элементах относительно папки, описываемой интерфейсом `ParentFolder:IshellFolder`, предложен в листинге 44.10. Помимо указателя на интерфейс демонстрационная функция `GetFolderList()` содержит еще два аргумента. Собранные сведения помещаются в список строк `FolderList`. Параметр `Flags` определяет, какие объекты следует передавать в список, по умолчанию включены все известные нам флаги.

Листинг 44.10. Построение списка дочерних объектов для папки

```
function GetFolderList(ParentFolder:IShellFolder;
                    var FolderList : TStringList;
                    Flags:DWORD=SHCONTF_FOLDERS OR
                    SHCONTF_INCLUDEHIDDEN OR SHCONTF_NONFOLDERS):boolean;
var EnumIDList: IEnumIDList;
    pidl: PItemIDList;
    CeltFetched : Cardinal;
    ParsingPath :string;
begin
  Result:=False;
  if (ParentFolder.EnumObjects(0, Flags, EnumIDList)=S_OK) then
  begin
    while EnumIDList.Next(1, pidl, CeltFetched)=S_OK do
    begin
      ParsingPath:=GetDisplayName(ParentFolder,PIDL,SHGDN_FORPARSING);
      //в список передаются текстовое описание и PIDL объекта
      FolderList.AddObject(ParsingPath, TObject(pidl));
    end;
    Result:=True;
  end;
end;
```

Атрибуты объекта

Для получения сведений о характеристиках одного или нескольких файловых объектов следует воспользоваться методом

```
function GetAttributesOf(cidl: UINT; var pidl: PItemIDList;
                         var rgfInOut: UINT): HResult;
```

Здесь: `cidl` — количество объектов, чьими атрибутами мы интересуемся; `pidl` — адрес массива указателей на структуры `ITEMIDLIST`, каждая из которых уникально идентифицирует файловый объект относительно родительской папки. Каждая структура `ITEMIDLIST` должна содержать одну структуру `SHITEMID`, завершающуюся нулем. Выходной параметр `rgfInOut` используется как для передачи в метод данных, так и для возврата результатов ее работы. Как входной параметр, он определяет, какие именно атрибуты мы планируем получить. Как выходной параметр, он содержит запрашиваемые данные — комбинации флагов представлены в табл. 44.3. В случае успеха метод `GetAttributesOf()` возвратит значение `S_OK`.

***Таблица 44.3.** Флаги параметра `rgfInOut` метода `GetAttributesOf()`*

Флаг	Описание
SFGAO_CANCOPY	Файловые объекты и папки могут быть скопированы
SFGAO_CANDELETE	Файловые объекты и папки могут быть удалены
SFGAO_CANLINK	Для файловых объектов и папок допускается создание ярлыков
SFGAO_CANMONIKER	Для файловых объектов и папок возможно создание псевдонимов
SFGAO_CANMOVE	Разрешено перемещение файловых объектов и папок
SFGAO_CANRENAME	Разрешено переименование
SFGAO_DROPTARGET	Файлы и папки могут быть целевыми в операциях перетаскивания
SFGAO_HASPROPSHEET	Файлы и папки обладают страницей свойств
SFGAO_GHOSTED	Файловые объекты и папки должны отображаться полупрозрачным значком (например, после операции вырезки в буфер обмена)
SFGAO_LINK	Файловые объекты являются ярлыками
SFGAO_READONLY	Файлы и папки доступны только для чтения
SFGAO_SHARE	Папки являются разделяемым ресурсом
SFGAO_HASSUBFOLDER	Папка может иметь дочерние папки
SFGAO_COMPRESSED	Выбранные элементы сжаты (например, архиватором ZIP)
SFGAO_FILESYSTEM	Выбранные папки и файловые объекты являются частью файловой системы
SFGAO_FILESYSANCESTOR	Выбранные папки содержат хотя бы один системный файл
SFGAO_FOLDER	Элемент является папкой
SFGAO_NEWCONTENT	Объект имеет новое содержание
SFGAO_NONENUMERATED	Элемент, не поддающийся идентификации
SFGAO_REMOVABLE	Папки и файловые объекты находятся на съемном устройстве
SFGAO_VALIDATE	Необходима проверка корректности кэшируемой информации. Например: хотя бы один из выбранных объектов, чьи атрибуты запрошены, уже не существует

Листинг 44.11 демонстрирует порядок работы с методом `GetAttributesOf()`, в предложенном коде мы осуществляем проверку файла (папки) на режим доступа только для чтения.

Листинг 44.11. Проверка файлового объекта на доступ только для чтения

```
function IsReadOnly(ParentFolder: IShellFolder;
                    PIDL: PItemIDList):boolean;
var Flags: LongWord;
begin
  Flags := SFGAO_DISPLAYATTRMASK;
  ParentFolder.GetAttributesOf(1, PIDL, Flags);
  Result:= SFGAO_READONLY and Flags <> 0;
end;
```

Сравнение объектов папки

Для сравнения (определения порядка следования) двух элементов `pidl1` и `pidl2` применяют метод

```
function CompareIDs(Param : Integer;
                    pidl1, pidl2 : PItemIDList) : HRESULT;
```

Первый аргумент метода определит механизм сравнения элементов. Наиболее часто применяется сортировка по имени объекта, такому режиму сверки соответствует нулевое значение параметра `Param`. Результаты сравнения возвращаются в `HRESULT` в виде числа `SmallInt`:

- меньше нуля: `pidl1<pidl2`;
- больше нуля: `pidl1>pidl2`;
- ноль: `pidl1=pidl2`.

> ***Примечание***
>
> Возможности интерфейса `IShellFolder` развивает его "коллега" — интерфейс `IShellFolder2`.

ГЛАВА 45

DataSnap

Язык программирования Delphi весьма многогранен. Мы с вами рассмотрели уже несколько способов проектирования распределенных приложений, в частности научились строить клиент-серверные проекты на основе сетевых технологий именованных каналов и сокетов, подробно изучили фирменный способ корпорации Microsoft, основанный на идее COM. В этой главе мы познакомимся еще с одним перспективным направлением проектирования клиент-серверных систем — технологией DataSnap.

История DataSnap началась вместе с Delphi 6, именно тогда DataSnap пришел на смену устаревшей MIDAS. Вплоть до выхода Delphi 2009 заложенные в DataSnap идеи существенным модификациям не подвергалась. Кардинальные изменения произошли в Delphi 2010. В этой версии языка программисты Embarcadero сделали важный шаг — устранили зависимость DataSnap от модели COM. В Delphi XE существенной доработке подвергся парк ориентированных на DataSnap компонентов и были разработаны мастера, упрощающие процесс создания приложений сервера и клиента. Одно из ключевых достоинств технологии DataSnap — независимость от платформы, поэтому, начиная с Delphi XE2, технология DataSnap с равным успехом может применяться как в проектах VCL (ориентированных на 32- и 64-разрядную Windows), так и в проектах FMX (поддерживающих как Windows, так и Mac OS X).

Современная версия DataSnap XE2 способна на многое:

- DataSnap, безусловно, поддерживает традиционный для клиент-серверной архитектуры обмен данными между сервером и клиентом, клиент запрашивает от сервера определенную услугу (обращается к методу сервера), сервер ее выполняет. Немаловажен тот факт, что в качестве параметра метода может быть задействован не только классический тип данных (целое или вещественное число, строка и т. п.), но и объект — потомок класса `TPersistent`;
- технология обеспечивает механизм обратного вызова, благодаря которому сервер получает возможность обратиться к методу клиента;
- DataSnap может задействоваться для разработки многоуровневых приложений баз данных. В таких проектах сервер DataSnap играет роль промежуточного слоя — сервера приложений, управляя доступом клиентов к системе управления базами данных;
- коммуникационный канал может быть построен на базе стека протокола TCP/IP и высокоуровневого протокола HTTP (включая HTTPS);
- передаваемые данные могут быть защищены с помощью криптографии;
- сервер поддерживает механизм аутентификации и авторизации.

Архитектура проекта DataSnap

Технология DataSnap ориентирована на создание проектов, работающих в режиме "клиент-сервер" и в многоуровневых приложениях баз данных. Сервер DataSnap представляет собой отдельный модуль, реализованный в виде обычного приложения или консольного приложения или службы Windows.

Концептуально сервер DataSnap состоит из двух взаимодействующих модулей: модуля контейнера сервера и модуля методов сервера (рис. 45.1). Контейнер сервера `TServerContainer` — весьма неординарный класс. Если при создании проекта вы укажете, что сервер должен представлять собой консоль или классическое приложение с формой VCL, то мастер построения сервера DataSnap создаст `TServerContainer` на основе обычного модуля данных `TDataModule`. Но если мы в самом начале проектирования объявим, что создаем сервис Windows, то в цепочку наследования контейнера попадет еще один класс — класс `TService`, инкапсулирующий службу (рис. 45.2).

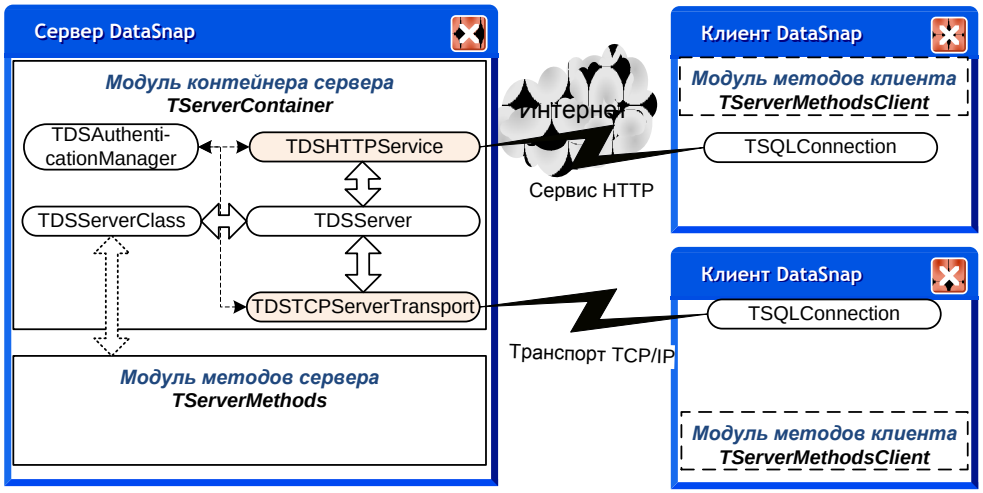


Рис. 45.1. Обобщенная архитектура проекта DataSnap

Контейнер сервера вместе с располагаемыми на нем компонентами отвечает за взаимодействие с удаленными клиентскими приложениями, для этого задействуется стек протоколов TCP/IP или более высокоуровневый протокол HTTP. Выбор того или иного протокола обмена в первую очередь определяется условиями функционирования сервера DataSnap. Если все клиентские приложения и сервер приложений сосредоточены в рамках локальной сети предприятия, то стоит остановить свой выбор на транспорте TCP/IP. Если роль опорной сети исполняет Интернет и активно задействованы Web-ресурсы, то используем всю мощь HTTP (в том числе HTTPS). В случае смешанного характера доступа к ресурсам БД технология DataSnap допускает создание приложений, поддерживающих оба протокола одновременно.

Воспользовавшись услугами компонента `TDSAuthenticationManager`, посвятившего себя обеспечению проверки подлинности клиента, мы сможем осуществлять аутентификацию и авторизацию клиентского приложения.

Модуль методов сервера представляет собой логическое ядро сервера приложений DataSnap. В исходном коде модуля методов объявляются и описываются дополнительные

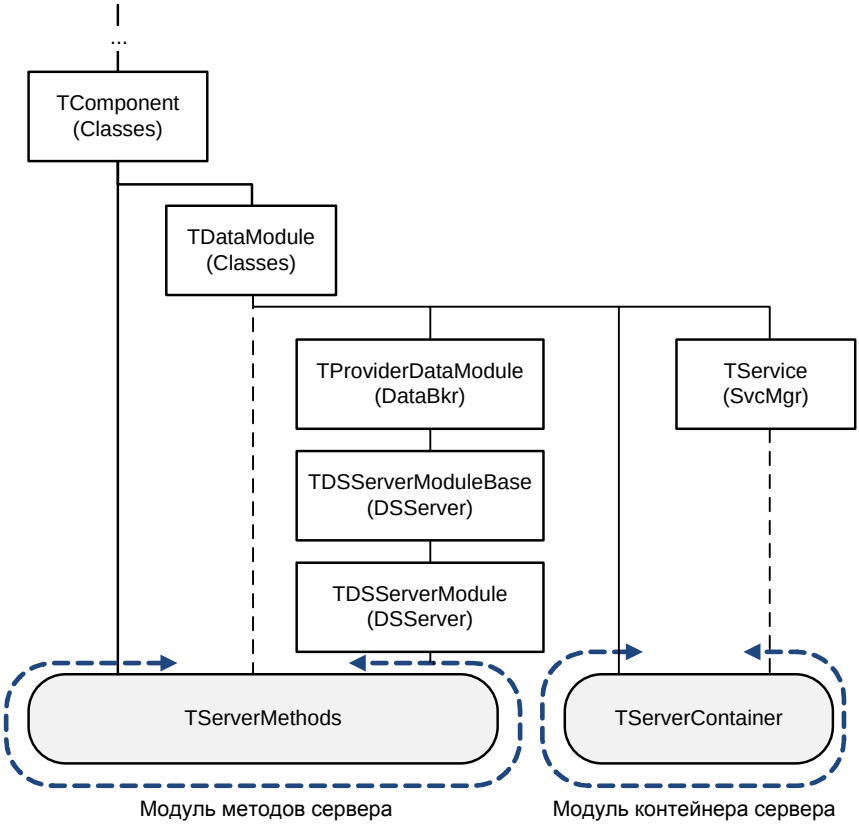


Рис. 45.2. Иерархия построения модулей `TServerContainer` и `TServerMethods`

методы сервера, которые могут быть предоставлены в распоряжение клиента, например, сервер в состоянии помочь клиентам осуществлять сложные математические расчеты.

Цепочка наследования модуля методов сервера также зависит от поставленных перед сервером задач. В простейшем случае `TServerMethods` может быть создан на основе простейшего класса `TComponent`. Если для функционирования модуля методов необходима помощь невизуальных элементов управления, то в цепочку наследования включается модуль данных `TDataModule`. Наконец, если логика функционирования сервера DataSnap подразумевает использование динамических методов, потребуется помощь еще трех промежуточных классов (см. рис. 45.2).

Нам осталось дать общую характеристику третьего ингредиента проекта DataSnap — клиентского приложения. Для того чтобы клиентское приложение получило возможность обращаться к методам сервера, у клиента создается собственный модуль методов (см. рис. 45.1), выступающий логическим проводником идей модуля методов сервера. Если в модуле сервера объявляется функция с условным названием `MyFunction()`, то в модуле методов клиента должна быть объявлена одноименная функция с аналогичным перечнем параметров. Клиентское приложение обращается к своему методу, даже не подозревая, что технология DataSnap позаботится перенаправить вызов к серверу. Сервер обработает запрос клиента и возвратит результат обратно в вызывающую функцию клиента. Сразу заострю ваше внимание на том, что совпадение заголовков методов сервера и клиента ни в коем

случае не говорит об идентичности реализации этих методов. Код метода сервера и метода клиента различаются на все сто процентов, но разговор об этом мы также ненадолго отложим, ведь вводная часть немного затянулась, и нам уже давно пора приступить к обсуждению компонентов проекта DataSnap.

Компоненты сервера

Рассмотрим основные компоненты, необходимые для построения сервера DataSnap.

Сервер *TDSServer*

Основу сервера приложений составляет компонент `TDSServer`. Это своего рода диспетчерский центр всего сервера. Во-первых, компонент отвечает за организацию связи между сервером DataSnap и его клиентами, для этого компонент управляет информационными потоками TCP/IP и HTTP. Во-вторых, `TDSServer` несет ответственность за предоставление в распоряжение клиентов методов сервера, для этого он обеспечивает пересылку запросов клиентов в модуль методов сервера. В-третьих, компонент позволяет создавать дополнительный коммуникационный туннель для передачи собственных сообщений в адрес клиентских приложений.

Класс описан в модуле `DSServer`, а соответствующий ему компонент вы обнаружите на странице **DataSnap Server** палитры компонентов Delphi. Во время разработки проекта компонент следует разместить в модуле контейнера сервера `TServerContainer`.

Несмотря на столь ответственный пост, `TDSServer` весьма скромен. На странице **Properties** Инспектора объектов у компонента опубликован минимум свойств. Самое главное из них:

```
property AutoStart: Boolean; //по умолчанию true
```

В состоянии `true` это свойство обеспечит автоматический старт сервера сразу после создания экземпляра класса. Кроме того, для запуска и останова сервера могут применяться методы:

```
procedure Start;
procedure Stop;
```

О факте успешного запуска можно судить по состоянию свойства

```
property Started: Boolean;
```

Второе интересующее нас свойство

```
property HideDSAdmin: Boolean; //по умолчанию false
```

определяет степень доверия сервера приложений к подключенным к нему клиентам. По умолчанию сервер разрешает клиентам обращаться к встроенным в него методам администрирования `DSAdmin`, но при желании компонент может и "повредничать", установив свойство в значение `false`. Как следствие, клиенты DataSnap окажутся на голодном пайке.

Обработка событий

Основное назначение всех имеющихся в распоряжении компонента `TDSServer` событий — информирование о поведении клиентских приложений. В момент подключения к серверу очередного клиента генерируется событие

```
property OnConnect: TDSConnectEvent;
type TDSConnectEvent =
     procedure(DSConnectEventObject: TDSConnectEventObject) of object;
```

С отключением клиента также связано отдельное событие:

```
property OnDisconnect: TDSConnectEvent;
```

Обратите внимание на тот факт, что оба события возвращают ссылку на экземпляр класса `TDSConnectEventObject`. В недрах этого объекта скрывается весьма полезная информация об особенностях соединения с клиентом (свойство `ConnectProperties`) и характеристики транспортного канала между сервером и клиентом (свойство `ChannelInfo`).

Чтобы воспользоваться услугами перечисленных событий, для ведения протокола работы сервера нам понадобится осуществить ряд подготовительных мероприятий. В первую очередь объявим глобальную файловую переменную

```
var LogFile:Textfile;
```

которая станет ссылаться на текстовый файл с информацией о работе сервера.

Подключение к файлу протокола производим в момент создания модуля контейнера так, как это продемонстрировано в листинге 45.1.

Листинг 45.1. Связывание файловой переменной с файлом протокола

```
procedure TServerContainer1.DataModuleCreate(Sender: TObject);
begin
  AssignFile(LogFile, 'datasnap.log'); //доступ к файловой переменной
  if FileExists('datasnap.log')=true then //если файл существует
     Append(LogFile) else //режим добавления строк
     Rewrite(LogFile);    //создание нового пустого файла
end;
```

В момент завершения работы сервера (листинг 45.2) разрываем связь с файловой переменной.

Листинг 45.2. Освобождение файловой переменной

```
procedure TServerContainer1.DataModuleDestroy(Sender: TObject);
begin
  CloseFile(LogFile); //закрываем файл
end;
```

В строках листинга 45.3 отражено самое главное — код обработки события, возникающего в момент подключения к серверу очередного клиента.

Листинг 45.3. Вносим в протокол данные о соединениях

```
procedure TServerContainer1.DSServer1Connect(
                          DSConnectEventObject: TDSConnectEventObject);
var i:integer;
begin
  WriteLn(LogFile, DateTimeToStr(now)+
                   ' соединение '+DSConnectEventObject.ChannelInfo.Info);
```

```
  WriteLn(LogFile,'================================');
  i:=DSConnectEventObject.ConnectProperties.Count;
  for i:=0 to DSConnectEventObject.ConnectProperties.Count-1 do
    WriteLn(LogFile,DSConnectEventObject.ConnectProperties.Properties[i]);
  WriteLn(LogFile,'================================');
end;
```

После того как клиентское приложение обращается к какому-либо методу сервера, на сервере возникает событие

```
property OnPrepare: TDSPrepareEvent;
type TDSPrepareEvent = procedure
                (DSPrepareEventObject: TDSPrepareEventObject) of object;
```

В рамках этого обработчика события программист может получить данные о подготовке к выполнению метода. Как минимум сможем выяснить имя вызываемого метода `MethodAlias`.

Заключительное событие

```
property OnError: TDSErrorEvent;
type TDSErrorEvent = procedure(
                      DSErrorEventObject: TDSErrorEventObject) of object;
```

генерируется при возникновении на сервере исключительных ситуаций.

Если вы нуждаетесь в очень подробной трассировке работы сервера, то обязательно обратите внимание на событие

```
property OnTrace: TDBXTraceEvent;
TDBXTraceEvent = function(TraceInfo: TDBXTraceInfo): CBRType of object;
```

Основной параметр события — объявленная в модуле `DBXCommon` запись `TDBXTraceInfo`. Запись содержит четыре поля-свойства (табл. 45.1), позволяющие обрабатывать данные, поступающие в момент генерации события.

***Таблица 45.1.** Свойства структуры `TDBXTraceInfo`*

Свойство	Описание
`property Message: UnicodeString;`	Текстовое сообщение
`property TraceFlag: TDBXTraceFlag;`	Флаги трассировки (`None`, `Prepare`, `Execute`, `Error`, `Command`, `Connect`, `Transact`, `Blob`, `Misc`, `Vendor`, `Parameter`, `Reader`, `DriverLoad`, `MetaData`, `Driver`, `Custom`) позволяют идентифицировать проводимые в данный момент операции
`property TraceLevel: TDBXTraceLevel;`	Уровень трассировки может принимать следующие значения: `None`, `Debug`, `Info`, `Warn`, `Error`, `Fatal`
`property CustomCategory: UnicodeString;`	Дополнительная классификация сообщения

ВНИМАНИЕ!

Если вы планируете воспользоваться событием `OnTrace()`, проверьте, чтобы к строке `uses` модуля были подключены файлы `DBXCommon` и `DBCommonTypes`.

Для вывода результатов трассировки можно задействовать код из листинга 45.4.

Листинг 45.4. Трассировка

```
function TServerContainer1.DSServer1Trace(
                                    TraceInfo: TDBXTraceInfo): CBRType;
begin
  WriteLn(LogFile, 'Трассировка '+TraceInfo.CustomCategory);
  WriteLn(LogFile, #9+TraceInfo.Message);
  Result:=cbrUSEDEF; //действие по умолчанию
end;
```

В результате вся дополнительная информация будет направляться в файл протокола.

Класс сервера *TDSServerClass*

В проектах DataSnap компонент `TDSServerClass` несет прямую ответственность за предоставление методов, опубликованных на сервере, клиентским приложениям. Компонент располагается в контейнере сервера и подключается к компоненту `TDSServer` с помощью свойства

```
property Server;
```

Еще одно свойство, которому следует уделить внимание, определяет особенности жизненного цикла экземпляра класса `TDSServerClass`:

```
property LifeCycle:string; //по умолчанию Session
```

Существуют три варианта настройки жизненного цикла компонента: `Session`, `Server` и `Invocation`. По умолчанию свойство устанавливается в состояние `Session`. Это означает, что в момент подключения к серверу клиентского приложения специально для него формируется отдельный объект `TDSServerClass`. Созданный объект обслуживает клиента на протяжении всей сессии и уничтожается сразу после отключения клиентского приложения от сервера. Перевод жизненного цикла в режим `Server` принципиально поменяет работу приложения, сервер будет обладать одним-единственным экземпляром `TDSServerClass`. И этот экземпляр постарается обслуживать все входящие подключения. Наконец, состояние `Invocation` может пригодиться в тех случаях, когда вы планируете, чтобы экземпляр объекта `TDSServerClass` появлялся на свет для обслуживания каждого обращения к методу сервера и выгружался из памяти сразу после выполнения метода. Таким образом, каждый очередной вызов метода обслуживается новым экземпляром `TDSServerClass`.

Наиболее примечательным событием компонента `TDSServerClass` является

```
property OnGetClass: TDSGetClassEvent;
TDSGetClassEvent = procedure (DSServerClass: TDSServerClass;
                        var PersistentClass: TPersistentClass) of object;
```

Это событие подготавливает появление на свет очередного экземпляра класса `TDSServerClass` (или любого другого заданного программистом класса), определяя особенности жизненного цикла экземпляра класса. Для того чтобы упростить работу программиста, в Delphi предусмотрен мастер создания проекта сервера DataSnap. Мастер самостоятельно добавит в код обработчика события `OnGetClass()` всего одну строку (листинг 45.5), указывающую на то, что создаваемый экземпляр класса должен продублировать настройки компонента расположенного в контейнере сервера.

Листинг 45.5. Создание экземпляра класса `TDSServerClass`

```
procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TServerMethods1;
end;
```

Процесс создания и уничтожения объекта `TDSServerClass` сопровождает пара событий:

```
property OnCreateInstance: TDSCreateInstanceEvent;
property OnDestroyInstance: TDSDestroyInstanceEvent;
```

Вы их можете использовать, например, для подсчета числа обращений к методам.

Заключительное событие

```
property OnPrepare: TDSPrepareEvent;
```

позволяет управлять процессом подготовки метода сервера к выполнению.

Транспортные компоненты *TDSTCPServerTransport* и *TDSHTTPService*

При построении транспортных каналов между клиентами и сервером DataSnap разработчик вправе сделать выбор между двумя вариантами обмена данными:

- низкоуровневое соединение на основе стека протоколов TCP/IP с применением компонента `TDSTCPServerTransport`;
- высокоуровневый сервис на базе протокола HTTP с применением компонента `TDSHTTPService`.

Соединение на основе стека TCP/IP наиболее универсальное. На сегодняшний день данный протокол является доминирующим как в локальных, так и в глобальных сетях. Поэтому разработанное с использованием компонента `TDSTCPServerTransport` приложение сохранит работоспособность как в сети предприятия, так и на просторах Интернета. Протокол HTTP также базируется на основе стека TCP/IP, но обладает более развитыми возможностями по работе в Web-среде. Поддержка сервиса HTTP осуществляется силами компонента `TDSHTTPService`, отличительная черта которого — поддержка *протокола репрезентативной передачи состояния* (Representational State Transfer, REST). Это новый архитектурный стиль, позволяющий клиентам обращаться к ресурсам сервера.

При построении ответственных за организацию транспортного канала компонентов разработчиками Delphi использовалась общая цепочка наследования VCL (рис. 45.3). Благодаря наличию одинаковых предков оба компонента приобрели несколько схожих черт. В первую очередь это и свойство, с помощью которого компоненты подключаются к серверу `TDSServer`:

```
property Server:TDSCustomServer;
```

Во вторую очередь коллекция фильтров:

```
property Filters: TTransportFilterCollection;
```

Фильтры определяют особенности процесса передачи потока байтов. Например, с Delphi поставляется фильтр ZLibCompression, позволяющий сжимать поток данных между сервером и клиентом. Кроме того, возможно создавать собственные транспортные фильтры.

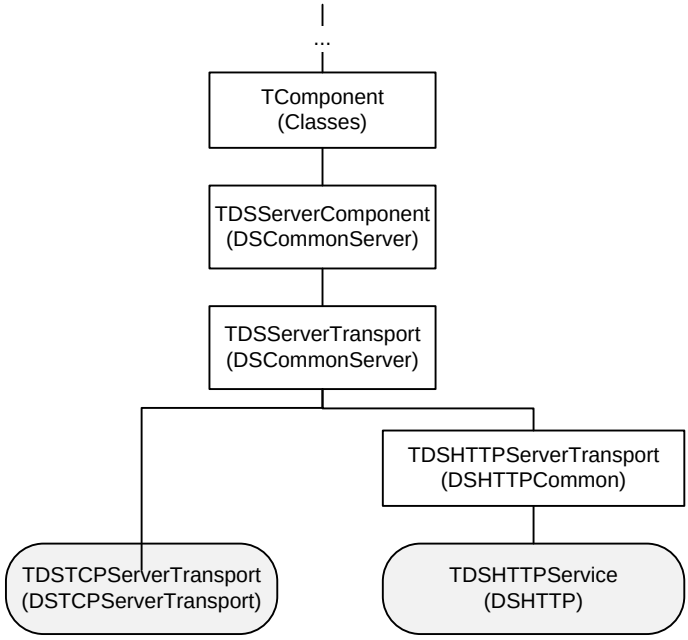


Рис. 45.3. Иерархия наследования `TDSTCPServerTransport` и `TDSHTTPService`

Индивидуальные черты компонентов вы обнаружите в табл. 45.2 и 45.3.

Таблица 45.2. *Описание параметров соединения* `TDSTCPServerTransport`

Свойство	Назначение
`property Port: Integer;`	Номер порта, к которому производится подключение клиента, по умолчанию за DataSnap закреплен порт 211
`property MaxThreads: Integer;`	Максимальное количество одновременно поддерживаемых потоков, по умолчанию ограничения отсутствуют — свойство установлено в 0
`property PoolSize: Integer;`	Размер пула соединений, по умолчанию равно 10. Свойство должно быть настроено до старта сервера
`property BufferKBSize: Integer;`	Размер буфера в килобайтах для кэширования операций чтения и записи, по умолчанию равен 32 Кбайт

Компонент `TDSTCPServerTransport` не обладает обработчиками событий.

Таблица 45.3. *Описание параметров соединения* `TDSHTTPService`

Свойство	Назначение
`property HttpPort: Word;`	Номер прослушиваемого порта (по умолчанию за HTTP закреплен 8080-й порт). Если на компьютере уже используется служба HTTP, то номер порта следует изменить
`property Active: Boolean;`	Включение и отключение соединения

Таблица 45.3 *(окончание)*

Свойство	Назначение
property RESTContext: String	Определяет URL-адрес для вызова сервера в качестве службы REST
property DSHostname: String;	Альтернатива свойству Server, обеспечивает способ подключения к серверу DataSnap по имени
property DSPort: Integer;	Используется только совместно со свойством DSHostname. Определяет номер порта
property AuthenticationManager: TDSHTTPServiceAuthenticationManager;	Свойство для подключения менеджера аутентификации, обеспечивающего проверку подлинности поступающей на сервер информации
property ServerSoftware: String;	Свойство информирует о программном обеспечении сервера, доступно только для чтения

На вооружении компонента TDSHTTPService имеется пять обработчиков событий. Из них четыре обслуживают стиль доступа REST и поэтому для относительно несложных клиент-серверных приложений существенного интереса не представляют. Оставшееся событие

```
property Trace: TDSHTTPServiceTraceEvent;
type TDSHTTPServiceTraceEvent = procedure(Sender: TObject;
          AContext: TDSHTTPContext; ARequest: TDSHTTPRequest;
                                    AResponse: TDSHTTPResponse) of object;
```

специализируется на трассировке работы компонента. Оно может пригодиться, если обмен между клиентом и сервером осуществляется в формате HTTP.

> ***Замечание***
>
> Очень полезным качеством сетевой составляющей DataSnap является возможность обмена данными внутри туннелей. Этот факт позволяет снизить проблему настройки сетевых экранов между сервером и клиентами.

Менеджер аутентификации *TDSAuthenticationManager*

При организации сетевого обмена в сервере DataSnap имеет смысл использовать компонент TDSAuthenticationManager. Благодаря ему мы приобретем возможность осуществлять дополнительную проверку подлинности передаваемых данных.

> ***Замечание***
>
> Компонент TDSAuthenticationManager появился в Delphi XE, он вобрал в себя функционал устаревших компонентов TDSTCPAuthenticationManager и TDSHTTPAuthenticationManager.

Компонент отвечает за авторизацию и аутентификацию запустившего клиентское приложение пользователя на стороне сервера DataSnap.

Аутентификация (authentication) — это процедура проверки подлинности пользователя. Обычно пользователь подтверждает то, что он является именно тем, за кого он себя выдает путем ввода в систему уникальной (неизвестной другим) информации о себе. В простейшем случае это символьный пароль. Если пользователь успешно прошел процедуру аутентификации СУБД, осуществляет его авторизацию.

Авторизация (authorization) — это процедура предоставления пользователю определенных ресурсов и прав на их использование.

Системы безопасности подавляющего большинства современных серверов для определения прав пользователей используют заранее сформированные списки привилегий. Администратор системы обладает самым широким перечнем полномочий, опытный пользователь — меньшим, полномочия рядового пользователя еще более сужены. Перечень заранее сформированных полномочий называют *ролью* (role). Роль значительно упрощает процесс администрирования подсистемы авторизации — вновь появившийся пользователь просто наделяется правами из той или иной роли. Это гораздо удобнее, чем явное описание перечня полномочий для каждого пользователя в отдельности.

На стороне сервера процесс аутентификации поддерживается компонентом `TDSAuthenticationManager` и сопровождается генерацией события

```
property OnUserAuthenticate: TDSAuthenticationEvent;
type  TDSAuthenticationEvent = procedure(Sender: TObject;
                const Protocol, Context, User, Password: UnicodeString;
                   var valid: boolean; UserRoles: TStrings) of object;
```

Если переданные клиентом DataSnap идентификационные данные корректны, параметру `valid` следует передать значение `true`. Последний параметр `UserRoles` содержит список ролей, к которым будет принадлежать пользователь.

Ключевое свойство компонента `TDSAuthenticationManager` — свойство-коллекция, хранящее роли сервера DataSnap:

```
property Roles: TDSRoleItems;
```

Отдельный элемент коллекции представляет собой роль `TDSRoleItem` с тремя свойствами, представленными в табл. 45.4.

Таблица 45.4. *Свойства роли* `TDSRoleItem`

Свойство	Назначение
`property AuthorizedRoles : TStrings;`	Список допустимых пользователей
`property DeniedRoles : TStrings;`	Список недопустимых пользователей
`property ApplyTo : TStrings;`	Список методов сервера и классов, к которым может быть применена роль

Если клиент DataSnap успешно преодолеет барьер аутентификации, то может быть осуществлена (но необязательно) его авторизация. Наделение клиента полномочиями сопровождается событием

```
property OnUserAuthorize: TDSAuthorizationEvent;
type  TDSAuthorizationEvent = procedure(Sender: TObject;
                    AuthorizeEventObject: TDSAuthorizeEventObject;
                                 var valid: boolean) of object;
```

Компоненты клиента

Для разработки клиентского приложения DataSnap как минимум потребуется помощь компонента `TSQLConnection`, отвечающего за установку соединения с сервером.

Соединение *TSQLConnection*

Наличие в проекте клиента DataSnap компонента `TSQLConnection` — обязательное условие для организации взаимодействия с сервером DataSnap. Компонент отвечает за два направления работы:

- управление соединением с сервером DataSnap;
- передачу серверу сведений, аутентифицирующих клиента.

Несмотря на многогранность и сложность компонента (он в том числе используется в клиент-серверных проектах баз данных), его настройка для работы в клиентском приложении DataSnap не вызывает никаких затруднений.

> ***ЗАМЕЧАНИЕ***
>
> Сфера применения компонента-соединителя `TSQLConnection` весьма обширна и не ограничивается только проектами DataSnap. Наличие в имени компонента префикса SQL подсказывает, что наш новый знакомый в первую очередь нацелен на работу в проектах БД.

Для обеспечения соединения клиентского приложения с сервером DataSnap в период визуального проектирования программисту требуется позаботиться о настройке комплексного свойства `Driver`. Для этого в Инспекторе объектов IDE Delphi требуется настроить следующие параметры:

- в строке `Driver` ввести текстовую строку `DataSnap`;
- развернув строку `Driver`, конкретизировать параметры соединения:
  - имя хоста `HostName`, на котором развернут сервер;
  - протокол соединения `CommunicationProtocol` (`tcp/ip` или `http`);
  - номер порта `Port` (по умолчанию для TCP/IP — 211, для HTTP — 8080, для HTTPS — 8081);
  - при необходимости указать пароль `Password`, имя `UserName` и другие параметры.

Если настройка соединения должна осуществляться динамически, во время выполнения клиентского приложения, то следует все параметры соединения передать в свойство

```
property Params: TStrings;
```

Физически свойство `Params` представляет собой список строк, хранящих пары "имя параметра — значение".

Задавая параметры, нам вновь придется определиться с именем хоста, на котором развернут сервер, именем базы данных, при необходимости с именем и паролем пользователя.

Для осуществления операции подключения и отключения клиента DataSnap от сервера у компонента `TSQLConnection` объявлены методы:

```
procedure Open;
procedure Close;
```

Тот же результат достигается за счет обращения к свойству

```
property Connected: Boolean;
```

Кроме того, по состоянию `Connected` можно судить о факте соединения с сервером.

Допустим, что мы создаем универсальное клиентское приложение DataSnap, которое должно позволять пользователю самостоятельно настраивать параметры соединения. Тогда нам

кроме собственно компонента TSQLConnection потребуется помощь ряда элементов управления:

- две кнопки выбора TRadioButton для определения сетевого протокола взаимодействия клиента и сервера (TCP/IP или HTTP);
- две строки ввода TEdit для ввода имени хоста (компонент edHostName) и номера порта (компонент edPort);
- кнопка btnOpen:TButton для активизации соединения.

Порядок настройки основных параметров соединения представлен в листинге 45.6.

Листинг 45.6. Динамическая настройка основных параметров соединения

```
procedure TForm1.btnOpenClick(Sender: TObject);
begin
with SQLConnection1.Params do
begin
  Clear;
  Values['DriverUnit']:='DBXDataSnap';    //драйвер
  Values['DatasnapContext']:='datasnap/'; //контекст
  Values['HostName'] :=edHostName.Text;   //имя хоста

  if RadioButton1.Checked then {выбор сетевого протокола}
      Values['CommunicationProtocol']:='tcp/ip' else //выбор TCP/IP
      Values['CommunicationProtocol']:='http';       //выбор HTTP
  Values['Port'] :=edPort.Text; //номер порта
  Values['Filters'] :='{}';     //набор фильтров
end;
SQLConnection1.Open;
end;
```

С процессом установления и разрыва соединения связан перечень событий, представленных в табл. 45.5.

***Таблица 45.5.** События, связанные с соединением и разрывом соединения*

Событие	Описание
`property BeforeConnect: TNotifyEvent;`	Генерируется перед установкой соединения
`property AfterConnect: TNotifyEvent;`	Генерируется после установки соединения
`property BeforeDisconnect: TNotifyEvent;`	Генерируется перед разрывом соединения
`property AfterDisconnect: TNotifyEvent;`	Генерируется после разрыва соединения

В коде, предложенном в листинге 47.6, не предусмотрена отправка в сторону сервера имени и пароля. Если такая необходимость существует, то проконтролируйте, чтобы в свойстве

```
property LoginPrompt:boolean;
```

было установлено значение true. В этом случае на экран компьютера отобразится диалоговое окно регистрации пользователя и будет вызвано событие

```
property OnLogin: TSQLConnectionLoginEvent;
type TSQLConnectionLoginEvent = procedure(Database: TSQLConnection;
                                          LoginParams: TStrings) of object;
```

В череде событий компонента `TSQLConnection` событие `OnLogin()` генерируется сразу после `BeforeConnect()`.

О текущем состоянии соединения можно судить по свойству

```
property ConnectionState: TConnectionState;
```

Возможные варианты состояний вы найдете в табл. 45.6.

Таблица 45.6. *Возможные значения* `TConnectionState`

Состояние	Описание
`csStateClosed`	Соединение отсутствует
`csStateOpen`	Компонент соединен с сервером
`csStateConnecting`	Процесс соединения инициирован, но еще не завершен
`csStateExecuting`	Компонент отправил в адрес сервера запрос, и он выполняется
`csStateFetching`	Компонент получает информацию от сервера
`csStateDisconnecting`	Момент разрыва соединения, но оно еще не завершено

Проект DataSnap с использованием мастера

После того как мы с вами получили все необходимые для самостоятельной разработки проекта DataSnap теоретические знания, перейдем к практике.

Нашу работу мы начнем с создания простого сервера приложений DataSnap. Для этого выбираем пункт меню **File | Other** и в окне **New Items** находим значок **DataSnap Server**. Щелчок по кнопке **OK** активизирует мастер по созданию сервера приложений.

На экране вашего компьютера отобразится окно мастера создания сервера **DataSnap Server**, позволяющее за несколько шагов настроить базовые параметры проекта (рис. 45.4). Начнем с определения типа приложения. На наш выбор Delphi предлагает три варианта будущего сервера: классическое приложение VCL, консольное приложение и сервис Windows. В первом окне мастера укажите, что мы намерены создать обычное приложение **VCL Forms Application**. Для пробы пера это наиболее подходящий выбор, но при этом надо понимать, что вариант обычного приложения VCL или консоли не вполне подходит для реализации профессиональных проектов. Причина в следующем — сервер приложений должен находиться в постоянной готовности к обслуживанию запросов клиентов. Однако если сервер реализован в формате обычного приложения, то для его запуска (даже если приложение установлено в автозагрузку) необходим вход пользователя в систему Windows, а это не всегда возможно. Поэтому в полноценных приложениях DataSnap единственным верным выбором становится разработка сервиса — **Service Application**. После соответствующей настройки сервисы Windows стартуют автоматически вместе с запуском системы, и в их работу не требуется никакое внешнее вмешательство.

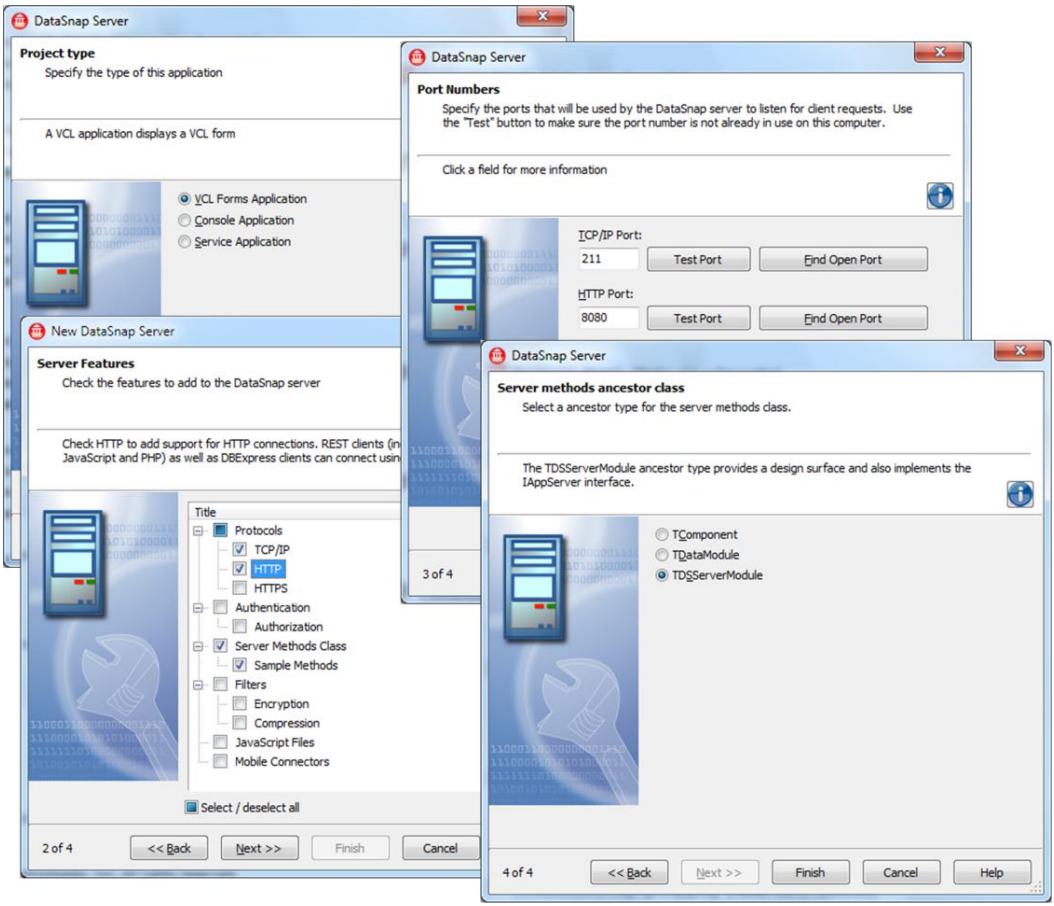


Рис. 45.4. Настройка параметров сервера приложений с помощью мастера

> ***Замечание***
>
> Если вы начинающий программист, то свои первые шаги в DataSnap лучше сделайте в проекте VCL Forms, это значительно упростит отладку проекта. Однако надо понимать, что полноценный сервер DataSnap должен быть реализован только в качестве сервиса Windows.

Второе окно мастера позволяет выбрать коммуникационный протокол, обеспечивающий взаимодействие между сервером и клиентами DataSnap. В нашем примере мы включим поддержку как TCP/IP, так и HTTP. Откажемся от услуг аутентификации, поэтому не ставим флажок **Authentication**. Кроме того, мастер предлагает нам создать демонстрационные методы сервера, об этом напоминает элемент **Sample Methods**. Мы не станем возражать и против этой помощи: методы-примеры помогут нам быстрее понять порядок объявления методов сервера.

В третьем окне необходимо определиться с номерами портов, которые станет прослушивать наш сервер. По умолчанию мастер предлагает задействовать порт 211 для стека протоколов TCP/IP и порт 8080 для протокола HTTP.

В завершающем работу мастера четвертом окне нам следует настроить порядок создания класса методов сервера. Мастер предлагает сделать выбор между тремя отправными клас-

сами: `TDSServerModule`, `TDataModule` и `TComponent`. Наилучшее решение — `TDSServerModule`. Согласимся с ним, и тогда в дальнейшем у нас не возникнет никаких проблем с динамическим вызовом методов сервера.

> ***Замечание***
>
> Если вы намерены воспользоваться протоколом HTTPS для обеспечения безопасной передачи данных, то у помощника создания сервера DataSnap добавится еще одно окно, предлагающее выбрать файл сертификата.

Щелчок по кнопке **OK** приведет к созданию проекта сервера приложений DataSnap. В проекте вы обнаружите следующий перечень модулей:

- модуль методов сервера `ServerMethods1` и соответствующий ему программный модуль ServerMethodsUnit1.pas;
- контейнер сервера `ServerContainer1` с модулем ServerContainerUnit1.pas. Заметьте, что мастер подготовил к работе все необходимые для создания сервера компоненты (рис. 45.5);
- форму `Form1` и ее программный модуль Unit1.pas;
- головной модуль будущего сервиса — `Project1`.

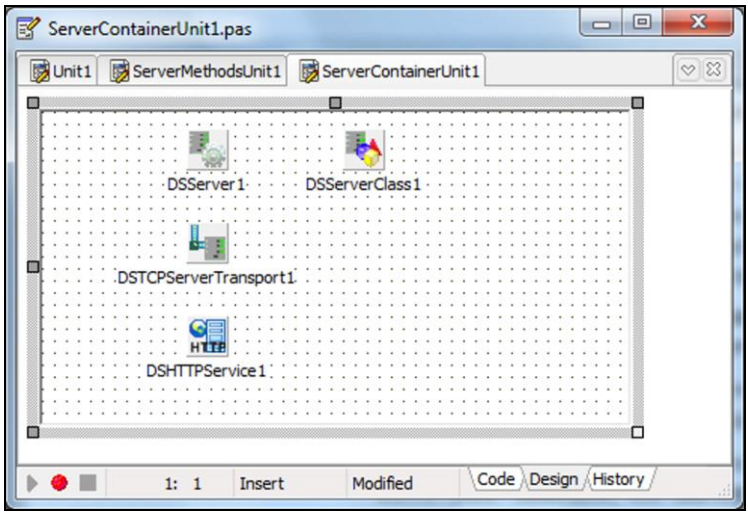


Рис. 45.5. Модуль контейнера сервера DataSnap

Сохраните проект в отдельном каталоге, пока не переименовывая модули, единственное исключение сделаем для имени головного файла проекта — назовем его DataSnapServer.dproj.

Изучите приведенный в листинге 45.7 код модуля методов ServerMethodsUnit1.pas.

Листинг 45.7. Листинг модуля методов `ServerMethodsUnit1`

```
unit ServerMethodsUnit1;
interface

uses SysUtils, Classes, DSServer;
```

```
type TServerMethods1 = class(TDSServerModule)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;//демонстрационные методы
    function ReverseString(Value: string): string;
  end;

implementation
{$R *.dfm}

function TServerMethods1.EchoString(Value: string): string;
begin
  Result := Value; //метод повторяет входящую строку
end;

function TServerMethods1.ReverseString(Value: string): string;
begin
  Result := StrUtils.ReverseString(Value); //реверс строки
end;
end.
```

В код модуля с нашего согласия мастером включены демонстрационные методы `EchoString()` и `ReverseString()`. Задача методов заключается в "передразнивании" клиентского приложения — они возвращает клиенту DataSnap полученную от него же текстовую строку.

Модуль контейнера сервера ServerContainerUnit1.pas также относительно мал (листинг 45.8) и не содержит никаких подводных камней. Самое первое, на что следует обратить внимание — контейнер `TServerContainer1` создается на основе модуля `TDataModule`. Это позволит размещать на модуле данных невизуальные компоненты.

Листинг 45.8. Модуль контейнера сервера `ServerContainerUnit1`

```
unit ServerContainerUnit1;

interface

uses SysUtils, Classes, DSTCPServerTransport, DSHTTPCommon, DSHTTP,
     DSServer, DSCommonServer, DSAuth;

type TServerContainer1 = class(TDataModule)
     DSServer1: TDSServer;
     DSTCPServerTransport1: TDSTCPServerTransport;
     DSHTTPService1: TDSHTTPService;
     DSAuthenticationManager1: TDSAuthenticationManager;
     DSServerClass1: TDSServerClass;
     procedure DSServerClass1GetClass(DSServerClass: TDSServerClass;
                                      var PersistentClass: TPersistentClass);
```

```
  private
    { Private declarations }
  public
  end;

var ServerContainer1: TServerContainer1;

implementation
uses Windows, ServerMethodsUnit1;

{$R *.dfm}

procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TServerMethods1;
end;
end.
```

Мастер DataSnap, формируя модуль контейнера сервера, самостоятельно объявил и реализовал метод компонента `DSServerClass1` под названием `GetClass()`, отвечающий за создание новых экземпляров класса `TServerMethods1`, которые будут появляться на свет для обслуживания каждого входящего клиентского подключения.

Наш сервер уже вполне работоспособен. Но для того чтобы опробовать его в деле, нам следует создать клиентское приложение.

Подготовка клиентского приложения

Основой клиентского приложения DataSnap может выступать обычный проект VCL. Расположите на главной форме компонент `SQLConnection1: TSQLConnection`. Сохраните проект в отдельном каталоге под любым именем.

> ***ВНИМАНИЕ!***
> Перед началом проектирования клиентского приложения запустите сервер DataSnap!

Выберите компонент `SQLConnection1`. Выберите свойство `Driver` и установите его в состояние `DataSnap`. Щелкнув в Инспекторе объектов по символу + слева от свойства `Driver`, разверните вложенные свойства. Свойству `CommunicationProtocol` присвойте значение `TCP/IP` и обязательно проверьте номер порта `Port`, по умолчанию он должен принимать значение `211`. В свойстве `HostName` укажите имя или IP-адрес хоста, на котором развернута служба DataSnap. Если вы осуществляете отладку сервера и клиента на одном компьютере, то воспользуйтесь аббревиатурой `localhost`. Отключите диалог ввода имени и пароля (на первых порах нам он не нужен), установив в `false` свойство `LoginPrompt`. Проверьте корректность подключения к службе сервера приложений, временно установив в состояние `true` свойство `Connected`. Убедившись в готовности `SQLConnection1` сотрудничать с разработанной нами службой DataSnap, верните свойство `Connected` в исходное состояние.

> ***ЗАМЕЧАНИЕ***
> При желании для создания заготовки клиентского приложения DataSnap можно воспользоваться мастером. Для этого после создания нового проекта VCL следует выбрать пункт меню **File | New | Other** и в окне **New Items** найти значок **DataSnap Client Module**.

Щелкните правой кнопкой мыши по компоненту SQLConnection1. В контекстном меню компонента найдите пункт **Generate DataSnap client classes**. Выбор этого элемента меню заставит Delphi сгенерировать модуль методов клиента (листинг 45.9).

> ***Внимание!***
>
> Если вами разработан проект DataSnap в более ранней версии Delphi, при переходе на Delphi XE2 обязательно регенерируйте клиентский класс!

Листинг 45.9. Автоматически созданный класс TServerMethods1Client

```
type
  TServerMethods1Client = class(TDSAdminClient)
  private
    FEchoStringCommand: TDBXCommand;
    FReverseStringCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
                       AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
  end;
```

Среди методов класса TServerMethods1Client для нас самый большой интерес представляют методы EchoString() и ReverseString(). Это не что иное, как способ вызова объявленных на сервере DataSnap одноименных методов (они станут работать "эхом" — возвращать клиенту его же текстовый запрос). Вновь сохраните проект, во время сохранения переименуйте автоматически сгенерированный модуль методов клиента в MethodsClient.pas.

Воспользовавшись пунктом меню **File | Use Unit...**, подключите к главной форме проекта модуль методов клиента MethodsClient.pas. Затем разместите на главной форме клиента строку ввода Edit1 и метку Label1, перейдите к событию OnChange() строки ввода и напишите там представленные в листинге 45.10 строки кода.

Листинг 45.10. Вызов демонстрационного метода сервера EchoString()

```
procedure TForm1.Edit1Change(Sender: TObject);
var SMC:TServerMethods1Client;
begin
  SMC:=TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  Label1.Caption:=SMC.EchoString(Edit1.Text);
  SMC.Free;
end;
```

В момент создания формы нам следует подключиться к службе сервера DataSnap. В простейшем случае это будет выглядеть так, как показано в листинге 45.11.

Обратная операция осуществляется при закрытии формы (листинг 45.12).

Листинг 45.11. Активируем соединение

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  SQLConnection1.Open;
end;
```

Листинг 45.12. Разрыв соединения

```
procedure TForm1.FormClose(Sender: TObject);
begin
  SQLConnection1.Close;
end;
```

Клиент готов к работе. Запустите проект и протестируйте работу метода `EchoString()`. Для обращения к методу `ReverseString()` вам достаточно переработать код из листинга 45.10.

Создание нового метода на сервере DataSnap

После того как мы с вами проверили свои силы на демонстрационных методах (которые автоматически объявляются во время работы мастера DataSnap), пора разработать собственный метод.

Вновь откройте наш проект сервера и перейдите к модулю методов (в нашей версии проекта он называется `ServerMethodsUnit1`). В области `public` объявления класса `TServerMethods1` опишите заголовок функции `GetServerDateTime()`. Как вы догадались, новый метод сервера возьмет на себя ответственность за уведомление клиента о своем системном времени (листинг 45.13).

Листинг 45.13. Доработка модуля методов сервера

```
...
{$METHODINFO ON} //добавьте эту директиву компилятора
type  TServerMethods1 = class(TDSServerModule)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
    function GetServerDateTime:TDateTime; //новый метод
  end;
{$METHODINFO OFF}//добавьте эту директиву компилятора

implementation
{$R *.dfm}

function TServerMethods1.GetServerDateTime: TDateTime;
begin
  Result:=Now;
end;
```

Обратите внимание, что в листинге 45.13 появились две директивы компилятора: `{$METHODINFO ON}` и `{$METHODINFO OFF}`. Первая из них указывает компилятору на необходимость включения в исполняемый файл дополнительной информации RTTI описания методов сервера. Наличие дополнительных сведений упростит разработку клиента. Директива `{$METHODINFO OFF}` указывает компилятору, что блок методов завершен.

Откомпилируйте проект, а затем вновь запустите сервер, выбрав пункт меню **Run | Run Without Debugging**.

Доступ к новому методу из клиентского приложения

Для того чтобы клиентское приложение смогло воспользоваться новым методом сервера DataSnap, нам необходимо регенерировать модуль методов клиента DataSnap. Эта операция должна осуществляться при наличии доступа клиентского приложения к выполняющемуся серверу. Убедившись, что сервер в работе, откройте проект клиентского приложения и вызовите контекстное меню компонента `SQLConnection1`, выбираем элемент меню **Generate DataSnap client classes** (генерировать модуль методов клиента) и обновляем модуль методов клиента.

Открыв модуль MethodsClient.pas, вы увидите, что он дополнился новым методом `GetServerDateTime()`, позволяющим узнать о дате/времени сервера. Как видите, клиент даже в период разработки достаточно легко отслеживает появление новых методов на сервере DataSnap.

> ***Замечание***
>
> В распоряжении программиста Delphi имеется специальный компонент, который позволяет очень просто обращаться ко всем имеющимся в распоряжении сервера методам, он называется `TSqlServerMethod`.

После того как клиентское приложение узнало о существовании на стороне сервера метода `GetServerDateTime()`, его вызов остался делом техники — для этого понадобятся услуги кнопки `btnServerDateTime` и метки `lbServerDateTime:TLabel` (листинг 45.14).

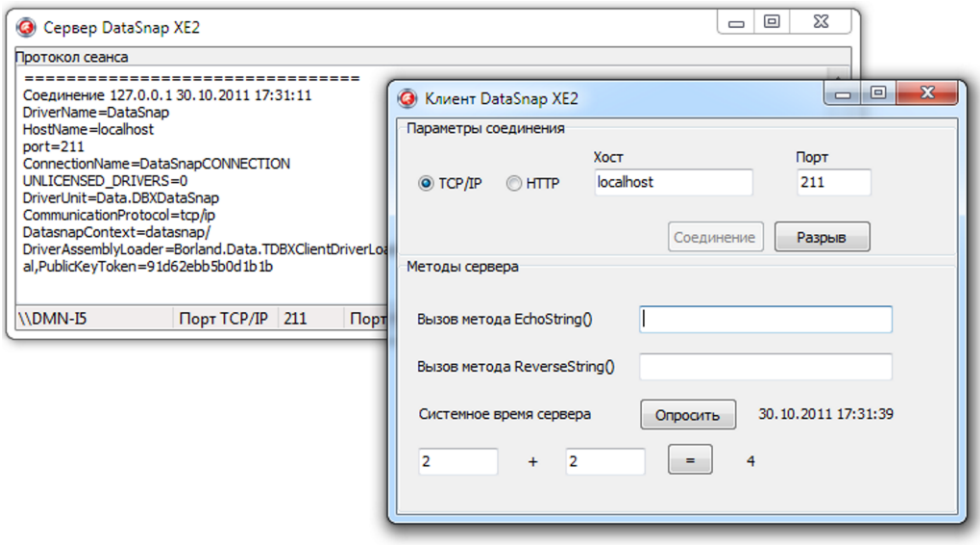


Рис. 45.6. Сервер и клиент DataSnap в работе

Листинг 45.14. Вызов метода `GetServerDateTime()` клиентом

```
procedure TForm1.btnServerDateTimeClick(Sender: TObject);
var SMC:TServerMethods1Client;
begin
  SMC:=TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  lbServerDateTime.Caption:=DateTimeToStr(SMC.GetServerDateTime);
  SMC.Free;
end;
```

Запускаем клиента и проверяем его работоспособность (рис. 45.6).

Проект DataSnap на основе пользовательского класса

В самом начале главы мы уже упомянули тот факт, что в роли параметров методов DataSnap могут задействоваться не только классические типы данных, но и любые классы Delphi — потомки `TPersistent`. Согласитесь, что передача объектов между приложениями, выполняющимися на различных хостах, — весьма нетривиальный сервис!

Сервер

Приступим к разработке сервера. На этот раз, создавая сервер DataSnap, не станем использовать встроенный мастер, а соберем проект "вручную". Для этих целей запустим новый проект **VCL Forms Application** и расположим на главной форме следующие компоненты:

- сервер `DSServer1:TDSServer`;
- класс сервера `DSServerClass1:TDSServerClass`, воспользовавшись свойством `Server` подключаем компонент к серверу `DSServer1`;
- транспорт TCP/IP `DSTCPServerTransport1:TDSTCPServerTransport`, воспользовавшись свойством `Server` подключаем компонент к серверу `DSServer1`.

При необходимости (если 211-й порт уже занят) подберите подходящий номер порта в свойстве `Port` компонента `DSTCPServerTransport1`.

Традиционно главную форму будущего сервера назовем `frmMain` и дадим команду на сохранение проекта. Во время сохранения приложения переименуем программный модуль главной формы в main.pas, а весь проект назовем dssrvcalc.dproj.

Теперь приступим к разработке класса, который станет служить средством обмена данными между сервером и клиентом. Для описания класса целесообразно создать отдельный программный модуль, поэтому выбираем пункт меню **File | New | Unit**. После появления в редакторе нового пустого модуля сразу сохраним его, на этот раз под именем `DSCalculator`.

Название модуля говорит само за себя — мы намерены разработать класс `TCalculator`, способный выполнять 4 основных арифметических операции (листинг 45.15).

Листинг 45.15. Листинг с описанием класса калькулятора `TCalculator`

```
unit DSCalculator;
interface

uses classes;
```

```
type
{$METHODINFO ON}
  TCalculator= class(TPersistent)
  public
   function additional(X,Y:double):double;      //сложение
   function subtraction(X,Y:double):double;     //вычитание
   function multiplication(X,Y:double):double; //умножение
   function division(X,Y:double):double;        //деление
end;
 {$METHODINFO OFF}

implementation

function TCalculator.additional(X, Y: double): double;
begin
  Result:=X+Y;
end;

function TCalculator.division(X, Y: double): double;
begin
  Result:=X/Y;
end;

function TCalculator.multiplication(X, Y: double): double;
begin
  Result:=X*Y;
end;

function TCalculator.subtraction(X, Y: double): double;
begin
  Result:=X-Y;
end;
end.
```

Подключите модуль калькулятора к программному модулю главной формы сервера. Для этого достаточно, открыв в редакторе кода модуль main.pas, выбрать пункт меню **File | Use unit** и выбрать в списке `DSClass`.

- Теперь нам следует позаботиться о создании экземпляра класса калькулятора при обращении к нему из клиентского приложения. Для этого предназначено событие `OnGetClass()` компонента `DSServerClass1` (листинг 45.16).

Листинг 45.16. Обработка события `OnGetClass()`

```
implementation
uses DSCalculator;

{$R *.dfm}
```

```
procedure TfrmMain.DSServerClass1GetClass(DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass:=TCalculator;
end;
```

Откомпилируйте проект и запустите сервер на выполнение. Разработка клиентского приложения должна осуществляться при включенном сервере.

Клиент

Создаем новый проект обычного приложения VCL и сохраняем его в отдельной папке (выбор имен модулей оставим на ваше усмотрение). Расположите на форме клиента следующие компоненты (рис. 45.7):

- компонент-соединитель `TSQLConnection`, настройте параметры соединения (свойство `Driver='DataSnap'`, протокол `CommunicationProtocol='tcp/ip'`, имя хоста `HostName` и номер порта `Port`), отключите свойство `LoginPrompt`. Проверьте корректность параметров соединения, ненадолго активировав свойство `Connected`;
- две строки ввода — `EditX:TEdit` и `EditY:TEdit`;
- группу переключателей `RadioGroup1:TRadioGroup`, позволяющую выбирать арифметическую операцию с четырьмя строками в свойстве `Items` (+, –, /, *);
- кнопку `Button1:TButton`, вызывающую ту или иную операцию в зависимости от выбора пользователя;
- метку `Label1:TLabel`, отображающую полученный результат.

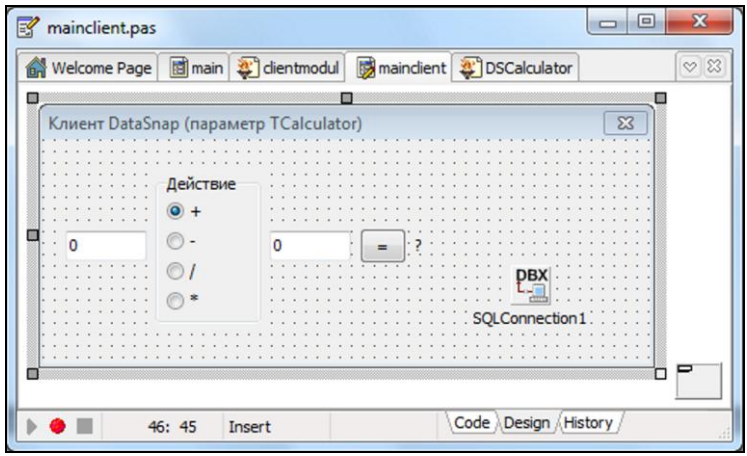


Рис. 45.7. Пользовательский интерфейс клиентского приложения

Выберите компонент `TSQLConnection` и в его контекстном меню найдите пункт **Generate DataSnap Client classes**. Щелчок по меню заставит Delphi создать модуль методов клиента с описанием класса `TCalculatorClient`. Сразу сохраним полученный код под именем clientmodul.pas, а после этого рассмотрим заголовок автоматически сгенерированного класса `TCalculatorClient` (листинг 45.17).

Листинг 45.17. Объявление класса TCalculatorClient в модуле методов клиента

```
type
  TCalculatorClient = class(TDSAdminClient)
  private
    FadditionalCommand: TDBXCommand;
    FsubtractionCommand: TDBXCommand;
    FmultiplicationCommand: TDBXCommand;
    FdivisionCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
                       AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function additional(X: Double; Y: Double): Double;
    function subtraction(X: Double; Y: Double): Double;
    function multiplication(X: Double; Y: Double): Double;
    function division(X: Double; Y: Double): Double;
  end;
```

Подключим к программному модулю главной формы клиентского приложения только что созданный модуль методов с описанием класса TCalculatorClient, для этого вызываем меню **Project | Add file to project**, переходим в папку с исходным кодом сервера и в ней находим файл clientmodul.pas.

Нам осталось позаботиться о подключении к серверу DataSnap и описать событие-щелчок по единственной кнопке клиентского приложения (листинг 45.18).

Листинг 45.18. Соединение с сервером и вызов метода класса

```
unit mainclient;
...

implementation
uses clientmodul;

{$R *.dfm}

procedure TfrmMainClient.FormCreate(Sender: TObject);
begin
  SQLConnection1.Open; //открываем соединение
end;

procedure TfrmMainClient.Button1Click(Sender: TObject);
var Calculator: TCalculatorClient;
    X, Y, Z: Double;
begin
  Calculator := TCalculatorClient.Create(SQLConnection1.DBXConnection);
  try
    X := StrToFloatDef(EditX.Text,0);
    Y := StrToFloatDef(EditY.Text,0);
```

```
    case RadioGroup1.ItemIndex of
      0 : z:=Calculator.additional(X,Y);      //сложение
      1 : z:=Calculator.subtraction(X,Y);     //вычитание
      2 : z:=Calculator.division(X,Y);        //деление
      3 : z:=Calculator.multiplication(X,Y); //умножение
    end;
    Label1.Caption := FloatToStr(Z);
  finally
    Calculator.Free;
  end;
end;

procedure TfrmMainClient.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  SQLConnection1.Close; //разрыв соединения
end;
```

Механизм обратного вызова

В технологии DataSnap поддерживается механизм обратного вызова, благодаря которому сервер сможет ненадолго поменяться местами с клиентом и приобрести право обратиться к клиентскому методу. Для того чтобы это произошло, необходимо выполнить одно обязательное условие — сервер должен получить от клиента ссылку на его функцию обратного вызова.

> ***Замечание***
>
> При реализации обратного вызова взаимодействие между сервером и клиентом осуществляется в текстовом формате обмена данными, основанном на JavaScript (JavaScript Object Notation, JSON). Данные передаются в виде массива, каждая из ячеек которого представляет собой пару: имя элемента и его значение. В Delphi формат JSON описан в модуле `DBXJSON`, который следует подключить как к серверу, так и к клиенту DataSnap.

Ссылка на функцию обратного вызова передается в адрес сервера в виде типизированного параметра `TDBXCallback` (параметр описан в модуле `DBXJSON`). Для этого в модуле методов клиента объявляется класс-потомок `TDBXCallback`, в котором переписывается метод `Execute()`. Средством взаимодействия между сервером и клиентом служит единственный аргумент метода `Arg`. Это структура `TJSONValue`, позволяющая представлять массив пар JSON. Реализация метода `Execute()` может выглядеть так, как показано в листинге 45.19.

Листинг 45.19. Объявление метода обратного вызова на стороне клиента

```
uses ...,  DBXJSON;
...
type
    TDemoCallback= class(TDBXCallback)
  public
    function Execute(const Arg:TJSONValue): TJSONValue; override;
end;
...
```

```
implementation
...
function TDemoCallback.Execute(const Arg: TJSONValue): TJSONValue;
var s:string;
begin
  Result:=TJSONValue(Arg.Clone);
  s:=TJSONObject(Result).Get(0).JsonString.ToString
                      +#13+TJSONObject(Result).Get(0).JSONValue.Value;
  ShowMessage(s);
end;
```

Теперь перейдем к серверу DataSnap. Допустим, что сервер-калькулятор объявляет метод, позволяющий получать квадратный корень из числа и поддерживающий возможность обратного вызова. В этом случае среди аргументов метода вы обнаружите параметр обратного вызова `Callback: TDBXCallback`. Реализация метода предложена в листинге 45.20.

Листинг 45.20. Реализация метода сервера с возможностью вызова клиента

```
function TCalculator.GetSqrt(X: double; Callback: TDBXCallback): double;
var JSONPair:TJSONPair;
    JSONObject:TJSONObject;
begin
 Result:=sqrt(X);
 JSONPair:=TJSONPair.Create('Метод выполнен в ',DateTimeToStr(Now));

 JSONObject:=TJSONObject.Create;
 JSONObject.AddPair(JSONPair);

 Callback.Execute(JSONObject);
end;
```

Помимо выполнения операции сложения сервер формирует пару JSON, в которую заносит информацию о дате и времени выполнения метода. И благодаря функции обратного вызова `Callback.Execute()` инициирует выполнение этой функции у клиента.

Из-за необходимости передавать ссылку на функцию обратного вызова несколько меняется порядок обращения из клиентского приложения к методу сервера (листинг 45.21).

Листинг 45.21. Обращение к методу сервера со стороны клиента

```
var x,z:double;
    Calculator: TCalculatorClient;
begin
  try
    x:=StrToFloatDef(Edit1.Text,0);

    Calculator:= TCalculatorClient.Create(SQLConnection1.DBXConnection);
    z:=Calculator.GetSqrt(x, TDemoCallback.Create);
    Label1.Caption:=FloatToStr(z);
```

```
  finally
    Calculator.Free;
  end;
end;
```

Проверьте работоспособность метода получения квадратного корня. Благодаря механизму обратного вызова сервер DataSnap получил возможность обращаться к методу `TDemoCallback.Execute` клиентского приложения.

ГЛАВА 46

LiveBindings

Число принципиально новых возможностей, появившихся в Embarcadero RAD Studio XE2, без преувеличения можно сравнить с прорывом первой версии Delphi в середине 1990-х годов. В этой главе нам предстоит обсудить очередную новацию Embarcadero — технологию "живого связывания" LiveBindings, позволяющую программисту организовать передачу данных в рамках приложения между произвольным источником и объектом-получателем данных.

Несмотря на то, что собственно идея живого связывания нашла свое полноценное программное воплощение в Embarcadero RAD Studio XE2, фундамент LiveBindings был заложен значительно раньше — еще в 2008 году. Именно тогда вышла в свет Delphi 2009 с принципиально усовершенствованным ядром RTTI (Run-time type information). Не стану перечислять положительные качества нового RTTI, отмечу лишь то, что с этого момента появилась возможность создать универсальную технологию, способную обеспечить взаимодействие между объектами приложения, строго говоря, без опоры на конкретную операционную систему. Поэтому технология LiveBindings одинаково хорошо функционирует как в классической библиотеке VCL, так и в новой кроссплатформенной FireMonkey.

Что такое LiveBindings? Если коротко, то это технология взаимодействия двух и более объектов. Связь может быть как односторонней, так и двусторонней. В качестве участника взаимодействия выступают два типа объектов: *управляемые объекты* (control objects) и *источники данных* (source objects). Обмен данными осуществляется между свойствами объектов, а правила обмена определяются с помощью регулярных *выражений* (expression). Что немаловажно, в выражения разрешается включать ссылки на другие внешние объекты.

> ***Внимание!***
> Строго говоря, за связь между двумя объектами LiveBindings отвечает третий объект, например, экземпляр класса `TBindExpression`.

Вводный пример LiveBindings

Чтобы сразу "потрогать руками" саму идею применения LiveBindings в проектах Delphi, создадим небольшой пример, позволяющий осуществить прямой обмен данными между двумя визуальными компонентами.

Запускаем новый проект VCL и размещаем на главной форме строку ввода `TEdit` и метку `TLabel`. В нашем примере метке отведена роль управляемого объекта — в ее свойство `Caption` станут поступать данные из свойства `Text` управляющего объекта `TEdit`.

Все подготовительные операции завершены, теперь нам следует осуществить связывание объектов. Распишем эту операцию по пунктам.

1. Выберите управляемый объект `Label1` и на странице событий (**Events**) в Инспекторе объектов найдите строку **LiveBindings**.
2. Для создания новой "живой связи" щелкните левой кнопкой мыши по элементу **New Binding** в строке Инспектора объектов (рис. 46.1).

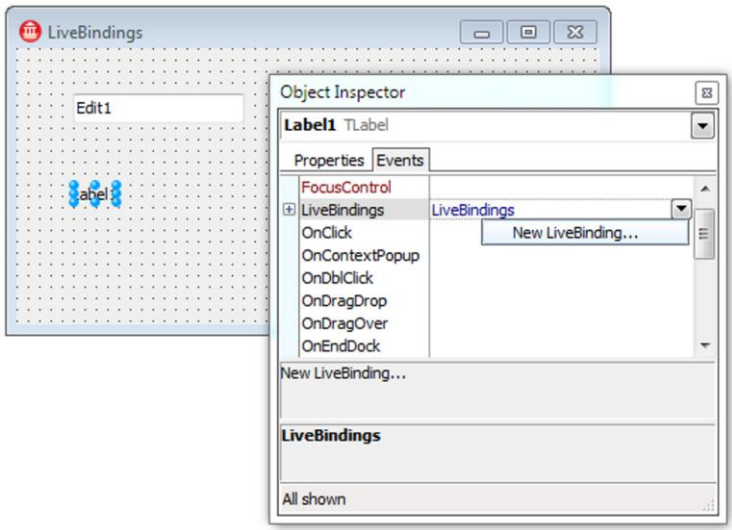


Рис. 46.1. Создание новой "живой связи" для компонента `Label1`

В ответ на щелчок по элементу **New Binding** среда проектирования сделает две вещи. Во-первых, на главной форме нашего проекта появится третий компонент — `BindingsList1:TBindingsList`. Это коллекция связей, в которой станут храниться все создаваемые в проекте связи LiveBindings. Во-вторых, на экране отобразится диалог выбора **New LiveBinding**, предлагающий пользователю указать разновидность связи.

3. В диалоге **New LiveBinding** выберите узел `TBindExpression` (это укажет Delphi на то, что мы намерены создать связь на основе выражения) и нажмите кнопку **OK** (рис. 46.2).

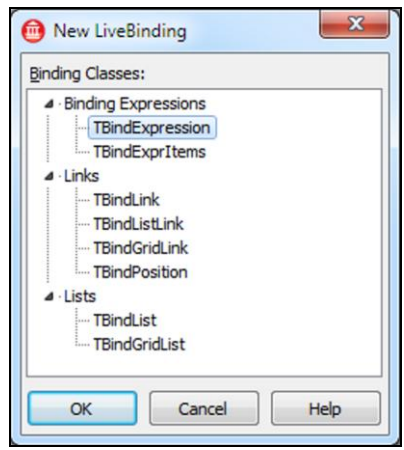


Рис. 46.2. Выбор типа выражения для связи

4. Для проверки создания новой связи дважды щелкните по только что любезно размещенному средой Delphi на форме нашего проекта менеджеру `BindingsList1:TBindingsList`. В ответ на это действие среда проектирования отобразит на экране редактор с перечнем

доступных связей. В нашем примере выбор связей не богат — это только что созданная нами связь `BindExpressionLabel11`.

5. Найдите в Инспекторе объектов связь `BindExpressionLabel11: TBindExpression` и перейдите на страницу свойств **Properties**. Теперь нам предстоит настроить ряд свойств связи (рис. 46.3):

```
ControlComponent:=Label1;    //управляемый объект
ControlExpression:=Caption; //получатель — заголовок
Managed:=true;               //выражение активно
SourceComponent:=Edit1;      //источник данных
SourceExpression:=Text;      //свойство источника данных
```

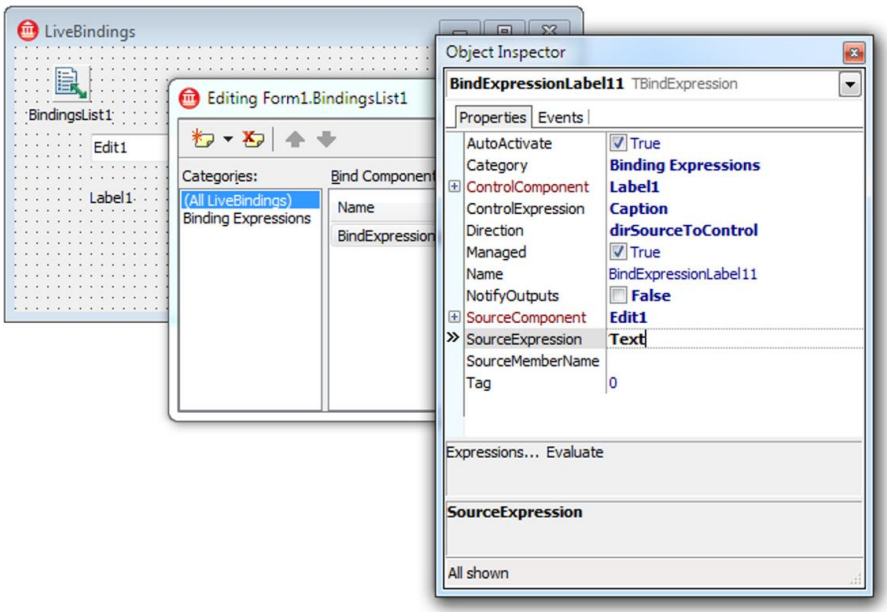


Рис. 46.3. Настройка свойств связи `TBindExpression`

Для завершения задуманного нам осталось немного поработать за клавиатурой. Листинг 46.1 содержит обработчик события `OnChange()` строки ввода. Это событие генерируется с каждым изменением содержимого компонента `Edit1`.

Листинг 46.1. Событие `OnChange()`

```
implementation
uses System.Bindings.Helper;
{$R *.dfm}

procedure TForm1.Edit1Change(Sender: TObject);
begin
  TBindings.Notify(Edit1, 'Text');
end;
```

На первый взгляд может показаться, что заданного результата (передачи текста из строки ввода в метку) мы достигли слишком высокой ценой. Но пример преследовал другую цель — обратить ваше внимание на то, что текст из строки ввода в заголовок метки отправился неявно. Действительно, ведь в коде даже нет и намека на `Label1.Caption:= Edit1.Text`!

> ***Замечание***
>
> В электронном архиве к книге *(см. приложение 10)* вы найдете пример, в котором связь организуется между компонентами, расположенными на разных формах. Особенность упоминаемого примера в том, что благодаря LiveBindings объект-источник передает данные управляемому объекту, даже не подозревая о его существовании!

Класс *TBindExpression*

Повторяя вводный пример, вы наверняка заметили, что основным действующим лицом первого приложения стал экземпляр класса `TBindExpression` (модуль `Data.Bind.Components`). Именно `TBindExpression` взял на себя ответственность за организацию взаимодействия между двумя компонентами: объектом-источником и управляемым объектом. Для этого понадобилось лишь внести ряд настроек в свойства объекта.

Ссылка на объект-источник заносится в свойство

```
property SourceComponent: TComponent;
```

Сведения об управлении окажутся неполными до тех пор, пока мы не опишем входное управляющее выражение в свойстве

```
property SourceExpression: string;
```

В простейшем случае управляющее выражение будет хранить имя свойства объекта-источника с данными, которые следует передать управляемому объекту.

Ссылка на второй участник взаимодействия LiveBindings заносится в свойство

```
property ControlComponent: TComponent;
```

Кроме имени управляемого объекта следует описать выходное управляющее выражение, для этого предназначено свойство

```
property ControlExpression: string;
```

Вновь заметим, что в простейшей ситуации здесь достаточно упомянуть название свойства, в которое поступят "живые" данные.

Направление потока данных между объектами определяется свойством

```
property Direction: TExpressionDirection;//dirSourceToControl
```

По умолчанию поток данных направлен от источника к управляемому объекту, однако существуют еще два варианта направлений:

```
type TExpressionDirection = (
             dirSourceToControl, //источник -> управляемый объект
             dirControlToSource, //управляемый объект -> источник
             dirBidirectional);  //двустороннее взаимодействие
```

Для построения примера двустороннего взаимодействия от нас не потребуется особых трудозатрат. Разместите на форме два компонента `TTrackBar`. Выбрав любой из компонентов,

создайте связь LiveBindings. Данные должны передаваться между свойствами `Position` (отвечающими за местоположение бегунка), а направление связи `Direction` установите в состояние `dirBidirectional` (рис. 46.4).

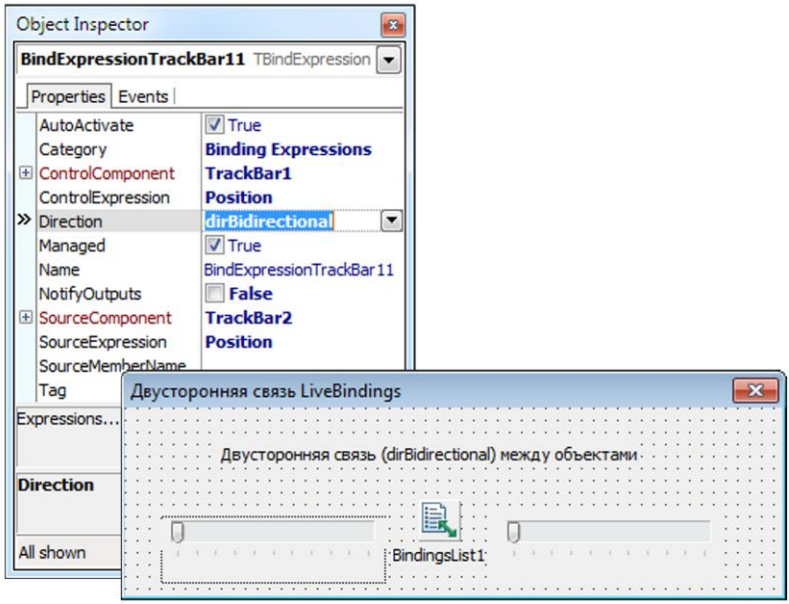


Рис. 46.4. Организация двусторонней связи LiveBindings

Для завершения примера выберите обработчик события `OnChange()` одного из компонентов и внесите в него всего одну строку кода (листинг 46.2). Последний штрих таков — сделайте событие общим для обоих компонентов.

Листинг 46.2. Отправка уведомления в событии `OnChange()`

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
   TBindings.Notify(Sender, '');
end;
```

Продолжим рассмотрение характеристик класса `TBindExpression`. Для деактивации объекта `TBindExpression` устанавливаем в `false` свойство

```
property Active: Boolean;
```

С переходом в активное и пассивное состояния связаны два события (табл. 46.1).

Таблица 46.1. *События* `TBindingExpression`

Событие	Описание
`property OnActivating: TNotifyEvent;`	Связь активируется
`property OnActivated: TNotifyEvent;`	Связь деактивируется

Таблица 46.1 *(окончание)*

Событие	Описание
property OnAssigningValueEvent: TBindingAssigningValueEvent; **type** TBindingAssigningValueEvent = **procedure**(AssignValueRec: TBindingAssignValueRec; **var** Value: TValue; **var** Handled: Boolean) **of object**;	В объект поступает новое значение Value. Установив переменную Handled в состояние true, указываем объекту, что обработка события завершена
property OnEvalErrorEvent: TBindingEvalErrorEvent; **type** TBindingEvalErrorEvent = **procedure**(AException: Exception) **of object**;	Генерируется в момент возникновения исключительной ситуации AException
property OnAssignedValueEvent: TBindingAssignedValueEvent; **type** TBindingAssignedValueEvent = **procedure**(AssignValueRec: TBindingAssignValueRec; **const** Value: TValue) **of object**;	В объект поступило новое значение Value

Еще один способ приостановки взаимодействия обеспечивает свойство

```
property Managed: Boolean; //по умолчанию true
```

Отказавшись от значения `true`, мы изымем связь из-под контроля менеджера связей.

Как и положено полноценному объекту, выражение обладает способностью реагировать на события (см. табл. 46.1).

Выражение LiveBindings

Взаимодействие между объектами LiveBindings осуществляется в форме регулярных выражений. Повторив вводный пример, мы с вами практически не задумывались над порядком создания выражения — за нас все сделала Delphi. Однако пытливого программиста такое положение вещей вряд ли устроит.

По своей сути выражение LiveBindings представляет собой обычную текстовую строку, отформатированную в соответствии с очень простыми правилами. Отправное условие формирования выражения заключается в том, что в строке выражения должно быть отведено место для описания ассоциации между объектами Delphi и их псевдонимами.

Допустим, что в нашей программе имеются два объекта источника данных — `ObjX` и `ObjY`, оба источника обладают свойством `Value`. Если мы намерены сформировать выражение LiveBindings, позволяющее суммировать числовые значения, хранящиеся в свойствах `Value` этих объектов, то можно поступить следующим образом. Объекту `ObjX` присваивается псевдоним `X`, а объекту `ObjY` — `Y` (рис. 46.5). Имена свойств в псевдонимах не нуждаются и передаются в выражение такими, какие они есть: "`X.Value+Y.Value`".

На роль управляемого объекта (получающего результат операции сложения) можно назначить обычную метку `Label1`. Нам следует зарегистрировать псевдоним метки (допустим, `Z`) и описать еще одно выражение, указывающее, куда именно следует поместить сумму: "`Z.Caption`".

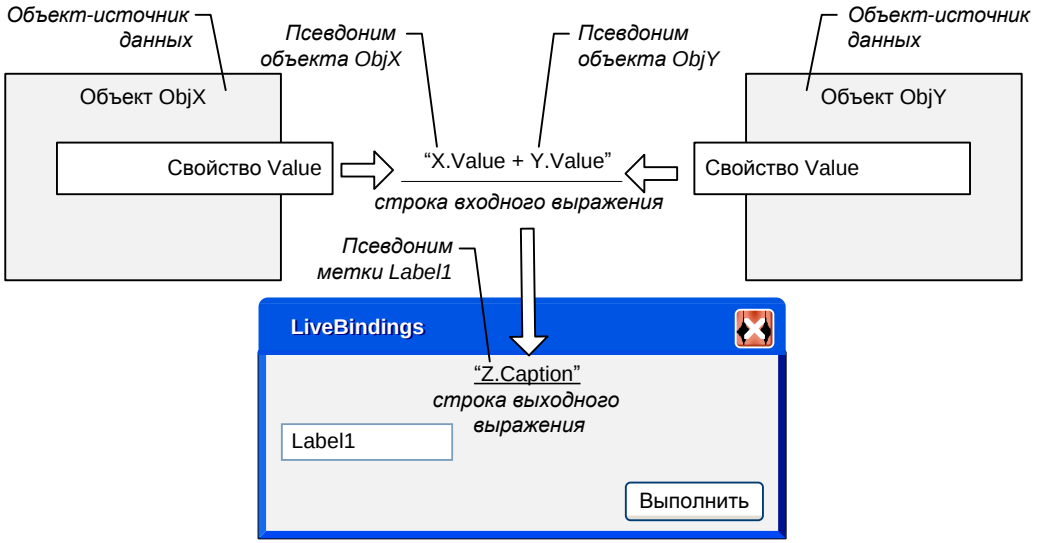


Рис. 46.5. Строка выражения LiveBindings

Программная связь, класс *TBindings*

Если логика приложения предполагает создание связи LiveBindings программным образом, то разработчику стоит познакомиться с классом `TBindings`. Сразу отметим тот факт, что `TBindings` попросту является владельцем некоторого набора методов, специализирующихся на создании живых связей. Все методы объявлены как методы класса (`class function`) и поэтому не нуждаются в физическом существовании экземпляра `TBindings`.

> ***Замечание***
>
> Основной код программной реализации технологии живого связывания сосредоточен в двух модулях: `System.Bindings.Expression` и `System.Bindings.Helper`.

Для создания простейшего выражения LiveBindings следует вызвать метод

```
class function TBindings.CreateExpression(
       const InputScopes: array of IScope; //массив источников данных
       const BindExprStr: string           //строка выражения
             ): TBindingExpression; overload;
```

Параметр `InputScopes` содержит массив ассоциаций, представляющий собой пары "объект источник данных — псевдоним объекта в выражении". Текст выражения, описывающего поведение источников данных, передается в параметр `BindExprStr`.

> ***Замечание***
>
> Еще одна особенность `TBindings` в том, что в результате действий класса на свет появляется нерассмотренный несколькими страницами ранее `TBindExpression`. Вместо этого `TBindings` оперирует с экземпляром класса — `TBindingExpression`.

Поучительной иллюстрацией работы метода создания выражения может стать пример программы, способной осуществлять расчет арифметических выражений, записанных в форме

обычной текстовой строки. Для этого нам потребуется помощь трех элементов управления (рис. 46.6):

- строки ввода `Edit1:TEdit`, которая предназначена для ввода текстового выражения;
- кнопки `Button1:TButton`, она даст команду на расчет выражения;
- метки `Label1:TLabel`, которая отобразит результат расчета.

Полагаете, что далее последует сложный пример с элементами синтаксического разбора сложных предложений? Нет! Технология LiveBindigs сделает это без посторонней помощи! На нашу долю выпадет только написать две строки кода в событии щелчка по кнопке `Button1` (листинг 46.2).

Листинг 46.2. Расчет математического выражения

```
uses System.Bindings.Helper;
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  with TBindings.CreateExpression([], Edit1.Text) do
       Label1.Caption:=Evaluate.GetValue.ToString;
end;
```

Единственная особенность нашего примера в том, что на этот раз мы даже не воспользовались услугами объекта-источника данных и оставили первый параметр метода `CreateExpression()` пустым.

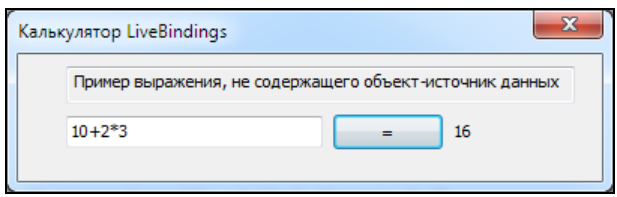


Рис. 46.6. Интерфейс приложения

Для создания связи программным образом обычно применяют метод класса

```
class function TBindings.CreateManagedBinding(
   const InputScopes: array of IScope;  //массив источников данных
   const BindExprStr: string;           //строка выражения
   const OutputScopes: array of IScope; //массив выходных значений
   const OutputExpr: string;            //выходное выражение
   const OutputConverter: IValueRefConverter;//выходное преобразование
         BindingEventRec: TBindingEventRec; //событие
         Manager: TBindingManager = nil;    //ссылка на менеджер
         Options: TCreateOptions = [coNotifyOutput]//опции
    ): TBindingExpression; overload;
```

Параметры `InputScopes` и `BindExprStr` нам уже знакомы. Параметр `OutputScopes` содержит массив пар "управляемый объект — псевдоним объекта в выходном выражении". Выходное выражение заносится в параметр `OutputExpr`. Правила преобразования данных определяют-

ся параметром OutputConverter. Шестой по счету параметр BindingEventRec предоставляет возможность подключения к созданному объекту связи обработчика события. Седьмой по счету параметр Manager требует более подробного рассмотрения.

В результате выполнения функции появившийся на свет экземпляр связи сразу передается под опеку менеджера взаимодействия (TBindingManager). Менеджер управляет всеми связями в приложении и уведомляет их обо всех изменениях с ассоциированными с этими связями объектами. В распоряжении менеджера могут состоять подчиненные менеджеры (субменеджеры), которым также рассылаются уведомления об изменениях в объектах. Если вы явным образом не укажете на экземпляр менеджера и передадите в параметр Manager неопределенный указатель nil, то связь автоматически передается под опеку глобальному менеджеру связей приложения.

> ***Замечание***
>
> Технически связь может быть создана без зависимости от менеджера взаимодействия. Для этого применяется метод класса CreateUnmanagedBinding().

Для создания ассоциации между реальным объектом Delphi и его псевдонимом в строке выражения задействуется метод

```
class function TBindings.CreateAssociationScope(
                          Assocs: array of TBindingAssociation): IScope;
```

В самом простейшем случае в единственный параметр метода — массив ассоциаций Assocs — передаются пары "объект — текстовый псевдоним". Эти пары легко формируются сервисной функцией

```
function Associate(RealObject: TObject;
                   const ScriptObject: String): TBindingAssociation;
```

После успешного формирования связи между объектами обсудим самый главный (уже знакомый по предыдущим примерам) метод-уведомление:

```
class procedure TBindings.Notify(Sender: TObject; PropName: string = '';
                        Manager: TBindingManager = nil);
```

Метод уведомляет об изменении свойства или объекта, чем стимулирует один из взаимодействующих объектов на выполнение выражения. Здесь: Sender — объект, инициирующий взаимодействие объектов; PropName — участвующее во взаимодействии свойство; Manager — необязательная ссылка на менеджер взаимодействия.

ЧАСТЬ IV

FireMonkey

Глава 47. Платформа FireMonkey
Глава 48. Приложение FireMonkey
Глава 49. Обзор компонентов для проектов HD
Глава 50. Анимация

ГЛАВА 47

Платформа FireMonkey

Совсем недавно — в начале осени 2011 года — язык программирования Delphi совершил очередной эволюционный скачок — в составе языка Delphi XE2 появилась принципиально новая возможность разработки кроссплатформенных приложений. Теперь Delphi XE2 позволяет создавать не только 32- и 64-битные приложения для Windows, но и полноценные программные продукты, предназначенные для работы под управлением Mac OS X (OS X 10.6 Snow Leopard и 10.7 Lion) и iOS (начиная с версии 4.2)! В основу кроссплатформы положена во всех отношениях уникальная библиотека FireMonkey (FMX).

Помимо кроссплатформы (что само по себе достаточное условие для того, чтобы заслужить уважение не только у многочисленных поклонников Delphi, но и у конкурентов) платформа FireMonkey обладает внушительным спектром дополнительных преимуществ. В первую очередь стоит отметить существенно возросшие графические возможности языка программирования (двух- и трехмерная деловая графика, расширенный графический интерфейс пользователя, улучшенные элементы управления). Кроме того, FMX работает в Unicode, упрощает разработку многоязыковых приложений и делает многое другое.

Сразу отметим, что появление библиотеки FMX ни в коем случае не говорит о закате VCL. Классическая библиотека компонентов, ориентированная на операционную систему Windows, по-прежнему будет совершенствоваться. Если вы разрабатываете программное обеспечение исключительно для Microsoft Windows, то стоит проявить здоровый консерватизм и создавать проекты на VCL. Основное объяснение тому в том, что библиотека VCL является более "квалифицированным специалистом" по Windows, чем универсальный FMX. Предпочтение FMX следует отдавать в тех случаях, когда проектируемое ПО вы намерены разворачивать не только на Windows.

Опорный класс *TFmxObject*

Идея библиотеки FMX основана на концепции объектно-ориентированного программирования, поэтому изучение FireMonkey следует начать с обзора опорных классов библиотеки. И здесь нас ждет первый сюрприз. Одного взгляда на иерархию наследования базового класса `TFmxObject` окажется достаточно для того, чтобы понять, что он создан не с "нуля". Впереди `TFmxObject` в цепи наследования вы обнаружите хорошо знакомые по классической библиотеке VCL классы `TObject`, `TPersistent` и, конечно же, `TComponent` (рис. 47.1).

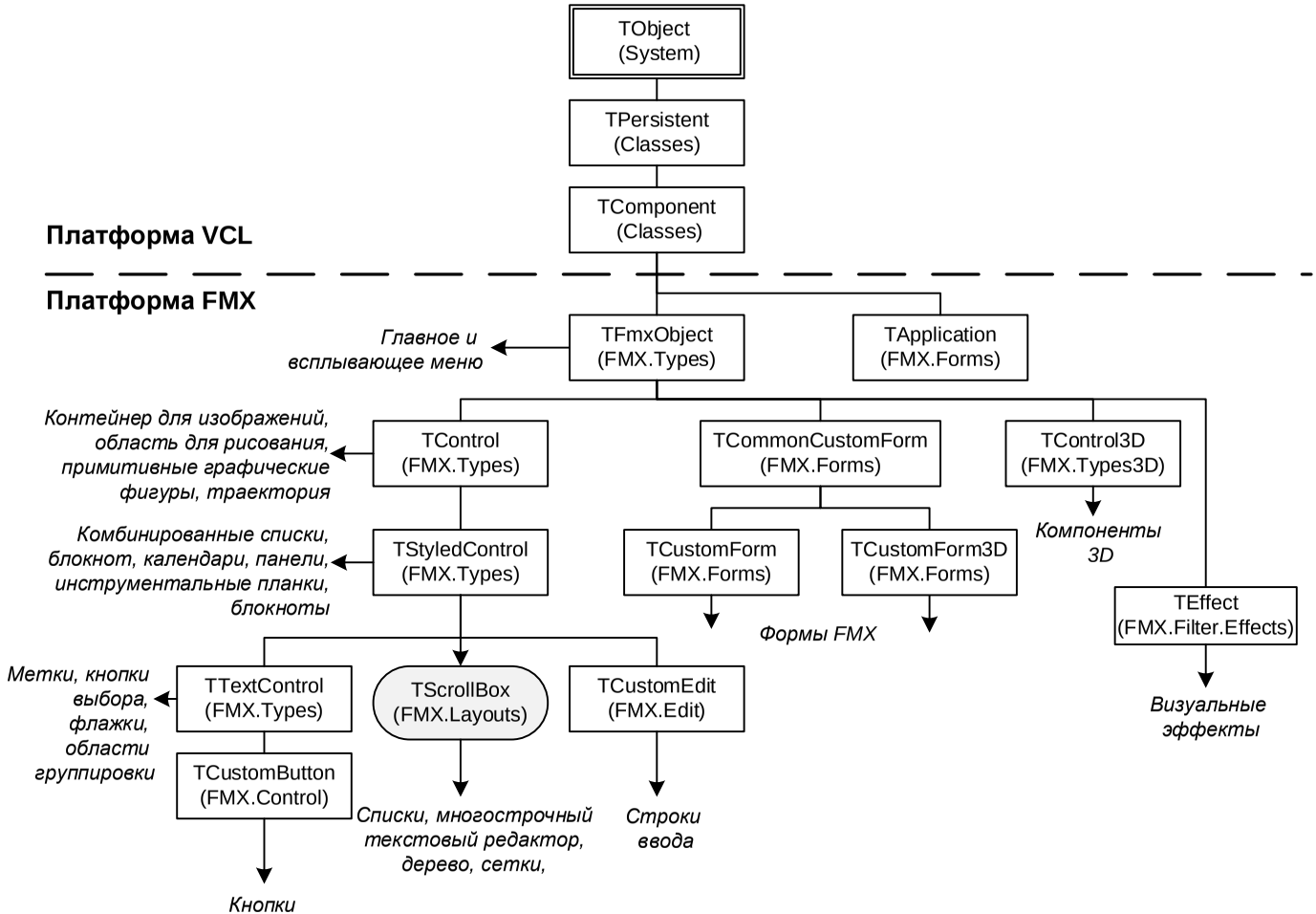


Рис. 47.1. Иерархия наследования VCL-FMX

Создание и уничтожение экземпляра класса

В джентльменском наборе опорного класса TFmxObject мы обнаружим все необходимое для обеспечения работы классического программного объекта. В первую очередь это конструктор и деструктор экземпляра класса:

```
constructor Create(AOwner: TComponent); override; //конструктор
destructor Destroy; override; //деструктор
```

В современном даже невысокой степени сложности приложении одновременно взаимодействуют десятки, а то и сотни объектов. Зачастую жизненный цикл одних объектов состоит в прямой зависимости от "жизни" или "смерти" других. Для того чтобы о факте уничтожения одного объекта узнал другой, в состав методов класса TFmxObject включена процедура

```
procedure FreeNotification(AObject: TObject); virtual;
```

Единственный параметр метода указывает на объект, которому будет автоматически направлено уведомление об уничтожении нашего компонента.

Обратная задача (вычеркивание объекта из перечня объектов, подлежащих уведомлению) решается методом

```
procedure RemoveFreeNotify(const AObject: IFreeNotification);
```

Отметим, что уведомление выбранных объектов осуществляется автоматически, для этого предназначен метод Notification(), объявленный на уровне класса TComponent.

Сохранение объекта в памяти

Очередное направление деятельности класса TFmxObject связано с манипулированием объектами FMX в памяти. В первую очередь это методы, осуществляющие загрузку/сохранение объекта в поток:

```
procedure LoadFromBinStream(const AStream: TStream);
procedure LoadFromStream(const AStream: TStream);
procedure SaveToStream(const Stream: TStream);
procedure SaveToBinStream(const AStream: TStream);
```

При желании объект в состоянии создать собственную копию:

```
function Clone(const AOwner: TComponent): TFmxObject;
```

Клон объекта передается в распоряжение владельца AOwner.

Управление дочерними объектами

В базовом классе TFmxObject предусмотрена возможность владения дочерними объектами. Доступ к принадлежащему дочернему объекту осуществляется при посредничестве свойства

```
property Children[Index: Integer]: TFmxObject;
```

Общее число подчиненных объектов предоставит свойство

```
property ChildrenCount: Integer; //только для чтения
```

В свою очередь дочерний объект знает своего владельца

```
property Parent: TFmxObject;
```

и порядковый номер в списке владельца

```
property Index: Integer; //только для чтения
```

Стоит отметить, что объект может быть вполне самостоятельным и не иметь владельца, об этом сигнализирует метод

```
function HasParent: Boolean;
```

Для добавления дочернего объекта вызываем процедуру

```
procedure AddObject(AObject: TFmxObject); virtual;
```

При необходимости дочерние объекты могут быть упорядочены, для этого вызывается метод

```
procedure Sort(Compare: TFmxObjectSortCompare);
```

Удаление дочернего объекта из списка осуществляет перегружаемый метод

```
procedure RemoveObject(AObject: TFmxObject); overload; virtual;
procedure RemoveObject(Index: Integer); overload; virtual;
```

В первой нотации производится удаление конкретного объекта `AObject`, во втором случае удаляется объект с индексом `Index`.

Объект-владелец может провести полную очистку списка дочерних объектов, для этого он обращается к методу

```
procedure DeleteChildren; virtual;
```

Отметим, что объекты не подлежат физическому уничтожению, а просто изымаются из списка.

Сопоставление дополнительных данных

В классах VCL было объявлено неиспользуемое системой свойство `Tag`, позволяющее программисту связывать с объектом целочисленное значение. В классах FMX на сопоставлении с объектом дополнительных данных специализируется целая группа свойств:

```
property TagFloat: Single;   //вещественное значение
property TagObject: TObject; //внешний объект
property TagString: string;  //текстовая строка
```

Элемент управления FMX — класс *TControl*

Все пользовательские элементы управления платформы FireMonkey берут начало от объявленного в модуле `FMX.Types` класса `TControl`. Несмотря на одинаковые имена, знак равенства между классическим `TControl` из состава VCL и классом `FMX.Types.TControl` ставить преждевременно. По сравнению со своим "коллегой" наш новый знакомый обладает существенно возросшим функционалом.

Если вы намерены досконально разобраться с классом, то сразу стоит взглянуть на его объявление:

```
type TControl = class(TFmxObject, IControl, IContainerObject,
                                  IAlignRoot, IAlignableObject)
```

Как видите, `FMX.Types.TControl` является не просто наследником опорного класса `TFmxObject`. В объявлении родоначальника всех элементов управления FireMonkey присутствуют четыре интерфейса.

- Благодаря `IControl` потомки класса `FMX.Types.TControl` приобретают право получать фокус ввода и позволяют пользователю передавать потомкам `FMX.Types.TControl` фокус ввода (например, с помощью клавиши <Tab>). Кроме того, интерфейс `IControl` обеспечит взаимодействие с клавиатурой и мышью. Напомню, что в VCL такими чертами обладали только оконные элементы управления (потомки `TWinControl`).
- Благодаря интерфейсам `IAlignRoot` и `IAlignableObject` реализовано улучшенное выравнивание элемента управления на поверхности родительского контейнера. А интерфейс `IContainerObject` в любой момент времени готов предоставить информацию о размерах родительского контейнера.

В классе `FMX.Types.TControl` принципиально изменился способ хранения местоположения элемента управления. Теперь допускается описывать позицию элемента управления в двух- и трехмерном пространствах с использованием векторных координат.

Отметим еще тот немаловажный факт, что на уровне `FMX.Types.TControl` осуществляется управление не только видимостью элемента управления, но и его прозрачностью, масштабом и вращением. Кроме того, на `FMX.Types.TControl` возложен фирменный функционал FireMonkey по контролю за прорисовкой, анимацией и другими визуальными эффектами.

Размещение и выравнивание элемента управления

Несмотря на внешнее сходство компонентов FMX и VCL, отличий между ними больше, чем сходства. Одна из особенностей элементов управления FireMonkey связана с подходом к вопросу размещения и выравнивания элемента управления на поверхности родительского контейнера. Первое новшество заключается в отказе от целых величин при описании размеров и местоположения элемента управления. Хотя высота и ширина объекта определяются свойствами с привычными названиями:

```
property Width: Single;  //ширина объекта
property Height: Single; //высота объекта
```

Но теперь свойства ориентированы на действительные числа. Такой подход существенно упрощает пересчет величин при проведении различных графических операций, поворотах, масштабировании и других преобразованиях объектов.

> ***Внимание!***
>
> При работе в двухмерном пространстве координаты объекта FMX (свойство `Position`) определяются в формате структуры `TPosition`. В трехмерном пространстве местоположение объекта описывается структурой `TPosition3D`.

Еще большие изменения постигли способ описания координат местоположения объекта. Очередное новшество FMX спрятано в недрах свойства

```
property Position: TPosition;
```

отвечающего за определение местоположения компонента в двухмерном или трехмерном пространстве. Как видите, теперь это не банальные свойства `Top` и `Left`, а интеллектуальный объект `TPosition`, определяющий позицию компонента как вектор в трехмерном пространстве (листинг 47.1).

Листинг 47.1. Описание структуры вектора

```
type TVector = record
    case Integer of
    0: (V: TVectorArray;); //type TVectorArray = array [0..2] of Single;
    1: (X: Single;   //горизонтальная ось
        Y: Single;   //вертикальная ось
        W: Single;); //ось объема
  end;
```

ВНИМАНИЕ!

В трехмерном пространстве (проекты на основе `TForm3D`) размещаемые на форме элементы позиционируются не относительно левого верхнего угла (как это было в VCL или в двухмерных проектах FireMonkey HD), а относительно своего центра.

Основные методы и свойства класса `TPosition` представлены в табл. 47.1.

Таблица 47.1. *Основные свойства и методы класса* `TPosition`

Свойство/метод	Описание
`property DefaultValue: TPointF;`	Возвращает значения координат по умолчанию — обычно (0, 0)
`property X: Single;`	Горизонтальная координата
`property Y: Single;`	Вертикальная координата
`property Point: TPointF;`	Представление координат в форме записи `TPointF`. Запись `TPointF` кроме хранения координат (`X`, `Y`) предоставляет услуги по сравнению, смещению, приращению координат
`property Vector: TVector;`	Представление координат в форме вектора
`function Empty: Boolean;`	Возвращает `true`, если координаты `X` и `Y` установлены в 0
`procedure Reflect(const Normal:TVector);`	Отражает текущий вектор перпендикулярно линии, проведенной из точки (0, 0, 0) в точку `Normal`
`property OnChange: TNotifyEvent;`	Событие, генерируемое в момент изменения координат

Выравнивание объекта

По сравнению с VCL, у объектов FMX имеется существенно расширенный набор возможностей по выравниванию элемента управления на поверхности родительского контейнера. Здесь (точно так же, как и у объектов VCL) предусмотрено свойство

```
property Align: TAlignLayout; //по умолчанию alNone
```

но вариантов выравнивания больше — целых 20 (против 6 в VCL).

```
type  TAlignLayout = (alNone, alTop, alLeft, alRight, alBottom, alMostTop,
alMostBottom, alMostLeft, alMostRight, alClient, alContents, alCenter, alVertCenter,
alHorzCenter, alHorizontal, alVertical, alScale, alFit, alFitLeft, alFitRight);
```

К вопросу выравнивания объекта имеют отношение еще два свойства:

```
property Margins: TBounds; //по умолчанию (0,0,0,0)
property Padding: TBounds; //по умолчанию (0,0,0,0)
```

Свойство `Margins` определяет отступы от границ родительского контейнера. Отступы выступают в качестве запрещенной зоны для выравниваемого дочернего объекта. Например, если вы установите свойство `Margins.Top` формы в состояние 20, то после этого все выравниваемые по верхнему краю элементы управления (`Align=alTop`) не смогут приблизиться к верхнему краю формы более чем на 20 пикселов.

Свойство `Padding` развивает идею запретных зон альтернативным способом — со стороны дочернего (выравниваемого) объекта. Свойство определяет отбивку дочернего объекта, указывая, на сколько пикселов ему разрешено приблизиться к границе своего контейнера-владельца.

Масштабирование и вращение объекта

Размеры любого потомка класса `TControl` могут быть пропорционально изменены. Коэффициент масштабирования задается в свойстве

```
property Scale: TPosition;
```

Напомним, что класс `TPosition` способен работать с координатами, заданными в форме вектора, поэтому свойство `Scale` способно управлять масштабом в трех измерениях одновременно.

Любой из потомков класса `TControl` может быть подвергнут вращению. Угол поворота в градусах (–180...+180) задается в свойстве

```
property RotationAngle: Single;
```

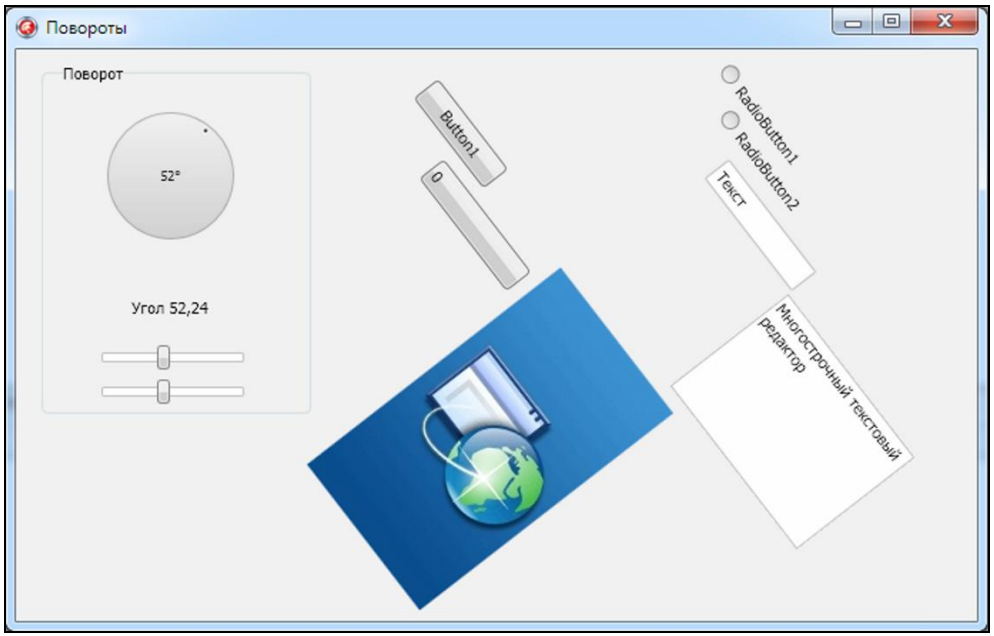


Рис. 47.2. Демонстрация возможностей по повороту элемента управления FMX

Объект поворачивается относительно центра вращения, который назначается в свойстве

```
property RotationCenter: TPosition;
```

Заметим, что точка центра вращения задается относительно клиентских координат объекта, которые нормируются к 1. По умолчанию центр находится в точке (0.5, 0.5) — это не что иное, как геометрический центр вращаемого объекта. Если мы зададим в качестве центра точку (1, 1) — центр вращения переместится в правый нижний угол объекта, (0, 0) — в левый верхний (рис. 47.2).

Для разработчиков компонентов будет полезно знать, что на уровне `TControl` объявлено свойство

```
property Skew: TPosition;
```

позволяющее деформировать объект. Указанное свойство еще активно применяется в классе `TTransform`, отвечающем за осуществление различного рода преобразований.

Видимость и прозрачность элемента управления

У всех визуальных компонентов из состава FMX сохранилась традиционное свойство

```
property Visible: Boolean;
```

делающее компонент невидимым при установке в состояние `false`. Но, как вы догадались, в FireMonkey все гораздо интереснее. Программисты Embarcadero свои компоненты наделили уникальной возможностью — постепенно становиться прозрачными. По умолчанию свойство

```
property Opacity: Single; //по умолчанию 1
```

принимает значение 1, что свидетельствует о полной непрозрачности компонента. Уменьшая значение до 0, мы добьемся требуемой степени прозрачности объекта вплоть до полной невидимости.

Обработка событий

Программисту, имеющему опыт разработки проектов в VCL, не составит никакого труда разобраться с основными обработчиками событий, описанных на уровне класса `TControl` платформы FMX. События `FMX.Types.TControl` во многом повторяют опорные события, имеющиеся в распоряжении классов `TControl` и `TWinControl` из дружеской библиотеки VCL. Основные отличия сопряжены с изменениями в позиционировании элементов управления (напомню, что FMX перешел на вещественные числа и способен описывать координатную точку в трехмерном пространстве).

Простейшие события — щелчок

В любой операционной системе, поддерживающей FMX, самыми востребованными событиями были и остаются одинарный и двойной щелчки по элементу управления.

```
property OnClick: TNotifyEvent;    //одинарный щелчок
property OnDblClick: TNotifyEvent; //двойной щелчок
```

Напомню, что инициатором щелчка могут стать левая кнопка мыши, а также нажатие клавиши <Enter> (<Пробел>) при условии, что объект управления находится в фокусе ввода.

Клавиатурные события

Для того чтобы ни один из визуальных элементов управления FMX не остался безучастным к нажатиям пользователем клавиш, на уровне `TControl` реализованы два самых важных клавиатурных события:

```
property OnKeyDown: TKeyEvent; //нажатие клавиши
property OnKeyUp: TKeyEvent;   //отпускание клавиши
```

Благодаря `OnKeyDown` и `OnKeyUp` мы получаем возможность отследить как нажатие, так и отпускание клавиши. Оба события типизированы одинаково:

```
type TKeyEvent = procedure(Sender: TObject; var Key: Word;
                  var KeyChar: WideChar; Shift: TShiftState) of object;
```

Здесь: `Sender` — ссылка на объект, вызвавший событие; `Key` — числовой код нажатой клавиши; `KeyChar` — параметр-переменная, с помощью которого можно считывать (а при необходимости и редактировать) символ нажатой клавиши; `Shift` обеспечит контроль состояния служебных клавиш.

События мыши

Разработка удобного пользовательского интерфейса современного приложения невозможна без активного использования мыши. Именно поэтому базовый класс `TControl` стал счастливым обладателем полудюжины событий, ориентированных на работу с мышью.

В первую очередь выделим события, связанные с фактом появления указателя мыши над областью элемента управления и с выходом из этой области:

```
property OnMouseEnter: TNotifyEvent; //вхождение в область
property OnMouseLeave: TNotifyEvent; //выход из области
```

Вторая категория событий позволяет отслеживать события нажатия и отпускания кнопок мыши:

```
property OnMouseDown: TMouseEvent; //нажатие кнопки
property OnMouseUp: TMouseEvent;   //отпускание кнопки
```

Перечисленные события типизируются процедурой

```
type TMouseEvent = procedure(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState; X, Y: Single) of object;
```

Благодаря параметру `Button` мы получаем возможность узнать, какая именно кнопка мыши была задействована в операции. Параметр `Shift` позволяет отследить нажатия служебных клавиш. Параметры `X` и `Y` информируют о координатах мыши.

При перемещении указателя мыши стоит обратить внимание на событие

```
property OnMouseMove: TMouseMoveEvent;
type TMouseMoveEvent = procedure(Sender: TObject;
                       Shift: TShiftState; X, Y: Single) of object;
```

позволяющее контролировать местоположение указателя. Назначения параметров нам уже известны, поэтому не станем повторяться. И сразу перейдем к заключительному событию мыши, описанному в рамках класса `TControl`.

Контроль вращения колесика мыши осуществляет единоличный представитель — событие

```
property OnMouseWheel: TMouseWheelEvent;
type TMouseWheelEvent = procedure(Sender: TObject; Shift: TShiftState;
                    WheelDelta: Integer; var Handled: Boolean) of object;
```

Благодаря параметру `WheelDelta` мы получаем сведения о направлении вращения колесика: отрицательное значение — вниз, положительное — вверх. Передав значение `true` в переменную `Handled`, мы уведомим систему, что программная логика отработала полностью и про событие можно забыть, в противном случае обработка события продолжится за счет вызова обработчика события по умолчанию.

События получения и утраты фокуса ввода

Среди компонентов FireMonkey нет строгого разграничения на оконные и графические элементы управления, подобного тому, которое существует у потомков классов `TWinControl` и `TGraphicControl` платформы VCL. Строго говоря, все визуальные компоненты FMX одновременно обладают чертами оконных и графических элементов управления. Одно из подтверждений тому — способность всех наследников `FMX.Types.TControl` реагировать на получение фокуса ввода.

Наряду с традиционной (имеющейся и у компонентов VCL) реакцией на получение и потерю фокуса ввода компонентом, осуществляемой событиями

```
property OnEnter: TNotifyEvent; //получение фокуса ввода
property OnExit: TNotifyEvent;  //потеря фокуса ввода
```

в FMX предусмотрено событие, позволяющее программисту запретить объекту реагировать на фокус ввода:

```
property OnCanFocus: TCanFocusEvent;
type TCanFocusEvent = procedure(Sender: TObject; var ACanFocus: Boolean) of object;
```

Разрешение или запрет выдается с помощью параметра-переменной `ACanFocus`.

Событие изменения размера

Изменение геометрических размеров любого потомка класса `TControl` незамедлительно приводит к генерации события

```
property OnResize: TNotifyEvent;
```

Отмечу, что по сравнению с событиями контроля за размерами объекта в VCL новая библиотека несколько отстала, но ведь в Delphi XE2 состоялся только дебют FireMonkey.

События перетаскивания drag and drop

Платформа FireMonkey поддерживает очень популярный интерфейс перетаскивания объектов мышью — в операции участвуют два объекта: компонент — источник данных и компонент — получатель данных.

Операцию перетаскивания инициирует компонент-источник. В его распоряжении имеется свойство, определяющее порядок инициализации процесса.

```
property DragMode: TDragMode; //по умолчанию dmManual
```

Если свойство установить в состояние `dmAutomatic`, то любое перемещение над компонентом-источником указателя мыши (у которой удерживается в нажатом состоянии левая или правая кнопка) автоматически активирует механизм drag and drop. В ручном режиме перетаскивания (`dmManual`) команду на перетаскивание должен дать программист.

В минимальной нотации для осуществления перетаскивания достаточно услуг всего двух событий, генерируемых на стороне компонента-приемника: `OnDragOver()` и `OnDragDrop()`. Первое из событий осуществляет проверку приемлемости поступающих данных и дает согласие на получение данных от компонента-источника.

```
property OnDragOver: TDragOverEvent;
type TDragOverEvent = procedure(Sender: TObject; const Data: TDragObject;
                      const Point: TPointF; var Accept: Boolean) of object;
```

Здесь `Sender` — компонент-получатель, вызвавший событие. Ссылка на объект, выступивший инициатором операции перетаскивания, находится в параметре `Data`. Это запись, состоящая из трех полей, ключевое из которых `Source` — именно в нем находятся все сведения об объекте-источнике операции drag and drop (листинг 47.2). Координаты указателя мыши можно прочитать в параметре `Point`. Свое согласие на получение данных компонент-получатель дает, передав в параметр `Accept` значение `true`.

Листинг 47.2. Объявление записи `TDragObject`

```
type TDragObject = record
    Source: TObject;         //объект-источник операции
    Files: array of string; //массив строк
    Data: Variant;           //дополнительные данные
  end;
```

Второе обязательное событие механизма drag and drop отвечает за обработку поступивших данных. Событие генерируется в момент отпускания кнопки мыши над компонентом-приемником.

```
property OnDragDrop: TDragDropEvent;
type TDragDropEvent = procedure(Sender: TObject;
                const Data: TDragObject; const Point: TPointF) of object;
```

Параметры события вам уже знакомы, поэтому в отдельных комментариях не нуждаются.

Для подтверждения теории на практике создайте новое приложение FireMonkey и разместите на форме произвольное количество компонентов `TImage`, специализирующихся на хранении и отображении рисунков. У всех компонентов установите свойство `DragMode` в автоматический режим. В любой из компонентов `TImage` загрузите произвольный рисунок.

Листинг 47.3 содержит код, общий для всех компонентов. Здесь описаны оба рассмотренных ранее обработчика событий.

Листинг 47.3. Объявление записи `TDragObject`

```
procedure TForm1.Image1DragOver(Sender: TObject; const Data: TDragObject;
  const Point: TPointF; var Accept: Boolean);
begin
  {проверка допустимости операции}
  Accept:=(Data.Source is TImage) and (Data.Source<>Sender) and
        (TImage(Data.Source).Bitmap.IsEmpty=false);
end;

procedure TForm1.Image1DragDrop(Sender: TObject; const Data: TDragObject;
  const Point: TPointF);
```

```
begin
  TImage(Sender).Bitmap.Assign(TImage(Data.Source).Bitmap); {передаем
                                                    картинку в приемник}
  TImage(Data.Source).Bitmap.Destroy; {удаляем исходную картинку
                                       у источника}
  TImage(Data.Source).Bitmap:=TBitmap.Create(1,1);{создаем новый пустой
                                                    Bitmap}
end;
```

Завершая разговор о drag and drop, отметим о существовании еще трех событий, имеющих отношение к перетаскиванию и решающих вспомогательные задачи. В момент появления указателя мыши над компонентом, принимающим участие в процессе перетаскивания, генерируется событие

```
property OnDragEnter: TDragEnterEvent;
```

В момент ухода указателя мыши вызывается еще одно событие

```
property OnDragLeave: TNotifyEvent;
```

О завершении процедуры перетаскивания сигнализирует событие

```
property OnDragEnd: TNotifyEvent;
```

Особенности прорисовки элемента управления

Визитной карточкой FireMonkey являются выдающиеся графические возможности. Немудрено, что каждый потомок класса `TControl` обладает способностью вывода своего изображения в любом подходящем для этого месте. Процесс вывода инициируется методом

```
procedure PaintTo(const ACanvas: TCanvas;
                  const ARect: TRectF;
                  const AParent: TFmxObject = nil);
```

Здесь, помимо ссылки на холст `ACanvas`, на поверхности которого будет осуществлен вывод, следует указать прямоугольную область `ARect`, ограничивающую участок вывода (листинг 47.4).

Листинг 47.4. Вывод изображения элемента управления на форме `Form1`

```
Form1.Canvas.BeginScene;
Button1.PaintTo(Form1.Canvas, RectF(0,0,Button1.Width * 2,Button1.Height*2));
Form1.Canvas.EndScene;
```

Еще одна особенность класса — наличие двух событий, контролирующих вывод на экран изображения элемента управления:

```
property OnPainting: TOnPaintEvent;
property OnPaint: TOnPaintEvent;
```

Оба события типизированы одинаково:

```
type TOnPaintEvent = procedure(Sender: TObject; Canvas: TCanvas;
                               const ARect: TRectF) of object;
```

Благодаря параметрам события, мы сможем получить доступ к холсту `Canvas` и узнать координаты прямоугольной области `ARect`.

ГЛАВА 48

Приложение FireMonkey

При проектировании компонентов FMX специалисты Embarcadero постарались сохранить привычные для нас (по классической библиотеке визуальных компонентов VCL) названия, функциональное назначение и, по возможности, набор свойств и методов новых классов. Поэтому вы вряд ли удивитесь, узнав, что приложение FMX описывается классом `TApplication`, форма HD (High Definition), специализирующаяся на работе в двухмерной графике — `TForm`, а трехмерная форма — `TForm3D`. Но на этот раз все перечисленные классы описаны в другом программном модуле — модуле `FMX.Forms`.

Выбор целевой платформы для проекта

Раз основная заслуга FireMonkey заключается в поддержке не только Windows, но и Mac OS X и iOS, то изучение приложения FMX начнем с определения целевой платформы для реализации приложения.

Создайте новый проект FMX. Для этого воспользуйтесь пунктом меню **File | New | FireMonkey HD Application**.

После появления на свет нового проекта FMX обратитесь к окну менеджера проекта (рис. 48.1). В дереве менеджера проекта найдите узел **Target Platforms** и, воспользовавшись услугами контекстного меню узла, добавьте интересующую вас платформу (32-разрядная Windows, 64-разрядная Windows или OS X). В результате у узла **Target Platforms** появится дочерний элемент с названием вновь добавленной платформы.

Выпуск приложения для 32- и 64-разрядной Windows не вызывает никаких затруднений. Программист просто указывает предпочтительную платформу и нажимает клавишу <F9>. Если же вы планируете создать релиз для Mac OS X, то придется еще немного потрудиться:

1. Сначала полностью отладьте приложение под управлением Windows. Это исключит многие проблемы при создании исполняемого бинарного кода под Mac OS X и в целом ускорит работу над выпуском релиза.
2. Соедините в сети два компьютера. На первом должна быть установлена операционная система Windows и развернуто ваше программное обеспечение Delphi XE2. Второй компьютер должен работать под управлением операционной системы Mac OS X (OS X 10.6 Snow Leopard или 10.7 Lion).
3. Обращаемся к компьютеру с Delphi XE2. Найдите в каталоге C:\Program Files\Embarcadero\RAD Studio\9.0\PAServer (или C:\Program Files (x86)\Embarcadero\RAD Studio\9.0\PAServer, если вы работаете в 64-разрядной Windows) архив setup_paserver.zip.

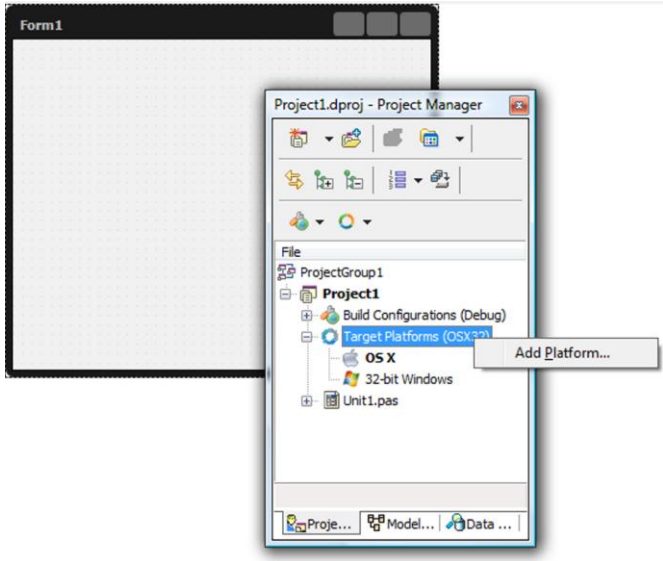


Рис. 48.1. Выбор целевой платформы для проекта FMX

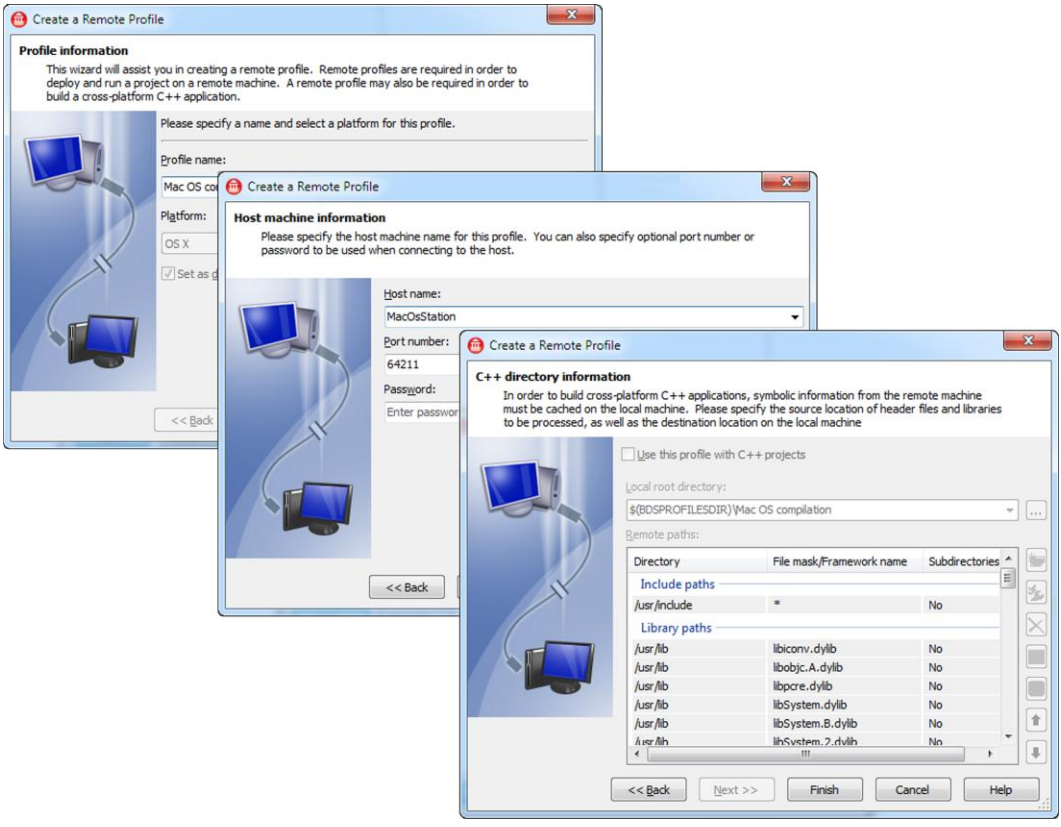


Рис. 48.2. Мастер создания удаленного профиля

4. Распакуйте архив setup_paserver.zip на рабочей станции с Mac OS X и разверните программное обеспечение **Platform Assistant**.
5. Запустите в терминальном окне компьютера с Mac OS X программное обеспечение PAServer.
6. Возвращаемся к компьютеру с Windows. В менеджере проектов Delphi XE2 создайте целевую платформу (**Target Platforms**) OS X и, дважды щелкнув по узлу, сделайте ее активной (см. рис. 48.1).
7. Щелкните правой кнопкой мыши по узлу **OS X** и выберите пункт меню **Assign Remote Profile**. Воспользовавшись услугами мастера, создайте профиль (рис. 48.2).
8. Вам осталось нажать клавишу <F9>...

> ***ВНИМАНИЕ!***
>
> Подробные инструкции по созданию бинарного кода для Mac OS X вы найдете в Интернете на страничке **http://docwiki.embarcadero.com/RADStudio/en/Installing_and_Running_the_Platform_Assistant_on_the_Target_Platform**.

Приложение *FMX.Forms.TApplication*

При знакомстве с классами FireMonkey любой программист станет сравнивать их с аналогами из VCL. На мой взгляд, в сравнении со старым `TApplication` класс `FMX.Forms.TApplication` несколько проигрывает. Впрочем, это объясняется объективной причиной. Нацеленная исключительно на Windows библиотека VCL позволила разработчикам Delphi спроектировать исключительно качественный специализированный класс, наделенный внушительным набором сервисных возможностей. В свою очередь, при работе над универсальным приложением, способным одинаково хорошо трудиться как под управлением Windows, так и под контролем Mac OS X, специалисты Embarcadero были связаны обязательством — одновременно удовлетворить требования абсолютно непохожих операционных систем. Так что по объективным причинам `FMX.Forms.TApplication` — это своего рода компромисс между Windows и Mac OS X.

Приложение `FMX.Forms.TApplication` несколько выбивается из стройных рядов классов FireMonkey. Несмотря на то, что в полном имени класса присутствует приставка "FMX" (уведомляющая программиста, что он имеет дело с кроссплатформенной библиотекой), объект-приложение строится на основе традиционного класса VCL `TComponent` (см. рис. 47.1).

Для организации работы с приложением не стоит вызывать его конструктор. Объект приложения создается автоматически. Чтобы в этом убедиться, выберите пункт меню **Project | View Source**, и в редакторе кода откроется головной модуль проекта FireMonkey (листинг 48.1).

Листинг 48.1. Головной модуль проекта приложения FireMonkey

```
program Project1;

uses FMX.Forms,
     Unit1 in 'Unit1.pas' {Form1};

{$R *.res}
```

```
begin
  Application.Initialize; //инициализация
  Application.CreateForm(TForm1, Form1); //создание главной формы
  Application.Run; //старт приложения
end.
```

Как видно из строк кода листинга, доступ к экземпляру приложения предоставляет глобальная (объявленная в модуле FMX.Forms) переменная Application. В головном модуле проекта приложение проходит инициализацию, создается главная форма и производится старт программного продукта.

Приложение обязано знать свою главную форму. Ссылка на эту форму находится в свойстве

```
property MainForm: TCommonCustomForm;
```

Подавляющее большинство элементов управления, составляющих пользовательский интерфейс управления приложения FireMonkey, может обладать единим стилистическим оформлением. Централизованное управление стилем обеспечивает приложение. Заранее подготовленные стили вы обнаружите в каталоге C:\Users\Public\Documents\RAD Studio\9.0\Styles в файлах с расширением имени style. За подключение файла стиля к приложению отвечает свойство

```
property StyleFileName: String;
```

Подключение файла с описанием стиля изменит внешний вид всех элементов управления.

Для экстренного завершения приложения можно воспользоваться методом

```
procedure Terminate;
```

Но следует понимать, что это слишком прямолинейный метод, поэтому им не стоит злоупотреблять.

Приложение FMX не в состоянии похвастаться богатым перечнем обработчиков событий. В минимальный джентльменский набор вошло событие, позволяющее осуществить централизованную обработку исключительных ситуаций:

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

Кроме того, имеется полезное во всех отношениях событие

```
property OnIdle: TIdleEvent;
type TIdleEvent=procedure(Sender: TObject; var Done: Boolean) of object;
```

позволяющее программисту вызывать код фонового режима в момент простоя приложения. На этом список событий исчерпывается.

Общие черты форм HD и 3D

Приложения FireMonkey могут использовать две разновидности форм: форму FMX.Forms.TForm, предназначенную для построения деловых приложений с высококачественным двухмерным визуальным интерфейсом, и форму FMX.Forms.TForm3D, реализующую всю мощь трехмерной графики.

Оба класса форм зиждутся на фундаменте родительского класса

```
type TCommonCustomForm = class(TFmxObject, IRoot,
                               IContainerObject, IAlignRoot)
```

Помимо уже знакомого нам класса `TFmxObject` здесь упоминаются три интерфейса. Интерфейс высокоуровневых контейнеров `IRoot` оказывает форме всестороннюю поддержку по доступу к активным объектам и элементам управления, находящимся в фокусе ввода. Интерфейсы `IContainerObject` и `IAlignRoot` упрощают выравнивание объектов на поверхности контейнера.

На уровне `TCommonCustomForm` класса заложен хорошо знакомый нам функционал:

- объявлены базовые конструкторы и деструктор формы;
- осуществляется управление размерами и местоположением формы;
- реализуется отображение формы (в том числе в модальном режиме) и скрытие формы с экрана.

Форма HD *FMX.Forms.TForm*

Форма HD (High Definition) во многом похожа на обычную форму VCL, поэтому если вы имеете хотя бы небольшой опыт программирования в предыдущих версиях Delphi, то быстро разберетесь со всеми нововведениями. Ключевых изменений всего три.

Во-первых, при позиционировании и определении размеров форма HD окончательно перешла на вещественные числа.

Во-вторых, у формы `FMX.Forms.TForm` принципиально изменен порядок работы с графикой. Теперь в своей работе форма HD не задействует механизм графического вывода Windows GDI. Вместо этого новая форма нацелена на работу с DirectX (если приложение выполняется под управлением Windows) или Quartz 2D (для приложения Mac OS X). Доступ к универсальному графическому механизму предоставляет интерфейс `IScene`. Вполне естественно, что смена графического "движка" привела к кардинальным переменам в недрах класса-холста `TCanvas`.

В-третьих, по сравнению с формой VCL у новой формы FMX несколько сокращен набор методов и обработчиков событий. Последнее несколько огорчает. Особенно неудобно то, что у формы нет событий, обеспечивающих реакцию на мышь и клавиатуру... Однако это не смертельно. На странице **Layouts** палитры компонентов FMX вы найдете несколько компонентов-контейнеров (`TGridLayout`, `TScrollBox` и т. д.), предоставляющих свою поверхность для размещения визуальных элементов управления. Перечисленные компоненты (в качестве бонуса) способны обслуживать все события мыши.

> ***Внимание!***
>
> Если программная логика вашего приложения требует "научить" форму HD реагировать на мышь, то стоит воспользоваться помощью компонентов со страницы **Layouts**.

Стили оформления формы, компонент *TStyleBook*

На странице **Standard** палитры компонентов FMX находится компонент `TStyleBook`, позволяющий осуществить индивидуальную настройку стиля оформления формы. Двойной щелчок по компоненту или обращение к свойству

```
property Resource: TStrings;
```

активирует редактор стиля. В самом простейшем случае можно воспользоваться уже подготовленными стилями оформления. Для этого достаточно в редакторе щелкнуть по кнопке

Load... и загрузить один из файлов с расширением имени style из каталога C:\Users\Public\Documents\RAD Studio\9.0\Styles. При желании вы можете внести правки в правила оформления того или иного типа элемента управления, а затем (удовлетворив свой дизайнерский порыв) нажать кнопку **Apply and Close**. В результате редактор закроется и изменения стиля применятся к компоненту `TStyleBook`.

> ***Замечание***
>
> Вместе с Delphi XE2 поставляется несколько заранее подготовленных стилей оформления формы. По умолчанию файлы стилей расположены в каталоге C:\Users\Public\Documents\RAD Studio\9.0\Styles.

Теперь следует подключить компонент стилей `TStyleBook` к форме. Для этого достаточно обратиться к свойству формы

```
property StyleBook: TStyleBook;
```

> ***Внимание!***
>
> В одном и том же приложении FireMonkey допускается раздельная настройка стилей оформления для каждой из форм проекта.

Трехмерная форма *FMX.Forms.TForm3D*

Трехмерная форма FireMonkey — это без всякого преувеличения инновационный класс, позволивший нам создавать управляемые 3D-сцены несколькими щелчками мыши. Код реализации трехмерной формы сосредоточен в рамках класса `TCustomForm3D` (модуля `FMX.Forms`). Объявление класса

```
type TCustomForm3D = class(TCommonCustomForm, IViewport3D)
```

содержит очень важный интерфейс `IViewport3D`, несущий ответственность за освещение трехмерной сцены и позволяющий управлять камерой (точкой наблюдения за сценой).

Форма `TForm3D` призвана работать в трехмерных прямоугольных координатах, начала которых находятся в центре клиентской области формы. Ось абсцисс *x* направлена из точки (0, 0, 0) вправо, ось ординат *y* — вниз, виртуальная ось аппликат *z* уходит в глубину формы (рис. 48.3).

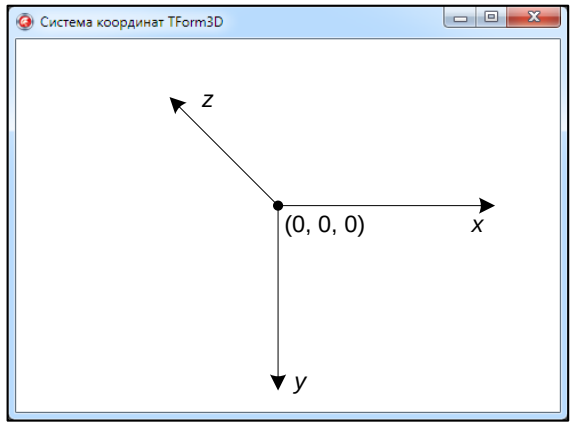


Рис. 48.3. Система координат `TForm3D`

> ***ВНИМАНИЕ!***
>
> Размещая на трехмерной форме 3D-элементы управления, помните, что эти элементы позиционируются не относительно своего левого верхнего угла (как это было в VCL или в двухмерных проектах FireMonkey), а относительно своего центра (свойство `Position`).

Еще одной важной особенностью 3D-формы является то, что графический вывод осуществляется с помощью трехмерного контекста графического устройства `FMX.Types3D.TContext3D`. Доступ к контексту предоставляет свойство

```
property Context: TContext3D;
```

Процесс графического вывода сопровождается генерацией события

```
property OnRender: TRenderEvent;
type TRenderEvent =
    procedure(Sender: TObject; Context: TContext3D) of object;
```

В числе ключевых параметров события вы вновь обнаружите контекст `Context`.

> ***ВНИМАНИЕ!***
>
> Для построения трехмерных сцен вместо прямого обращения к контексту `TContext3D` следует воспользоваться услугами компонентов 3D (страницы палитры компонентов **3D Scene**, **3D Shapes** и **3D Layers**).

В простейшем случае цвет заливки формы назначается свойством

```
property Color: TAlphaColor;
```

Для просмотра трехмерных сцен форма инкапсулирует камеру (объект `TCamera`). Камера — это не что иное, как удобная абстракция наблюдателя. Выбрав ту или иную позицию наблюдения, вы сможете взглянуть на одну и ту же сцену с разных ракурсов. Интегрированная в состав формы камера установлена прямо перед сценой. По умолчанию камера включена, для ее отключения установите в `false` свойство

```
property UsingDesignCamera: Boolean;//по умолчанию true
```

> ***ЗАМЕЧАНИЕ***
>
> Если вас не устраивает камера по умолчанию, то обратитесь к странице палитры компонентов **3D Scene**. Здесь вы найдете самостоятельный компонент `TCamera` и ряд других компонентов, предназначенных для дизайна трехмерных сцен.

Трехмерная форма обладает весьма скромным набором обработчиков событий, в частности форма не в состоянии реагировать на события мыши и клавиатуры. Если логика программы предполагает организацию взаимодействия с мышью и клавиатурой, на помощь форме следует призвать трехмерный слой, например компонент `TLayer3D` (палитра **3D Layers**).

Компонент `TLayer3D` поможет нам провести небольшое исследование трехмерной координатной системы. Для этого создаете новый проект 3D и разместите на главной форме:

- способный реагировать на мышь слой `Layer3D1:TLayer3D`. Для чистоты эксперимента убедитесь, что координаты слоя обнулены (свойство `Position= (0,0,0)`);
- в любом месте формы разместите два текстовых слоя `TTextLayer3D`.

Теперь воспользуемся обработчиком события `OnMouseMove()` компонента `Layer3D1` (листинг 48.2).

Листинг 48.2. Анализ координат указателя мыши на поверхности слоя TLayer3D

```
procedure TForm1.Layer3D1MouseMove(Sender: TObject; Shift: TShiftState;
                                   X, Y: Single; RayPos, RayDir: TVector3D);
begin
  Form1.Caption:=Format('2D координаты: (X=%.2f,Y=%.2f)',[X,Y]);
  TextLayer3D1.Text:=Format('RayPos.X=%f RayPos.Y=%f
                    RayPos.Z=%f',[RayPos.X,RayPos.Y,RayPos.Z]);
  TextLayer3D2.Text:=Format('RayDir.X=%f RayDir.Y=%f
                    RayDir.Z=%f',[RayDir.X,RayDir.Y,RayDir.Z]);
end;
```

Перемещая мышь над поверхностью компонента Layer3D1, вы увидите, каким образом работает координатная система трехмерной формы. В ходе экспериментов обязательно измените положение и углы наклона слоя Layer3D1.

> ***ЗАМЕЧАНИЕ***
>
> Если в трехмерном проекте вам необходима помощь двухмерных элементов управления, то их следует размещать на компоненте TLayer3D.

Пример 3D-проекта

Создайте новый проект 3D. Для этого следует воспользоваться пунктом меню **File | New | Other** и в открывшемся диалоге **New Items** найти значок **FireMonkey 3D Application**. Сохраните проект под любым именем и перейдите к главной форме приложения.

Сразу раскрою секрет нашего примера — мы создадим модель вымышленной планетарной системы. Для этого разместите на главной форме следующие компоненты (рис. 48.4):

- сфера Sphere1:TSphere возьмет на себя функции единственной звезды, вокруг которой станут двигаться планеты. Расположите сферу точно в центре формы (свойство Position=(0,0,0)). Обратившись к свойству Material, измените стиль заливки FillMode на fmWireframe — это позволит наблюдателю заглянуть за сферу. Воспользовавшись свойствами Width, Height и Depth, придайте светилу достойные размеры (например, 4, 4, 4);
- для того чтобы наша звезда смогла светиться, нам понадобится источник света Light:TLight, который вы найдете на странице **3D Scene**. Поместите компонент на форму и присоедините к Sphere1 (в результате его положение Position совпадет с положением звезды). Переключите свойство LightType в состояние ltPoint, это заставит источник света испускать свои "фотоны" во всех направлениях;
- разместите на форме несколько сфер-планет TSphere. В примере я ограничился тремя компонентами, но вашу фантазию я ограничивать не собираюсь;
- компонент таймер Timer1:TTimer возьмет в свои руки управление процессом движения планет. Установите интервал отсчета равным 100—300 мс.

Задача планет — вращаться по эллиптическим орбитам вокруг звезды. Чтобы вы не утруждали себя поисками уравнения эллипса в учебниках геометрии или астрономии, сразу предложу вашему вниманию скромное параметрическое выражение:

$$\begin{cases} x = a\cos t, \\ y = b\sin t, \end{cases}$$
где $0 \le t \le 2\pi$.

Здесь *a* и *b* описывают большую и малую полуоси эллипса.

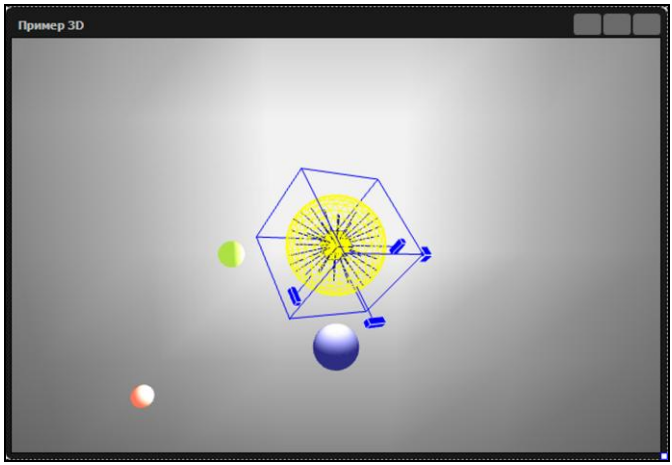


Рис. 48.4. Макет формы 3D с планетарной системой

Осталось дело за малым — изучить листинг 48.3, в котором содержится исходный код программы, моделирующей планетарную систему из звезды и трех планет.

Листинг 48.3. Исходный код программы планетарной системы

```
var Form1: TForm1;
    t1,t2,t3:single;

const a1=3;  b1=4;  //полуоси эллиптических орбит
      a2=4;  b2=8;
      a3=10; b3=5;
implementation

{$R *.fmx}

procedure TForm1.Form3DCreate(Sender: TObject);
begin
  t1:=0;    //инициализация исходных значений
  t2:=pi/2;
  t3:=pi/4;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Sphere2.Position.X:=a1*cos(t1); //орбита планеты 1
  Sphere2.Position.Y:=b1*sin(t1);
```

```
  Sphere3.Position.Z:=a2*cos(t2); //орбита планеты 2
  Sphere3.Position.Y:=b2*sin(t2);

  Sphere4.Position.X:=a3*cos(t3); //орбита планеты 3
  Sphere4.Position.Z:=b3*sin(t3);

  t1:=t1+0.1;  t2:=t2+0.15; t3:=t3-0.25;
end;
```

Основные расчеты осуществляются в рамках события `OnTimer()`. Самостоятельно подберите интервал срабатывания таймера, чтобы получить реалистичную картину перемещения планет.

ГЛАВА 49

Обзор компонентов для проектов HD

С точки зрения функциональных возможностей большинство элементов управления FMX представляет собой аналоги своих "коллег" из VCL с той лишь разницей, что в иерархии классов FireMonkey в принципе отсутствует существующее в VCL разделение на оконные и графические элементы управления. В FMX любой потомок класса `TControl` (программный модуль `FMX.Controls`), строго говоря, одновременно и графический, и оконный. Благодаря схожести функциональных возможностей элементов управления FMX и VCL и преемственности названий свойств, методов и обработчиков событий программисту, хорошо владеющему VCL, не составит никакого труда освоить новую библиотеку компонентов. Поэтому я предлагаю заострить наше внимание только на некоторых компонентах-новинках, у которых нет аналогов в VCL, и на элементах управления, претерпевших кардинальные изменения.

Панель-выноска *TCalloutPanel*

Как и следует из названия, компонент `TCalloutPanel` играет роль панели-выноски, которую можно использовать в качестве указателя или пояснительного блока к какому-то элементу пользовательского интерфейса приложения (рис. 49.1).

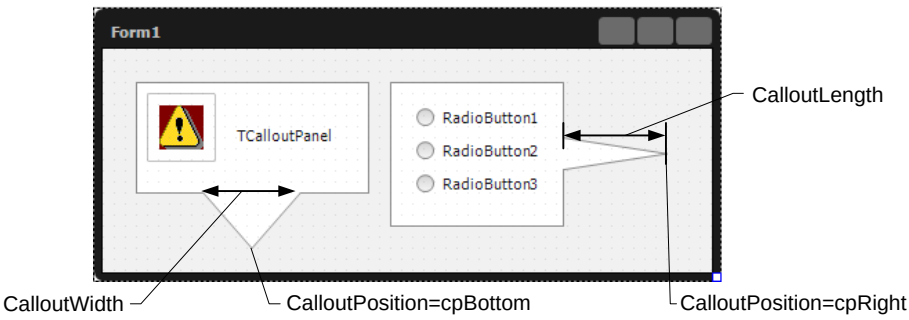


Рис. 49.1. Внешний вид элемента управления `TExpander`

Указатель может исходить из одной из сторон панели, для выбора стороны следует воспользоваться свойством

```
property CalloutPosition: TCalloutPosition;
type TCalloutPosition = (cpTop, cpLeft, cpBottom, cpRight);
```

Длина и ширина указателя зависит от состояния свойств

```
property CalloutLength: Single;
property CalloutWidth: Single;
```

Кроме того, при поддержке свойства

```
property CalloutOffset: Single; //по умолчанию 0
```

указатель может быть смещен относительно центра панели.

Разворачивающаяся панель *TExpander*

Компонент `TExpander` окажется весьма полезным в проектах с пользовательским интерфейсом, насыщенным большим числом элементов управления. В подобных проектах разработчик гарантированно сталкивается с дилеммой нехватки рабочего пространства. И здесь, как нельзя кстати, окажется разворачивающаяся панель `TExpander`. Как и любая другая панель, `TExpander` предназначена для исполнения роли контейнера для других элементов управления, но при этом компонент способен сворачиваться, тем самым скрывая ненужные в данный момент элементы интерфейса (рис. 49.2).

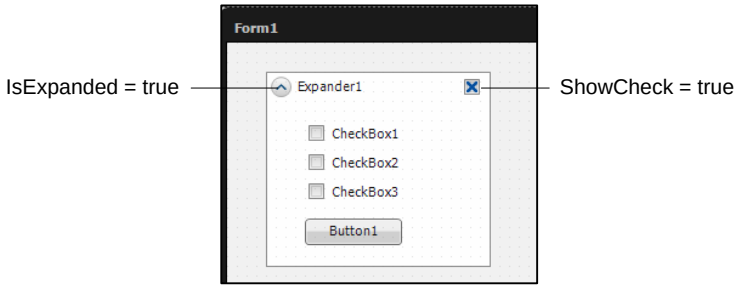


Рис. 49.2. Внешний вид элемента управления `TExpander`

Наиболее важное свойство компонента

```
property IsExpanded: Boolean; //по умолчанию true
```

позволяет сворачивать (состояние `false`) и разворачивать панель.

Текстовая надпись в заголовке панели определяется свойством

```
property Text: string;
```

Благодаря свойству

```
property IsChecked: Boolean;//по умолчанию true
```

все принадлежащие панели элементы управления можно сделать пассивными (состояние `false`).

По умолчанию управление свойством `IsChecked` предоставляется только из кода программы. Однако, воспользовавшись свойством

```
property ShowCheck: Boolean; //по умолчанию false
```

их можно вывести на экран. В момент изменения состояния панели генерируется событие

```
property OnCheckChange: TNotifyEvent;
```

Компонент *TArcDial*

Основная ассоциация, которая возникает при первой встрече с элементом управления `TArcDial`, такова — компонент напоминает регулятор громкости в аудиоаппаратуре. И действительно, `TArcDial` позволяет создавать пользовательский интерфейс, в котором управление значением осуществляется за счет поворота "регулятора".

Основное свойство элемента управления

```
property Value: Single; //от 0 до 359.99 градусов
```

определяет угол поворота регулятора в градусах. При любом изменении значения свойства генерируется событие

```
property OnChange: TNotifyEvent;
```

Активировав свойство

```
property ShowValue: Boolean; //по умолчанию false
```

вы заставите компонент отображать числовое значение, содержащееся в свойстве `Value`.

Компонент *TNumberBox*

Компонент `TNumberBox` является ближайшим родственником обычной строки ввода и специализируется на предоставлении пользователю услуг ввода целых или вещественных чисел. Для ввода значения достаточно передать элементу управления фокус ввода и несколько раз нажать клавишу управления курсором.

Тип обслуживаемых данных определяется свойством

```
property ValueType: TNumValueType;
type TNumValueType = (vtInteger, vtFloat);
```

Текущее значение хранится в свойстве

```
property Value: Single;
```

Диапазон допустимых значений ограничивается свойствами

```
property Min: Single;
property Max: Single;
```

При вводе вещественных чисел точность данных определяется свойством

```
property DecimalDigits: Integer; //по умолчанию 2
```

Шаг приращения величины `Value` зависит от состояния свойств

```
property HorzIncrement: Single;
property VertIncrement: Single;
```

Компонент *TComboTrackBar*

Что будет, если объединить идеи комбинированного списка и ползунка? Правильно — получится компонент `TComboTrackBar`. Элемент управления позволяет пользователю вводить данные как вручную (с помощью цифровой клавиатуры), так и при помощи ползунка, "выпадающего" из компонента (рис. 49.3).

Основное свойство компонента:

```
property Value: Single;
```

Здесь хранится установленное пользователем значение. Допустимый диапазон значений задается свойствами

```
property Min: Single;
property Max: Single;
```

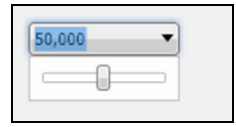


Рис. 49.3. Внешний вид элемента управления `TExpander`

При желании программист сможет получить доступ к бегунку. Для этого в коде программы стоит упомянуть свойство

```
property TrackBar: TTrackBar;
```

Из событий компонента следует упомянуть событие

```
property OnChangeTracking: TNotifyEvent;
```

генерируемое при изменении положения бегунка, и событие, вызываемое при редактировании значения `Value`:

```
property OnChange: TNotifyEvent;
```

Компонент *TPopupBox*

Элемент управления `TPopupBox` по своему духу очень близок к классическому комбинированному списку. Разница заключается лишь в том, что на этот раз список текстовых элементов не выпадает из компонента, а всплывает над ним (рис. 49.4).

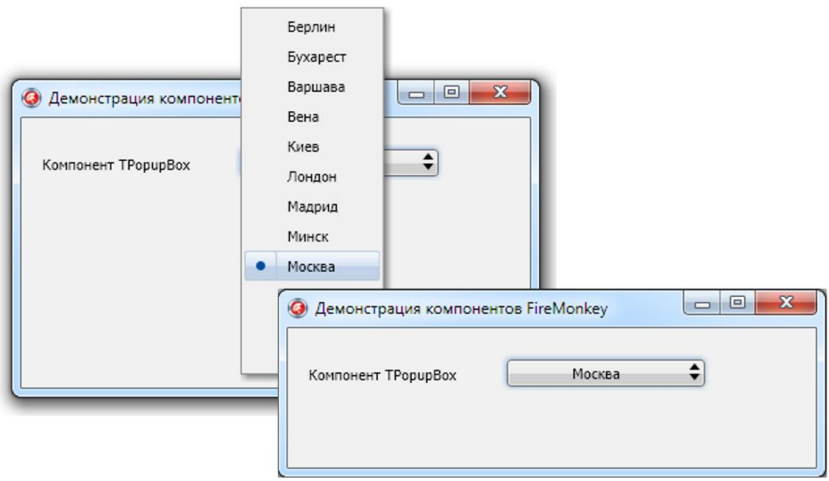


Рис. 49.4. Элемент управления `TPopupBox`

Для управления компонентом достаточно знать о трех свойствах. Свойство

```
property Items: TStrings;
```

предоставляет доступ к текстовому списку элементов.

Индекс выбранного элемента отражает свойство

```
property ItemIndex: Integer;
```

Если индекс принимает значение больше и равное 0 (другими словами, пользователь выбрал элемент в списке), то можно прочитать текст выбранной надписи, для этого предназначено свойство

```
property Text :string;
```

Основным событием компонента является событие, генерируемое в момент изменения активного элемента:

```
property OnChange: TNotifyEvent;
```

Сетки *TGrid* и *TStringGrid*

В составе элементов управления FMX имеются два компонента, специализирующихся на представлении данных в табличном виде, — это компоненты `TGrid` и `TStringGrid`. Оба элемента управления объявлены в модуле `FMX.Grid` и являются наследниками одного и того же опорного класса сеток:

```
type TCustomGrid = class(TScrollBox, IItemsContainer)
```

Основная разница между компонентами в том, что сетка `TStringGrid` нацелена на обслуживание и хранение текстовых данных, а сетка `TGrid` более универсальна и позволяет работать с более широким спектром данных. Однако `TGrid` не способна их хранить. Такое функциональное разделение несколько напоминает ситуацию с сетками `TStringGrid` и `TDrawGrid` из библиотеки VCL, но принцип построения сеток `FMX.Grid.TGrid` и `FMX.Grid.TStringGrid` абсолютно не похож на архитектуру их "коллег" из состава VCL.

Основной строительный элемент сеток FMX — колонка. Колонка — это экземпляр класса `TColumn` или экземпляр одного из потомков этого класса. Например, сетка строк `TStringGrid` специализируется исключительно на текстовых колонках `TStringColumn`, а сетка `TGrid` вдобавок умеет обслуживать весьма экзотические колонки (табл. 49.1).

Таблица 49.1. *Колонки и ячейки сеток*

Тип колонки	Тип ячейки	Описание
`TStringColumn`	`TTextCell = class(TEdit)`	Текст
`TCheckColumn`	`TCheckCell = class(TCheckBox)`	Флажок
`TProgressColumn`	`TProgressCell = class(TProgressBar)`	Шкала
`TPopupColumn`	`TPopupCell = class(TPopupBox)`	Всплывающая панель
`TImageColumn`	`TImageCell = class(TImageControl)`	Изображение

Для создания колонок во время визуального проектирования проще всего воспользоваться контекстным меню компонента-сетки и выбрать там пункт **Items Editor**. В ответ на это действие Delphi отобразит на экране окно дизайнера **Items Designer** (рис. 49.5). Дальше программист выбирает тип колонки и добавляет колонку в сетку. Колонка является вполне самостоятельным объектом, обладающим свойствами, методами и обработчиками событий.

Каждая из ячеек колонки во время редактирования данных интерпретируется как элемент управления определенного класса (см. табл. 49.1). Например, в колонке `TStringColumn`, специализирующейся на обслуживании, текста каждая из ячеек фактически является строкой ввода `TEdit`.

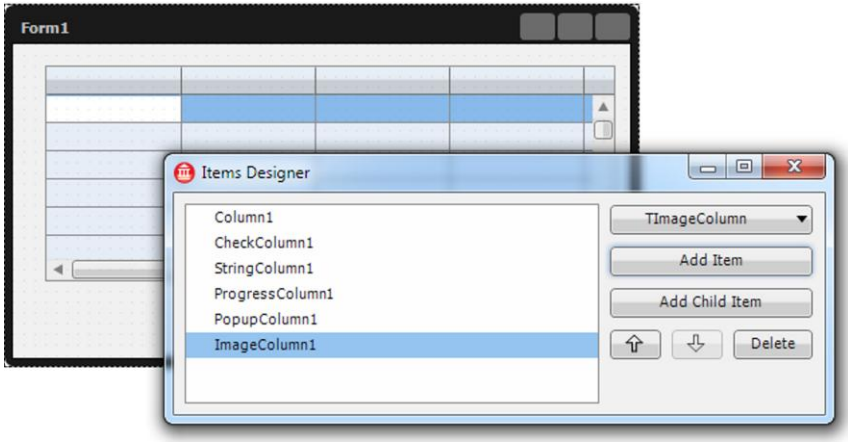


Рис. 49.5. Редактор колонок компонента-сетки `TGrid`

Размер сетки можно уточнить благодаря свойствам:

```
property RowCount: Integer;
property ColumnCount: Integer; //только для чтения
```

Обратите внимание на то, что определяющее количество строк в таблице свойство `RowCount` позволяет изменять число строк в сетке. А свойство `ColumnCount` доступно только для чтения, т. к. количество колонок в сетке определяется числом объектов `TColumn`.

Доступ к экземпляру колонки проще всего производить по ее индексу:

```
property Columns[Index: Integer]: TColumn;
```

Кроме того, существует метод, идентифицирующий колонку по экранным координатам:

```
function ColumnByPoint(const X, Y: Single): TColumn;
```

Для доступа к интегрированному в ячейку элементу управления следует вызвать метод, возвращающий ссылку на объект:

```
function CellControlByRow(Row: Integer): TStyledControl;
```

Заметьте, что в данном случае возвращается обезличенный тип данных `TStyledControl`, который следует привести к требуемому виду при посредничестве классов `TTextCell`, `TCheckCell`, `TProgressCell`, `TPopupCell` или `TImageCell`.

Во время создания пользовательского хранилища данных окажутся полезными события:

```
property OnSetValue: TOnSetValue;
type TOnSetValue = procedure(Sender: TObject; const Col, Row: Integer;
    const Value: Variant) of object;
property OnGetValue: TOnGetValue;
type TOnGetValue = procedure(Sender: TObject; const Col, Row: Integer;
    var Value: Variant) of object;
```

Событие `OnSetValue` генерируется в момент получения ячейкой с координатами `Col` и `Row` значения `Value`. Обратное событие `OnGetValue` вызывается во время чтения значения `Value` из ячейки.

Упомянем еще одно событие сетки, связанное с обслуживанием данных:

```
property OnEdititingDone: TOnEdititingDone;
type TOnEdititingDone = procedure(Sender: TObject;
                        const Col, Row: Integer) of object;
```

На этот раз речь идет о редактировании содержимого ячейки.

> ***Внимание!***
>
> Данные не сохраняются в сетке `TGrid`! Сетка `TGrid` не обладает навыками по хранению данных — эту задачу программист должен решить самостоятельно, создав собственное хранилище.

Полезное преимущество сетки строк `TStringGrid` перед обычной сеткой `TGrid` заключается в том, что она способна хранить свои текстовые данные. Несмотря на изменения во внутреннем построении сетки строк `TStringGrid`, разработчики компонента сохранили в нем традиционное свойство

```
property Cells[ACol, ARow: Integer]: string;
```

Благодаря этому свойству сетку можно рассматривать как двумерный текстовый массив, в котором каждая ячейка адресуется по индексу колонки и ряда.

ГЛАВА 50

Анимация

Прежде чем мы начнем говорить, без всякого преувеличения, о богатейших возможностях FireMonkey по анимации, уделим внимание источникам этих возможностей.

В классических приложениях VCL для Windows весь механизм графического вывода построен на фундаменте GDI (Graphics Device Interface). Для того чтобы нарисовать линию или вывести строку текста, программисту достаточно получить доступ к контексту нужного графического устройства и воспользоваться услугами стандартных функций двумерной графики Windows. Несмотря на относительную "древность", GDI превосходно справляется с большинством задач, которые решают деловые приложения. А если программист ставит перед собой более амбициозные цели, то в качестве инструментария Microsoft предлагает воспользоваться GDI+, Direct2D и Direct3D.

Скажем несколько слов о графическом механизме Mac OS X. Я думаю, что не стоит объяснять, что эта ОС ничего не знает ни о GDI, ни о DirectX. В Mac OS X работа с двумерной графикой осуществляется силами QuickDraw и Quartz 2D, а трехмерные сцены создаются на основе графической библиотеки OpenGL.

Анимация

Современный пользователь крайне требователен и избалован. Он хочет работать на самом совершенном программном обеспечении с самым изысканным пользовательским интерфейсом. О разработке профессионального программного обеспечения мы говорили на протяжении всей книги, а теперь настала очередь уделить внимание проектированию привлекательного интерфейса наших приложений.

Библиотека FireMonkey позиционируется не только как средство разработки кроссплатформенных приложений для Windows и Mac OS X, но и как инструмент построения изысканного пользовательского интерфейса. Особо отрадно, что для создания привлекательного интерфейса от вас, как от программиста, потребуется минимум затрат. В какой-то степени будет корректнее говорить не столько о программировании, сколько о дизайне. Еще один плюс FireMonkey в том, что если вы обладаете достаточно высоким художественным вкусом, то внешний вид и привлекательность ваших проектов окажется вне всякой конкуренции. А это очень важно для победы над конкурентом на рынке программного обеспечения, ведь, как гласит поговорка, встречают по одежке...

Как сделать интерфейс наглядным и интуитивно понятным? Существует множество рекомендаций по размещению и группировке элементов управления, по заданию очередности

передачи фокуса ввода, по использованию пояснительных надписей, по подбору цветовых схем и по многому другому. Сегодня мы рассмотрим еще одно направление в разработке пользовательского интерфейса — анимированные элементы управления.

> ***Замечание***
>
> В FireMonkey под анимацией понимается управляемое воздействие на элементы управления пользовательского интерфейса, приводящее к изменению их местоположения, размеров и визуальных характеристик во времени и в пространстве.

Простой пример анимации

Механизм анимации настолько прост, что в минимальной нотации не потребует от нас ни одной строки кода. Создайте новый проект FireMonkey и разместите на форме круг `Circle1:TCircle` (страница палитры компонентов **Shapes**). Теперь обратитесь к странице палитры компонентов **Animations** и перенесите на форму компоненты:

- `FloatAnimation1:TFloatAnimation`, этот компонент позволяет управлять параметрами элемента управления описываемыми вещественными числами;
- `ColorAnimation1:TColorAnimation`, этот компонент предназначен для управления цветом компонента.

Подключите компоненты-аниматоры к кругу `Circle1`. Для этого проще всего воспользоваться услугами окна **Structure**, отображающего структуру нашего проекта (рис. 50.1). Убедившись, что отвечающие за анимацию компоненты попали в нужные "руки", перейдите к Инспектору объектов.

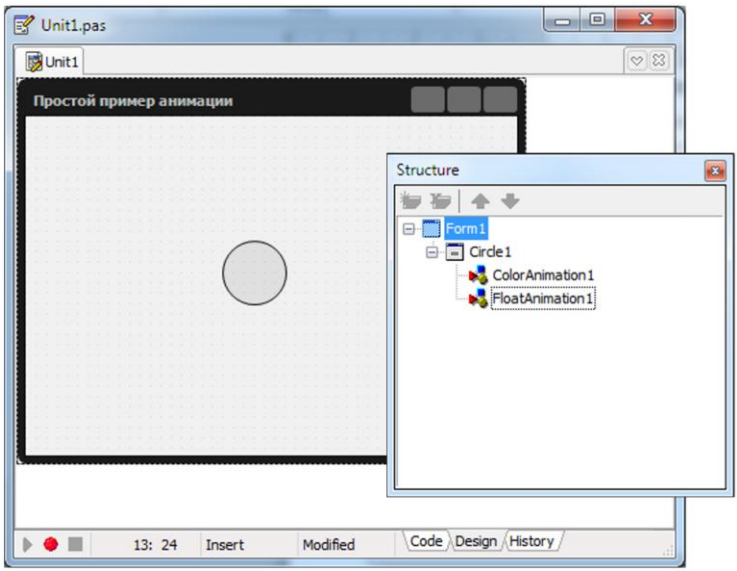


Рис. 50.1. Подключение объектов анимации к фигуре `TCircle`

Выберите в Инспекторе объектов компонент `FloatAnimation1`. Научим его управлять горизонтальным размером круга `Circle1`. Для этих целей, вооружившись мышью, осуществим 4 операции:

1. В свойстве `PropertyName` находим анимируемое свойство `Scale.X`, отвечающее за горизонтальный масштаб компонента.
2. Настраиваем свойства, задающие диапазон изменяемых значений масштаба `StartValue=1` и `StopValue=2`.
3. Выбираем событие-триггер, выступающее инициатором анимации. Установив свойство `Trigger` в состояние `IsMouseOver=true`, мы укажем, что анимация начинается в момент появления над компонентом указателя мыши.
4. Выбираем в свойстве `TriggerInverse` событие, завершающее анимацию: `IsMouseOver=false` (указатель мыши покинул анимируемый объект).

На этом настройка анимации для компонента `FloatAnimation1` завершена. Теперь выберите компонент `ColorAnimation1` и в Инспекторе объектов настройте его поведение в соответствии с рис. 50.2. Анимированный проект готов к старту. Согласитесь, что программисты Embarcadero потрудились на славу, оставив на нашу долю лишь визуальную сторону проектирования.

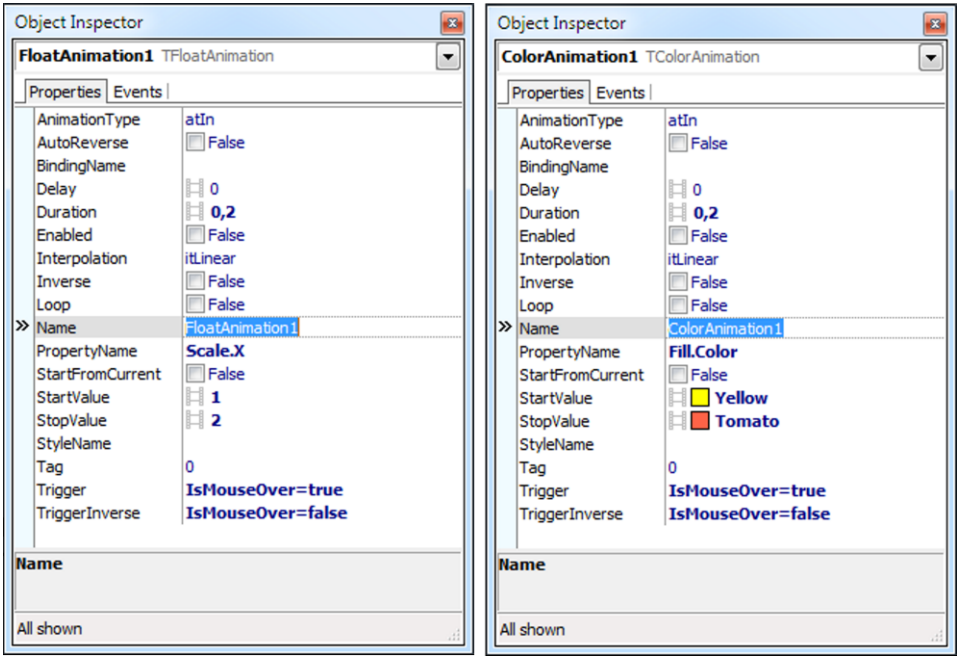


Рис. 50.2. Настройка свойств компонентов `FloatAnimation1` и `ColorAnimation1`

Общие черты компонентов-аниматоров, класс *TAnimation*

Все компоненты-аниматоры (которые вы обнаружите на странце **Animation** палитры компонентов Delphi) берут начало от класса `TAnimation`. Соответственно большинство свойств и методов наследники "впитывают" от своего предка. Рассмотрим наиболее важные из них.

Длительность анимации определяется значением свойства

```
property Duration: Single; //единица измерения — секунды
```

При необходимости перед началом анимации можно выждать небольшую паузу. Время задержки устанавливается в свойстве

```
property Delay: Single; //по умолчанию 0 секунд
```

Закон, в соответствии с которым анимируемый объект станет переходить из одного состояния в другое, определяется свойством

```
property Interpolation: TInterpolationType;
type TInterpolationType = (itLinear, itQuadratic, itCubic, itQuartic,
    itQuintic, itSinusoidal, itExponential, itCircular,
    itElastic, itBack, itBounce);
```

По умолчанию установлен обычный линейный переход `itLinear`, приводящий к последовательной смене состояний (например, цвета или размера). Вместе с тем тип данных `TInterpolationType` предоставляет такой большой выбор допустимых вариантов поведения, что законом интерполяции наверняка стоит поэкспериментировать. На математический аспект интерполяции оказывает влияние вид анимации:

```
property AnimationType: TAnimationType;//по умолчанию atIn
type TAnimationType = (atIn, atOut, atInOut);
```

Сторонникам бесконечного цикла смены размеров (местоположения, цвета и т. п.) стоит установить в `true` свойство

```
property Loop: Boolean; //по умолчанию false
```

Управление анимацией обычно осуществляется за счет свойств-триггеров

```
property Trigger: TTrigger;        //старт анимации
property TriggerInverse: TTrigger; //завершение анимации
```

указывающих, какое именно событие и при каких условиях послужит поводом для начала и завершения анимации. В качестве примера таких событий можно привести получение и утрату элементом фокуса ввода (`IsFocused=true` и `IsFocused=false`) или появление над объектом указателя мыши (`IsMouseOver=true`) и уход указателя (`IsMouseOver=false`).

Если процесс анимации в данный момент активен, то свойство

```
property Running: Boolean; //только для чтения
```

возвратит значение `true`.

При необходимости старт и остановка анимации осуществляются из кода:

```
procedure Start;         //старт анимации
procedure Stop;          //остановка анимации
procedure StopAtCurrent; //остановка анимации в текущем состоянии
```

Временно приостановить процесс позволит свойство

```
property Pause: Boolean;
```

Как и положено полноценным объектам, все компоненты-аниматоры способны генерировать события. Старт анимации сопровождается вызовом события

```
property OnProcess: TNotifyEvent;
```

О завершении анимации уведомляет событие

```
property OnFinish: TNotifyEvent;
```

Полученных знаний вполне достаточно для того, чтобы попробовать анимировать объект не в период визуальной разработки, а во время выполнения программы. Во фрагменте кода, предложенном в листинге 50.1, динамически создается и подключается к панели `Panel1` экземпляр класса `TFloatAnimation`.

Листинг 50.1. Создание анимации во время выполнения программы

```
with TFloatAnimation.Create(Panel1) do
begin
  Parent:=Panel1;
  PropertyName:='RotationAngle';
  StartValue:=0;
  StopValue:=180;
  Trigger:='IsMouseOver=true';
  TriggerInverse:='IsMouseOver=false';
end;
```

Обратите внимание на то, что для подключения к анимируемому объекту нам потребовалось передать ссылку на этот объект в свойство `Parent`.

Свойство

```
property Inverse: Boolean; //по умолчанию false
```

позволяет инвертировать анимацию (например, заставить сменяться кадры `TBitmapAnimation` в обратном порядке).

Индивидуальные особенности компонентов-аниматоров

На странице **Animations** палитры компонентов Delphi расположилось 9 компонентов, предоставляющих программисту разноплановые услуги по анимации объектов FMX. Каждый из компонентов обладает индивидуальными чертами, определяющими область применения того или иного потомка класса `TAnimation`.

После подключения к анимируемому объекту (во время выполнения программы для этого достаточно воспользоваться свойством `Parent`, а во время визуального проектирования просто перетащить мышью узел компонента-аниматора в подчинение обслуживаемого объекта (рис. 50.1)) следует выбрать управляемое свойство. Для этого предназначено свойство

```
property PropertyName: AnsiString;
```

Надо понимать, что далеко не все свойства могут быть анимированы. Так, если вы пытаетесь управлять заливкой объекта с помощью компонента `TColorAnimation`, то в `PropertyName` могут быть переданы только названия свойств, работающих с цветом кисти (`Fill` и `Stroke`).

> ***Замечание***
>
> Во время визуального проектирования Инспектор объектов подскажет программисту, какие именно свойства могут быть задействованы в анимации.

Для управления правилами анимации у большинства компонентов-аниматоров имеется пара свойств: `StartValue` и `StopValue`, однако тип обслуживаемых значений у каждого из аниматоров индивидуальный, зависящий от стоящих перед компонентом задач.

Цветовая анимация, компонент *TColorAnimation*

Компонент `TColorAnimation` позволяет управлять цветовыми характеристиками кисти заливки (свойство `Fill`) или кисти вывода графических примитивов (свойство `Stroke`). Анимируются два цвета, значения которых указываются в свойствах:

```
property StartValue: TAlphaColor;
property StopValue: TAlphaColor;
```

Градиентная анимация, компонент *TGradientAnimation*

Благодаря градиентной анимации, реализуемой с помощью компонента `TGradientAnimation`, можно значительно улучшить внешний вид объектов, способных управлять своими цветовыми характеристиками с помощью свойств `Fill` и `Stroke`.

Параметры анимации назначаются при посредничестве свойств:

```
property StartValue: TGradient;
property StopValue: TGradient;
```

Но теперь вместо данных о цвете (как это было в `TColorAnimation`) мы должны указать градиентные настройки `TGradient`.

Анимированная картинка, компонент *TBitmapAnimation*

Компонент `TBitmapAnimation` может работать совместно с объектами, способными отображать растровые картинки (обладающие свойством `Bitmap`). Эффект анимации достигается путем смены пары картинок, хранимых в свойствах

```
property StartValue: TBitmap;
property StopValue: TBitmap;
```

Анимированный ряд, компонент *TBitmapListAnimation*

Возможности компонента `TBitmapAnimation` меркнут по сравнению с его "коллегой" `TBitmapListAnimation`, ведь последний умеет управлять не парой, а целым списком карти-

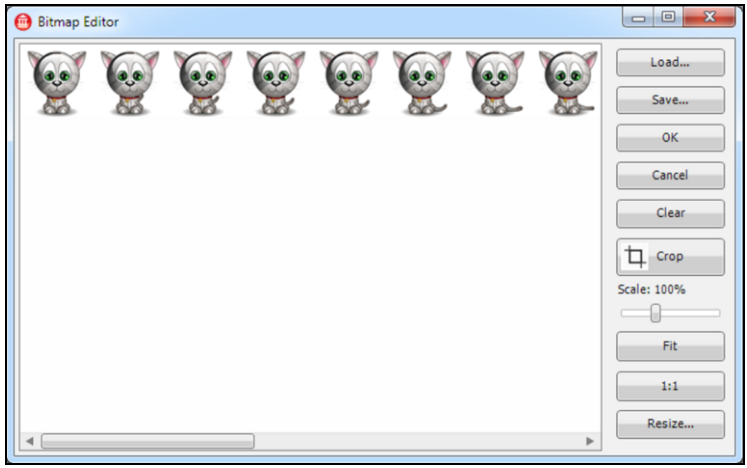


Рис. 50.3. Редактор **Bitmap Editor** компонента `TBitmapListAnimation`

нок. Так что при остром желании TBitmapListAnimation способен показать пользователю короткометражный мультфильм. К кадрам анимации не предъявляется особых требований за исключением одного — все они должны быть одинакового размера и склеены один за другим в одну ленту (рис. 50.3).

Лента с картинками передается в свойство

```
property AnimationBitmap: TBitmap;
```

Для того чтобы компонент-аниматор смог корректно воспроизводить наш "шедевр" мультипликации, следует уведомить его о количестве кадров в ленте:

```
property AnimationCount: Integer;
```

Анимация числовых свойств, компонент *TFloatAnimation*

Компонент TFloatAnimation предназначен для управления свойствами компонентов, обслуживающих числовые значения вещественного типа. Среди потенциальных клиентов компонента-аниматора свойства, отвечающие за определение местоположения элемента управления, размеры элемента управления, угол поворота, масштаб, прозрачность и т. п. Стартовое и конечное значения анимации заносятся в традиционные свойства

```
property StartValue: Single;
property StopValue: Single;
```

Анимация прямоугольной области, компонент *TRectAnimation*

Задача компонента TRectAnimation заключается в воздействии на значения свойств Margins или Padding элемента управления (напомню, что названные свойства определяют величины отступов краев дочернего объекта от границ клиентской области родительского контейнера). В результате мы получаем эффект изменения размеров анимируемого объекта относительно его контейнера-владельца.

Вновь нашими основными помощниками станут свойства

```
property StartValue: TBounds;
property StopValue: TBounds;
```

но на этот раз они типизируются структурой TBounds.

Анимация траектории, компонент *TPathAnimation*

Проявив немного терпения можно заставить объект двигаться по заданной траектории. Для анимации этого весьма занятного действа стоит воспользоваться услугами компонента TPathAnimation. Траектория движения задается с помощью свойства

```
property Path: TPathData;
```

Свойство предоставляет доступ к классу TPathData, предоставляющему услуги по определению контрольных точек траектории.

Стоит отметить существование еще одного свойства:

```
property Rotate: Boolean; //по умолчанию false
```

которое в состоянии true заставит вращаться анимируемый объект вокруг своей оси.

Проиллюстрируем работу с анимированной траекторией на небольшом примере. Для этого нам понадобится помощь фигуры Circle1:TCircle, к которой следует подключить компо-

нент `PathAnimation1:TPathAnimation`. Настройте компонент `PathAnimation1` следующим образом:

- триггер `Trigger` активации анимации переведите в состояние `IsVisible=true` (анимация включена, если объект виден);
- продолжительность анимации `Duration=6`;
- активируем автореверс `AutoReverse=true`;
- выберем синусоидальную интерполяцию `Interppolation=itSinusoidal`.

Точки траектории заполним в коде программы, для этого подойдет событие создания формы (листинг 50.2). В качестве контрольных точек мы выберем четыре угла родительского контейнера, которому принадлежит окружность `Circle1:TCircle`.

Листинг 50.2. Настройка параметров траектории

```
with PathAnimation1 do
begin
  Stop;
  Circle1.Position.Point:=PointF(0,0);
  Path.Clear;
  Path.MoveTo(PointF(0,0));
  Path.MoveTo(PointF((TControl(Circle1.Parent).Width -
                      Circle1.Width)/2,0));
  Path.MoveTo(PointF(TControl(Circle1.Parent).Width -
                     Circle1.Width,0));
  Path.MoveTo(PointF(TControl(Circle1.Parent).Width-Circle1.Width,
                TControl(Circle1.Parent).Height-Circle1.Height));
  Path.MoveTo(PointF((TControl(Circle1.Parent).Width -Circle1.Width)/2,
                 TControl(Circle1.Parent).Height-Circle1.Height));
  Path.MoveTo(PointF(0,TControl(Circle1.Parent).Height-Circle1.Height));
  Path.MoveTo(PointF(0,0));
  Path.ClosePath;
  Start;
end;
```

После старта приложения окружность начнет свое движение вдоль границ контейнера.

Управление графической производительностью

При работе с графикой приложения FireMonkey в первую очередь стараются задействовать всю вычислительную мощь процессора видеокарты, и только если это невозможно, нагрузка перекладывается на центральный процессор. Вместе с тем, нам как разработчикам программного обеспечения следует понимать, что не каждый пользователь будет готов немедленно отправиться в магазин за очередным видеоадаптером, лишь бы насладиться изысканным интерфейсом наших с вами программ.

Для того чтобы пользователь смог оптимизировать вычислительную нагрузку, накладываемую на графический и центральный процессоры компьютера, программисту стоит позволить ему управлять состоянием ряда глобальных переменных, объявленных в модуле `FMX.Types` (табл. 50.1).

Таблица 50.1. *Настройка графической производительности Windows*

Переменная	Умолчание	Описание
`var GlobalDisableFocusEffect: Boolean;`	`false`	Для устройств с низкой производительностью установить в `true`
`var GlobalUseDirect2D: Boolean;`	`true`	Использовать аппаратный Direct2D
`var GlobalUseDirect2DSoftware: Boolean;`	`false`	Установить программную эмуляцию Direct2D
`var GlobalUseHWEffects: Boolean;`	`true`	Использовать аппаратное ускорение, где это возможно

ПРИЛОЖЕНИЯ

Приложение 1. Математика, статистика и тригонометрия
Приложение 2. Работа со строками и символами
Приложение 3. Работа с датой и временем
Приложение 4. Работа с памятью
Приложение 5. Управление ходом выполнения программы
Приложение 6. Работа с именами папок и файлов
Приложение 7. Модуль *IOUtils*
Приложение 8. Константы CSIDL
Приложение 9. Холст *FMX.Types.TCanvas*
Приложение 10. Описание электронного архива

ПРИЛОЖЕНИЕ 1

Математика, статистика и тригонометрия

Большинство представленных в табл. П1.1—П1.10 функций требует подключения модуля `Math`.

Таблица П1.1. *Математические функции*

Функция/процедура	Описание
`function Sqr(X: Extended): Extended;`	Возведение в квадрат
`function IntPower(Base: Extended; Exponent: Integer): Extended;`	Возводит число `Base` в степень `Exponent` (целого типа)
`function Power(Base, Exponent: Extended): Extended;`	Возводит число `Base` в степень `Exponent` (действительного типа)
`function Ldexp(X: Extended; P: Integer): Extended;`	Умножает число `X` на `2` в степени `P`
`function Sqrt(X: Extended): Extended;`	Квадратный корень
`procedure DivMod(Dividend: Cardinal; Divisor: Word; var Result, Remainder: Word);`	Возвращает результат целочисленного деления, включая остаток от деления
`function Exp(X: Real): Real;`	Экспонента
`procedure Frexp(X: Extended; var Mantissa: Extended; var Exponent: Integer) register;`	Возвращает мантиссу и порядок заданной величины
`function Ln(X: Real): Real;`	Натуральный логарифм
`function LnXP1(X: Extended): Extended;`	Натуральный логарифм числа (`X`+1)
`function Log2(X: Extended): Extended;`	Логарифм `X` по основанию 2
`function Log10(X: Extended): Extended;`	Логарифм `X` по основанию 10
`function LogN(N, X: Extended): Extended;`	Логарифм `X` по основанию `N`
`function Sign(const AValue: Extended): TValueSign;`	Возвращает знак числа `AValue`: 0 — нулевое значение; –1 — отрицательное значение; 1 — положительное значение
`function IsZero(const A: Extended; Epsilon: Extended): Boolean;`	Проверка на приближение значения к 0
`function IsInfinite(const AValue: Extended): Boolean;`	Проверка на приближение числа к бесконечности

Таблица П1.2. *Округление чисел*

Метод	Описание
`function Abs(X);`	Возвращает абсолютное значение числа — число по модулю: `R:=Abs(-2.3); // 2.3`
`function Frac(X: Extended): Extended;`	Возвращает дробную часть числа: `R:=Frac(-123.456); // -0.456`
`function Int(X: Extended): Extended;`	Возвращает целую часть от числа: `R:=Int(123.456); // 123.0`
`function Round(X: Extended): Int64;`	Округляет до целого числа

`function RoundTo(const AValue: Extended; const ADigit: TRoundToEXRangeExtended): Extended;`	Округляет действительное число `AValue` до указанного знака после запятой (отрицательное значение `ADigit`) или степени 10 (положительное значение `ADigit`). Использует не математическое округление, а так называемое "округление банкира", например:
	`R:=RoundTo(2.445, -2)// 2.44`
	`R:=RoundTo(2.4451, -2);//2.46`
	`R:=RoundTo(78537568, 2);//78537600`

`function SimpleRoundTo(const AValue: Extended; const ADigit: TRoundToRange = -2): Extended;`	Осуществляет математическое округление действительного числа `AValue` до указанного знака после запятой (отрицательное `ADigit`) или степени 10 (положительное `ADigit`)

`function Floor(X: Extended): Integer;`	Округляет вещественное число до целого в его меньшую сторону:
	`R:=Floor(-2.8) // -3`
	`R:=Floor(2.8)  // 2`

`function Ceil(X: Extended):Integer;`	Округляет вещественное число до целого в большую сторону:
	`R:=Ceil(-2.8); // -2`
	`R:=Ceil(2.8);  // 3`

`function Trunc(X: Extended): Int64;`	Преобразует вещественное число в целое, усекая дробную часть
`function Odd(X: Longint): Boolean;`	Возвращает `true`, если `X` — нечетное число

Таблица П1.3. *Проверка вхождения значения в диапазон*

Функция	Описание

`function InRange(const AValue, AMin, AMax: Integer): Boolean; overload;`
`function InRange(const AValue, AMin, AMax: Int64): Boolean; overload;`
`function InRange(const AValue, AMin, AMax: Double): Boolean; overload;`
	Перегружаемые версии функции, проверяющей принадлежность значения `AValue` диапазону от `AMin` до `AMax`. При выполнении условия возвращают `true`

`function EnsureRange(const AValue, AMin, AMax: Integer): Integer; overload;`
`function EnsureRange(const AValue, AMin, AMax: Int64): Int64; overload;`
`function EnsureRange(const AValue, AMin, AMax: Double): Double; overload;`
	Перегружаемые функции, возвращающие значение, гарантированно входящее в диапазон от `AMin` до `AMax`. Если `AValue` принадлежит диапазону, то функция возвратит `AValue`. Если `AValue` меньше минимально допустимого значения, то функция возвратит `AMin`. Если `AValue` превышает верхнюю границу диапазона, то функция возвратит `AMax`

Таблица П1.3 *(окончание)*

Функция	Описание
`function VarInRange(const AValue, AMin, AMax: Variant): Boolean;`	Значением `true` сигнализирует о вхождении значения `AValue` в диапазон `AMin..AMax`. Все параметры вариантного типа
`function Low(X);`	Возвращает значение самого малого аргумента из диапазона (массива или строки)
`function High(X);`	Возвращает значение самого большого аргумента из диапазона (массива или строки)
`function Hi(X): Byte;`	Возвращает содержимое старшего байта аргумента `X`
`function Lo(X): Byte;`	Возвращает содержимое младшего байта аргумента `X`

Таблица П1.4. *Сравнение чисел*

Функция	Описание
`function Max(A,B: Integer): Integer;`	Возвращает наибольшее из двух чисел. Функция перегружаемая и работает как с целыми, так и с действительными типами
`function Min(A,B: Integer): Integer;`	Возвращает наименьшее из двух чисел. Функция перегружаемая и работает как с целыми, так и с действительными типами

Таблица П1.5. *Работа с массивами*

Функция	Описание
`function MaxIntValue(const Data: array of Integer): Integer;`	Возвращает наибольшее число из массива чисел целого типа
`function MinIntValue(const Data: array of Integer): Integer;`	Возвращает наименьшее число из массива чисел целого типа
`function MaxValue(const Data: array of Double): Double;`	Возвращает наибольшее число из массива чисел действительного типа
`function MinValue(const Data: array of Double): Double;`	Возвращает наименьшее число из массива действительного типа
`function SumInt(const Data: array of Integer): Integer;`	Возвращает сумму элементов массива целого типа
`function SUM(const Data: array of Extended): Extended;`	Возвращает сумму элементов массива действительного типа
`function SumOfSquares(const Data: array of Extended): Extended;`	Возвращает сумму возведенных в квадрат элементов массива действительного типа
`procedure SumsAndSquares(const Data: array of Extended; var Sum, SumOfSquares: Extended);`	Возвращает сумму элементов и сумму возведенных в квадрат элементов массива действительного типа
`function Slice(var A: array; Count: Integer): array;`	Возвращает часть массива от первого значения до значения `Count`

Таблица П1.6. *Тригонометрические процедуры и функции*

Функция	Описание
`function Pi: Extended;`	Значение константы π = 3.14159...
`function Cos(X: Extended): Extended;`	Косинус в радианах
`function ArcCos(X: Extended): Extended;`	Арккосинус
`function Cosh(X: Extended): Extended;`	Гиперболический косинус
`function ArcCosh(X: Extended): Extended;`	Гиперболический арккосинус
`function Sin(X: Extended): Extended;`	Синус в радианах
`function ArcSin(X: Extended): Extended;`	Арксинус. Значение `X` должно быть в пределах от –1 до 1. Возвращает значение в радианах
`function Sinh(X: Extended): Extended;`	Гиперболический синус
`function ArcSinh(X: Extended): Extended;`	Гиперболический арксинус
`function Tan(X: Extended): Extended;`	Тангенс
`function ArcTan(X: Extended): Extended;`	Арктангенс
`function ArcTanh(X: Extended): Extended;`	Гиперболический арктангенс. Значение `X` должно быть в пределах от –1 до 1
`function ArcTan2(Y, X: Extended): Extended;`	Вычисляет `ArcTan(Y/X)` и возвращает угол в правильном квадранте. Значения `X` и `Y` должны быть между –264 и 264. Кроме того, значение `X` не может быть равно 0
`function Cotan(X: Extended): Extended;`	Котангенс угла `X`, $X \neq 0$
`procedure SinCos(Theta: Extended; var Sin, Cos: Extended);`	Вычисление синуса и косинуса
`function Hypot(X, Y: Extended): Extended;`	Расчет гипотенузы

Таблица П1.7. *Пересчет углов (радианы, циклы, градусы)*

Функция	Описание
`function CycleToRad(Cycles: Extended): Extended;`	Циклы в радианы: radians = 2 × π × cycles
`function RadToCycle(Radians: Extended): Extended;`	Радианы в циклы: cycles = radians/(2 × π)
`function DegToRad(Degrees: Extended): Extended;`	Градусы в радианы: radians = degrees(π/180)
`function RadToDeg(Radians: Extended): Extended;`	Радианы в градусы: degrees = radians(180/π)
`function GradToRad(Grads: Extended): Extended;`	Грады в радианы: radians = grads(π/200)
`function RadToGrad(Radians: Extended): Extended;`	Радианы в грады: grads = radians(200/π)

Таблица П1.8. *Суммы, среднее арифметическое, дисперсия*

Функция/процедура	Описание
`function Mean(const Data: array of Double): Extended;`	Возвращает среднее арифметическое от массива действительного типа
`function StdDev(const Data: array of Double): Extended;`	Возвращает среднеквадратическое отклонение
`procedure MeanAndStdDev(const Data: array of Double; var Mean, StdDev: Extended);`	Одновременно вычисляет среднее арифметическое и среднеквадратическое отклонение
`function Sum(const Data: array of Double): Extended; register;`	Сумма всех элементов массива действительных чисел
`function SumInt(const Data: array of Integer): Integer register;`	Сумма всех элементов массива целых чисел
`function SumOfSquares(const Data: array of Double): Extended;`	Сумма квадратов всех элементов массива
`procedure SumsAndSquares(const Data: array of Double; var Sum, SumOfSquares: Extended) register;`	Одновременно вычисляет сумму и сумму квадратов элементов массива
`function Norm(const Data: array of Double): Extended;`	Возвращает квадратный корень из суммы квадратов (норму)
`function Variance(const Data: array of Double): Extended;`	Вычисляет статистическую типовую дисперсию из массива данных
`function TotalVariance(const Data: array of Double): Extended;`	Полная дисперсия чисел `SUM(i=1,N)[(X(i) — Mean)**2]`
`function PopnVariance(const Data: array of Double): Extended;`	Вычисляет дисперсию совокупности всех значений в массиве данных, используя `n` и (смещенный) метод: `TotalVariance/n`
`procedure MomentSkewKurtosis(const Data: array of Double; var M1, M2, M3, M4, Skew, Kurtosis: Extended);`	Вычисляет первые четыре порядка `M1...M4` статистических моментов, асимметрию `Skew` и эксцесс `Kurtosis` для массива данных `Data`

Таблица П1.9. *Генерация псевдослучайных чисел*

Функция/процедура	Описание
`procedure Randomize;`	Автоматическая инициализация генератора псевдослучайных чисел
`function RandG(Mean, StdDev: Extended): Extended;`	Генерирует псевдослучайное число, используя распределение Гаусса
`function Random[(Range: Integer)];`	Генерирует псевдослучайное число в пределах от 0 до значения `Random`. Для установки начального значения генератора псевдослучайных чисел используйте переменную `RandSeed:LongInt`

Таблица П1.10. *Финансовые процедуры и функции*

Функция	Описание
function DoubleDecliningBalance(Cost, Salvage: Extended; Life, Period: Integer): Extended;	Вычисляет амортизацию актива методом двойного баланса
function FutureValue(Rate: Extended; NPeriods: Integer; Payment, PresentValue: Extended; PaymentTime: TPaymentTime): Extended;	Вычисляет будущее значение капиталовложения
function InterestPayment(Rate: Extended; Period, NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Вычисляет проценты по оплате ссуды
function InterestRate(NPeriods: Integer; Payment, PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Вычисляет норму прибыли для получения запланированной суммы
function InternalRateOfReturn(Guess: Extended; const CashFlows: array of Double): Extende	Вычисляет внутреннюю скорость оборота капиталовложения для проведения последовательных выплат
function NetPresentValue(Rate: Extended; const CashFlows: array of Double; PaymentTime: TPaymentTime): Extended;	Вычисляет текущее значение стоимости вложения для проведения последовательных выплат с учетом процентной ставки
function NumberOfPeriods(Rate, Payment, PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Возвращает число периодов, за которые вложение достигнет заданной величины
function Payment (Rate: Extended; NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Расчет размеров периодической выплаты для погашения ссуды
function PeriodPayment(Rate: Extended; Period, NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Расчет платежей по процентам за определенный период
function PresentValue(Rate: Extended; NPeriods: Integer; Payment, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Текущее значение вложения
function SLNDepreciation(Cost, Salvage: Extended; Life: Integer): Extended;	Расчет амортизации методом постоянной нормы
function SYDDepreciation(Cost, Salvage: Extended; Life, Period: Integer): Extended;	Расчет амортизации при помощи весовых коэффициентов

ПРИЛОЖЕНИЕ 2

Работа со строками и символами

Таблица П2.1. *Функции и процедуры для работы со строками* `String`

Функция/процедура	Описание
Удаление пробелов	
`function Trim(const S: string): string;`	Удаляет все пробелы и управляющие символы в начале и конце текстовой строки
`function TrimLeft(const S: string): string;`	Удаляет пробелы в начале строки и все управляющие символы
`function TrimRight(const S: string): string;`	Удаляет пробелы в конце строки и все управляющие символы
Сравнение строк	
`function CompareText (const S1, S2: string): Integer;`	Сравнивает две строки и возвращает 0, если строки идентичны. Функция не учитывает регистр символов
`function CompareStr(const S1, S2: string): Integer;`	Сравнивает две строки и возвращает 0, если строки идентичны. Функция чувствительна к регистру символов
`function AnsiSameText(const S1, S2: string): Boolean;`	Сравнивает две строки без учета регистра, но при этом контролируется кодовая страница строки. В случае, если строки идентичны, возвратит `true`
Форматирование строк	
`function Format(const Format: string; const Args: array of const): string;`	Функция возвращает текстовую строку, заполняя ее аргументами из массива `Args`. Место вставки аргументов и их форматирование определяются в строке `Format()` (табл. П2.2 и П2.3)
`procedure FmtStr(var StrResult: string; const Format: string; const Args: array of const);`	Аналогично функции `Format()`. Результат помещается в строку `StrResult`
`function LowerCase(const S: string): string;`	Возвращает строку в нижнем регистре
`function UpperCase(const S: string): string;`	Возвращает строку в верхнем регистре
`function QuotedStr(const S: string): string;`	Возвращает строку `S` в одинарных кавычках

Таблица П2.1 *(продолжение)*

Функция/процедура	Описание
Преобразование других типов данных к `String`	
function IntToStr(Value: Integer): **string**;	Преобразует целое число в строку
function StrToInt(**const** S: **string**): Integer;	Преобразует строку в целое число
function StrToIntDef (**const** S: **string**; Default: Integer): Integer;	Работает как `StrToInt()`, но при ошибке преобразования возвращает значение `Default`
function IntToHex(Value: Integer; Digits: Integer): **string**;	Преобразует целое число в строку, соответствующую его шестнадцатеричному представлению
function FloatToStr (Value: Extended): string;	Преобразует число с плавающей точкой в строку
function FloatToStrF (Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): **string**;	Преобразует значение `Value` в строку, в соответствии с форматом **type** TFloatFormat = (ffGeneral, ffExponent, ffFixed, ffNumber, ffCurrency); Точность преобразования определяется в `Precision`. Так для типа данных `Single` обычно `7`, `Double` — `15`, `Extended` — 18. Параметр `Digits` определяет число знаков после запятой
function FormatFloat (**const** Format: string; Value: Extended): **string**;	Наиболее совершенная функция по преобразованию вещественного числа в текстовую строку. Форматирование осуществляется в соответствии со спецификаторами формата в параметре `Format` (табл. П2.4)
function FormatMaskText (**const** EditMask: **string**; const Value: **string**): **string**;	Возвращает строку `Value`, отформатированную в соответствии с маской `EditMask`, функция объявлена в модуле `Mask`
function CurrToStr(Value: Currency): **string**;	Преобразует денежный тип в строку
function StrToCurr(**const** S: **string**): Currency;	Преобразует строку в денежный тип
function CurrToStrF (Value: Currency; Format: TFloatFormat; Digits: Integer): **string**;	Преобразует денежный тип в строку в соответствии с форматом `Format:TFloatFormat`
function FormatCurr(**const** Format: **string**; Value: Currency): **string**;	Аналогична функции `FormatCurr()`, преобразует денежный тип в строку (табл. П2.4)
function QuotedStr(**const** S: **string**): **string**;	Возвращает значение `S` внутри одинарных кавычек (`'`)
function AnsiQuotedStr (**const** S: **string**; Quote: Char): **string**;	Возвращает значение `S` внутри символа, определенного параметром `Quote`
Другие функции/процедуры	

procedure Delete(**var** S: string; Index, Count: Integer);	Удаляет из строки `S Count` символов, начиная с позиции `Index`:
	`s:='Embarcadero Delphi';`
	`Delete(s,1,12); // Delphi`

Таблица П2.1 *(окончание)*

Функция/процедура	Описание
`function Concat(s1 [, s2,..., sn]: string): string;`	Возвращает объединение двух и более строк
`function Copy(S; Index, Count: Integer): string;`	Возвращает субстроку из `Count` символов, начиная с позиции `Index`: `s:=Copy('Embarcadero Delphi',13,6);//Delphi`
`procedure Insert(Source: string; var S: string; Index: Integer);`	Вставляет в строку `S` подстроку `Source`, место вставки определяется через `Index`
`function Length(S): Integer;`	Возвращает длину строки `S`
`function Pos(Substr: string; S: string): Integer;`	Возвращает индекс первого вхождения символа `Substr` в строку `S`. Если такого символа в строке нет, то результат равен нулю
`function AdjustLineBreaks (const S: string): string;`	Возвращает строку, заканчивающуюся символами `CR/LF` (возврат каретки/новая строка)
`function AnsiReverseString (const AText: AnsiString): AnsiString;`	Возвращает реверсивное значение `AText`

ЗАМЕЧАНИЕ

Для работы со строками ANSI с учетом языкового драйвера предусмотрены функции, аналогичные рассмотренным в табл. П1.2. Это функции: `AnsiCompareStr()` — аналог `CompareStr()`, `AnsiLowerCase()` — `LowerCase()` и т. д.

ЗАМЕЧАНИЕ

Шаблон в строке форматирования имеет следующую форму (табл. П2.2):
`"%" [index ":"] ["-"] [width] ["." prec] type`.

Таблица П2.2. *Шаблоны форматирования для функции* `Format()`

Спецификатор	Описание
`%`	Обязательный признак начала спецификатора формата
`[index ":"]`	Индекс аргумента
`["-"]`	Выравнивать влево
`[width]`	Ширина
`["." prec]`	Точность представления вещественного числа
`type`	Символ типа преобразования (табл. П2.3)

Таблица П2.3. *Символы типа преобразования функции* `Format()`

Тип	Описание
`d`	**Десятичный тип.** Аргумент должен быть целым числом. `s:=Format('Целое число %d',[15]); //Целое число 15`
`u`	**Десятичный тип без знака.** Аналогичен типу `d`, но используется только с неотрицательными числами

Таблица П2.3 *(окончание)*

Тип	Описание

e	**Научный тип.** Аргумент должен быть числом с плавающей точкой. Аргумент преобразуется в строку вида "–d.ddd...E+ddd" (характеристика и мантисса). Точность (по умолчанию) составляет 15 знаков после точки.
	`s:=Format('1/3= %e',[1/3]); //1/3=3.3333333333333333E-01`
	Допустимо изменить точность с помощью поля `prec`:
	`s:=Format('1/3=%1.4e',[1/3]); // 1/3=3.333E-01`

f	**Фиксированный тип.** Аргумент должен быть числом с плавающей точкой. Значение аргумента преобразуется к виду "–ddd.ddd...". По умолчанию округление производится до второго знака после точки. `s:=Format('1/3=%f',[1/3]); // 1/3=0.33`
g	**Основной тип.** Аргумент должен быть числом с плавающей точкой. Задача основного типа заключается в преобразовании значения аргумента в самую короткую текстовую строку. Исходя из этого критерия, будет использован фиксированный или научный формат
n	**Числовой тип.** Аргумент должен быть числом с плавающей точкой. Значение аргумента преобразуется к виду "–d ddd,ddd ddd...". Между тысячами ставится разделитель. По умолчанию округление производится до второго знака после точки. `s:=Format('1E3*1E3=%n',[1E3*1E3]);//1E3*1E3=1 000 000.00`
m	**Денежный тип.** Аргумент должен быть числом с плавающей точкой. Значение аргумента преобразуется в текстовую строку в виде денежной суммы. Вид результата определяется значениями полей `CurrencyString`, `CurrencyFormat`, `NegCurrFormat`, `ThousandSeparator`, `DecimalSeparator` и `CurrencyDecimals` (табл. П2.5). По умолчанию округление производится до второго знака после точки. `s:=Format('Сумма = %m',[251.616+345.44]);//Сумма = 597.06р.`
p	**Указатель.** Аргумент должен быть указателем. Значение аргумента конвертируется в 8-символьную текстовую строку шестнадцатеричных цифр
s	**Строковый тип.** Тип аргумента: `string`, `PChar` или `Char`. Допускается применять совместно с полем `prec`, определяющим максимальную длину строки. `s:=Format('%.5s',['0123456789ABCDE']); // 01234`
x	**Шестнадцатеричный тип.** Аргумент должен быть целым числом. Значение аргумента преобразуется в строку шестнадцатеричных цифр. `s:=Format('24=%x',[26]); // 24=1A`

Таблица П2.4. *Спецификаторы формата функций* `Format()` *и* `FormatCurr()`

Спецификатор	Описание
0	**Обязательное цифровое поле.** В случае если преобразуемая величина имеет в этой позиции цифру, то ставится цифра. Иначе в поле будет вставлен "0" . `s:=FormatFloat('0.0',9.1989); // 9.2`
#	**Необязательное цифровое поле.** В случае если преобразуемая величина имеет в этой позиции цифру, то ставится цифра. Иначе в поле останется пустым. `s:=FormatFloat('0.0#####',9.1989);//9.1989`
.	**Десятичная точка.** Разделитель целой и дробной частей числа. Символ разделителя определяется в поле `DecimalSeparator` (табл. П2.5)
,	**Разделитель тысяч.** Разделяет число на группы по три символа, влево от десятичной точки. Символ разделителя определяется в поле `ThousandSeparator` (табл. П2.5). `s:=FormatFloat('#,##0.0',91989.111);// 91,989.1`

***Таблица П2.4** (окончание)*

Спецификатор	Описание

E+, E-, e+, e-	Научный формат. Число преобразуется в научный формат. Количество нулей после спецификатора определяет ширину мантиссы:
	`s:=FormatFloat('0.0##E+00',91989.111);//9,199E+04`
	`s:=FormatFloat('0.0##E-00',91989.111); //9,199E04`

'xx'/"xx"	**Символы пользователя.** В кавычки заключаются символы пользователя `s:=FormatFloat('"Число="0.0',9.198);//Число=9.2`
;	Разделитель представления положительных, отрицательных и нулевых чисел. Например: `#,##0.00;(#,##0.00)`

Системные настройки форматирования и класс *TFormatSettings*

Операционная система Windows разрешает пользователю осуществлять тонкую настройку формата представления чисел, денежных величин, даты и времени. Для этого требуется обратиться к апплету **Язык и региональные стандарты** в Панели управления компьютером. Единожды внеся изменения в настройки по умолчанию, мы сразу повлияем на поведение всех приложений, работающих под управлением Windows.

В модуле `SysUtils` объявлен класс `TFormatSettings`, поля которого специализируются на хранении текущих системных настроек по представлению денежных величин, значений даты/времени и т. п. Перечисленные в табл. П2.5 переменные инициализируются автоматически согласно настройкам операционной системы.

> ***Внимание!***
>
> В устаревших версиях Delphi перечисленные в таблице поля были объявлены как самостоятельные глобальные переменные. При разработке проекта в современных версиях Delphi вместо обращения к глобальной переменной следует воспользоваться услугами глобального объекта `FormatSettings:TFormatSettings`, например: `s:=FormatSettings.CurrencyString`.

***Таблица П2.5.** Поля класса `TFormatSettings`*

Поле класса `TFormatSettings`	Описание
`CurrencyString: string;`	Обозначение денежной единицы
`CurrencyFormat: Byte;`	Формат представления положительной денежной величины
`NegCurrFormat: Byte;`	Представление отрицательной денежной величины
`ThousandSeparator: Char;`	Разделитель групп разрядов
`DecimalSeparator: Char;`	Разделитель дробной и целой части числа
`CurrencyDecimals: Byte;`	Количество дробных знаков
`DateSeparator: Char;`	Разделитель даты
`ShortDateFormat: string;`	Краткий формат даты: "`dd.MM.yyyy`"

Таблица П2.5 *(окончание)*

Поле класса `TFormatSettings`	**Описание**
`LongDateFormat: string;`	Полный формат даты: "`d MMMM yyyy 'г.'`"
`TimeSeparator: Char;`	Разделитель времени
`TimeAMString: string;`	Обозначение времени до полудня
`TimePMString: string;`	Обозначение времени после полудня
`ShortTimeFormat: string;`	Краткий формат времени: "`h:mm`"
`LongTimeFormat: string;`	Полный формат времени: "`h:mm:ss`"
`ShortMonthNames: array[1..12] of string;`	Массив с сокращенными названиями месяцев года
`LongMonthNames: array[1..12] of string;`	Массив с полными названиями месяцев года
`ShortDayNames: array[1..7] of string;`	Массив с сокращенными названиями дня недели
`LongDayNames: array[1..7] of string;`	Массив с полными названиями дней недели
`SysLocale: TSysLocale;`	Структура, содержащая региональные параметры операционной системы
`TwoDigitYearCenturyWindow: Word = 50;`	Определяет порядок преобразования двух последних символов года в полный год
`ListSeparator: Char;`	Разделитель элементов списка

ПРИЛОЖЕНИЕ 3

Работа с датой и временем

В табл. П3.1 приведены основные процедуры и функции для работы с датой и временем. Эти методы объявлены в модулях `SysUtils` и `DateUtils`.

Таблица П3.1. *Процедуры и функции по работе с датой и временем из модуля* `SysUtils`

Функция/процедура	Описание
Выяснение системной даты/времени и управление датой/временем	
`function Now : TDateTime;`	Возвращает текущую дату и время
`function Date: TDateTime;`	Возвращает текущую дату
`function Time : TDateTime;`	Возвращает текущее время
`function CurrentYear: Word;`	Текущий год
`function DayOfWeek(Date: TDateTime): Integer;`	Возвращает день недели. Первым днем недели является воскресенье (значение 1), последним — суббота (значение 7)
`function IncMonth(const AValue: TDateTime; NumberOfMonths: Integer): TDateTime;`	Осуществляет приращение даты (`AValue`) на количество месяцев, определенное в параметре `NumberOfMonths`
`procedure IncAMonth(var Year, Month, Day: Word; NumberOfMonths: Integer = 1);`	Осуществляет приращение значения даты на число месяцев, определенное в параметре `NumberOfMonths`, и декодирует результат в номер года, месяца и дня
`procedure ReplaceTime(var DateTime: TDateTime; const NewTime: TDateTime);`	Заменяет время в переменной `DateTime` новым значением `NewTime`. Дата в `DateTime` остается неизменной
`procedure ReplaceDate(var DateTime: TDateTime; const NewDate: TDateTime);`	Заменяет дату в переменной `DateTime` новым значением `NewDate`. Время в `DateTime` остается неизменным
`function IsLeapYear(Year: Word): Boolean;`	Проверка на високосный год
Преобразование даты/времени в числовые значения	
`function TimeStampToDateTime(const TimeStamp: TTimeStamp): TDateTime;`	Конвертирует дату/время из формата `TTimeStamp` в `TDateTime`
`function DateTimeToTimeStamp (DateTime: TDateTime): TTimeStamp;`	Конвертирует дату/время из формата `TDateTime` в `TTimeStamp`

Таблица П3.1 *(окончание)*

Функция/процедура	Описание
`function MSecsToTimeStamp(MSecs: Comp): TTimeStamp;`	Преобразует число миллисекунд к типу данных `TTimeStamp`
`function TimeStampToMSecs(const TimeStamp: TTimeStamp): Comp;`	Преобразует значение `TTimeStamp` в число миллисекунд
`function EncodeDate(Year, Month, Day: Word): TDateTime;`	Конвертирует год, месяц и день в дату в формат `TDateTime`
`function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;`	Кодирует часы, минуты, секунды и миллисекунды во время в формате `TDateTime`
`function TryEncodeDate(Year, Month, Day: Word; out Date: TDateTime): Boolean;`	Перед преобразованием года, месяца и даты в формат `TDateTime` контролирует корректность значений в параметрах `Year`, `Month` и `Day`. Если в параметре ошибка (например, указан 13-й месяц), функция ввернет `false`
`function TryEncodeTime(Hour, Min, Sec, MSec: Word; out Time: TDateTime): Boolean;`	Перед преобразованием числовых значений часа, минуты, секунды и миллисекунды в формат `TDataTime` проверяет корректность этих значений. В случае ошибки функция вернет `false`, в случае удачи — `true`, а результат преобразования внесет в параметр `Time`
`procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);`	Конвертирует дату в формате `TDateTime` в год, месяц и день
`function DecodeDateFully(const DateTime: TDateTime; var Year, Month, Day, DOW: Word): Boolean;`	Полное декодирование даты (включая день недели `DOW`)
`procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);`	Конвертирует время в формате `TDateTime` в часы, минуты, секунды и миллисекунды
`procedure DateTimeToSystemTime(DateTime: TDateTime; var SystemTime: TSystemTime);`	Конвертирует дату/время из формата `TDateTime` в формат, понятный Windows
`function SystemTimeToDateTime(const SystemTime: TSystemTime): TDateTime;`	Конвертирует системные дату/время (формат Windows) в формат `TDateTime`
`function TrySystemTimeToDateTime(const SystemTime: TSystemTime; out DateTime: TDateTime): Boolean;`	То же самое, что `SystemTimeToDateTime`, но перед началом преобразования проверяется корректность аргументов

Таблица П3.2. *Процедуры и функции по работе с датой/временем из модуля* `DateUtils`

Функция/процедура	Описание
Основные процедуры и функции	
`function IsInLeapYear(const AValue: TDateTime): Boolean;`	Проверка на високосный год
`function IsAM(const AValue: TDateTime): Boolean;`	Значение времени находится в пределах от полуночи до 12 часов дня
`function IsPM(const AValue: TDateTime): Boolean;`	Значение времени находится в пределах от 12 часов после полудня до полуночи

Таблица П3.2 *(продолжение)*

Функция/процедура	Описание
function IsValidDate(**const** AYear, AMonth, ADay: Word): Boolean;	Проверяет, чтобы значения аргументов года, месяца и дня были корректны, в случае успеха возвратит true
function IsValidTime(**const** AHour, AMinute, ASecond, AMilliSecond: Word): Boolean;	Проверяет, чтобы значения аргументов (часы, минуты и секунды) были корректны, в случае успеха возвратит true
function IsValidDateTime(**const** AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word): Boolean;	Проверяет корректность аргументов (год, месяц, день, час и т. д.), в случае успеха возвратит true
function IsValidDateDay(**const** AYear, ADayOfYear: Word): Boolean;	Проверяет факт принадлежности дня ADayOfYear году AYear, в случае успеха возвратит true
function IsValidDateWeek(**const** AYear, AWeekOfYear, ADayOfWeek: Word): Boolean;	Проверяет факт принадлежности недели AWeekOfYear году AYear, в случае успеха возвратит true
function IsValidDateMonthWeek(**const** AYear, AMonth, AWeekOfMonth, ADayOfWeek: Word): Boolean;	Проверяет корректность аргументов (год, месяц, номер недели в месяце, день недели и т. д.), в случае успеха возвратит true
function WeeksInYear(**const** AValue: TDateTime): Word;	Возвратит число недель в году в соответствии со стандартом ISO 8601 (первой неделей года считается неделя, содержащая первый четверг года)
function WeeksInAYear(**const** AYear: Word): Word;	
function DaysInYear(**const** AValue: TDateTime): Word; **function** DaysInAYear(**const** AYear: Word): Word;	Возвратит число дней в году
function DaysInMonth(**const** AValue: TDateTime): Word; **function** DaysInAMonth(**const** AYear, AMonth: Word): Word;	Возвратит число дней в месяце
function Today: TDateTime;	Сегодня
function Yesterday: TDateTime;	Вчера
function Tomorrow: TDateTime;	Завтра
function IsToday(**const** AValue: TDateTime): Boolean;	Проверяет, является ли значение AValue сегодняшней датой
function IsSameDay(**const** AValue, ABasis: TDateTime): Boolean;	Проверяет, являются ли значения AValue и ABasis одним и тем же днем
Кодирование и декодирование значения даты/времени	
function EncodeDateTime(**const** AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word): TDateTime;	Формирует значение даты/времени по году, месяцу, дню, часу, минуте, секунде и миллисекунде
procedure DecodeDateTime(**const** AValue: TDateTime; out AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word);	Декодирует значение даты/времени, возвращая год, месяц, день, час, минуту, секунду и миллисекунду

Таблица П3.2 *(продолжение)*

Функция/процедура	Описание
function EncodeDateDay(**const** AYear, ADayOfYear: Word): TDateTime;	Формирует значение даты/времени по году и номеру дня в году
procedure DecodeDateDay(**const** AValue: TDateTime; out AYear, ADayOfYear: Word);	Декодирует дату/время, возвращая год и номер дня в году
function DateOf(const AValue: TDateTime): TDateTime;	Возвращает только значение даты
function TimeOf(const AValue: TDateTime): TDateTime;	Возвращает только значение времени
function DayOfWeek(Date: TDateTime): Integer;	Возвращает день недели. Первым днем недели является воскресенье (значение 1), последним — суббота (значение 7)
function DayOfTheWeekconst AValue: TDateTime): Word;	Возвращает день недели. Но в этом случае первым днем недели является понедельник (значение 1), а последним — воскресенье (значение 7)
function YearOf(const AValue: TDateTime): Word;	Возвращает год из даты AValue
function MonthOf(const AValue: TDateTime): Word;	Возвращает месяц из даты AValue
function WeekOf(const AValue: TDateTime): Word;	Возвращает номер недели из даты AValue
function DayOf(const AValue: TDateTime): Word;	Возвращает день из даты AValue
function HourOf(const AValue: TDateTime): Word;	Возвращает часы из времени AValue
function MinuteOf(**const** AValue: TDateTime): Word;	Возвращает минуты из времени AValue
function SecondOf(**const** AValue: TDateTime): Word;	Возвращает секунды из времени AValue
function MilliSecondOf(**const** AValue: TDateTime): Word;	Возвращает миллисекунды из времени AValue
function DateOfconst AValue: TDateTime): TDateTime;	Возвращает только значение даты (отрезая дробную часть от AValue)
function TimeOfconst AValue: TDateTime): TDateTime;	Устанавливает целую часть AValue в ноль
function MonthOfTheYear(**const** AValue: TDateTime): Word; **inline**;	Декодирует параметр AValue и возвращает номер месяца с начала года
function WeekOfTheYear(**const** AValue: TDateTime): Word; **overload**; **function** WeekOfTheYear(**const** AValue: TDateTime; **var** AYear: Word): Word; **overload**;	Декодирует параметр AValue и возвращает номер недели с начала года
function DayOfTheYear(**const** AValue: TDateTime): Word;	Декодирует параметр AValue и возвращает номер дня с начала года
function HourOfTheYear(**const** AValue: TDateTime): Word;	Декодирует параметр AValue и возвращает число часов с начала года

***Таблица П3.2** (продолжение)*

Функция/процедура	Описание
`function MinuteOfTheYear(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает число минут с начала года
`function SecondOfTheYear(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает число секунд с начала года
`function MilliSecondOfTheYear(const AValue: TDateTime): Int64;`	Декодирует параметр `AValue` и возвращает число миллисекунд с начала года
`function WeekOfTheMonth(const AValue: TDateTime): Word; overload;` `function WeekOfTheMonth(const AValue: TDateTime; var AYear, AMonth: Word): Word; overload;`	Декодирует параметр `AValue` и возвращает номер недели с начала месяца
`function DayOfTheMonth(const AValue: TDateTime): Word; inline;`	Декодирует параметр `AValue` и возвращает номер дня с начала месяца
`function HourOfTheMonth(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает число часов с начала месяца
`function MinuteOfTheMonth(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает число минут с начала месяца
`function SecondOfTheMonth(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает число секунд с начала месяца
`function MilliSecondOfTheMonth(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает число миллисекунд с начала месяца
`function DayOfTheWeek(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает номер дня с начала недели
`function HourOfTheWeek(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает количество часов с начала недели
`function MinuteOfTheWeek(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает количество минут с начала недели
`function SecondOfTheWeek(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает количество секунд с начала недели
`function MilliSecondOfTheWeek(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает количество миллисекунд с начала недели
`function HourOfTheDay(const AValue: TDateTime): Word; inline;`	Декодирует параметр `AValue` и возвращает количество часов относительно начала суток
`function MinuteOfTheDay(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает количество минут относительно начала суток
`function SecondOfTheDay(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает количество секунд относительно начала суток
`function MilliSecondOfTheDay(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает количество миллисекунд относительно начала суток
`function MinuteOfTheHour(const AValue: TDateTime): Word; inline;`	Декодирует параметр `AValue` и возвращает количество минут относительно начала часа
`function SecondOfTheHour(const AValue: TDateTime): Word;`	Декодирует параметр `AValue` и возвращает количество секунд относительно начала часа

***Таблица П3.2** (продолжение)*

Функция/процедура	Описание
`function MilliSecondOfTheHour(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает количество миллисекунд относительно начала часа
`function SecondOfTheMinute(const AValue: TDateTime): Word; inline;`	Декодирует параметр `AValue` и возвращает количество секунд относительно начала минуты
`function MilliSecondOfTheMinute(const AValue: TDateTime): LongWord;`	Декодирует параметр `AValue` и возвращает количество миллисекунд относительно начала минуты
`function MilliSecondOfTheSecond(const AValue: TDateTime): Word; inline;`	Декодирует параметр `AValue` и возвращает количество миллисекунд относительно начала секунды
Приращение значения даты/времени	
`function IncYear(const AValue: TDateTime; const ANumberOfYears: Integer = 1): TDateTime; inline;`	Приращение года (параметр `AValue`) на некоторое число из параметра `ANumberOfYears`
`function IncWeek(const AValue: TDateTime; const ANumberOfWeeks: Integer = 1): TDateTime; inline;`	Приращение недели
`function IncDay(const AValue: TDateTime; const ANumberOfDays: Integer = 1): TDateTime;`	Приращение дня
`function IncHour(const AValue: TDateTime; const ANumberOfHours: Int64 = 1): TDateTime;`	Приращение часов
`function IncMinute(const AValue: TDateTime; const ANumberOfMinutes: Int64 = 1): TDateTime;`	Приращение минут
`function IncSecond(const AValue: TDateTime; const ANumberOfSeconds: Int64 = 1): TDateTime;`	Приращение секунд
`function IncMilliSecond(const AValue: TDateTime; const ANumberOfMilliSeconds: Int64 = 1): TDateTime;`	Приращение миллисекунд
Сравнение двух значений даты/времени	

`type TValueRelationship = -1..1;`
`const`
`  LessThanValue = Low(TValueRelationship);`
`  EqualsValue = 0;`
`  GreaterThanValue = High(TValueRelationship);`
	Тип данных `TValueRelationship` применяется в операциях сравнения двух величин:
	• `LessThanValue`, если `A<B`;
	• `EqualsValue`, если `A=B`;
	• `GreaterThanValue`, если `A>B`

`function CompareDateTime(const A, B: TDateTime): TValueRelationship;`	Сравнение даты/времени с возвратом результата в виде `TValueRelationship`
`function SameDateTime(const A, B: TDateTime): Boolean;`	При равенстве даты/времени возвратит `true`
`function CompareDate(const A, B: TDateTime): TValueRelationship;`	Сравнение двух дат с возвратом результата в виде `TValueRelationship`

Таблица П3.2 *(продолжение)*

Функция/процедура	Описание
`function SameDate(const A, B: TDateTime): Boolean;`	При равенстве даты возвратит `true`
`function CompareTime(const A, B: TDateTime): TValueRelationship;`	Сравнение двух значений времени возвратом результата в виде `TValueRelationship`
`function SameTime(const A, B: TDateTime): Boolean;`	При равенстве двух значений времени возвратит `true`
Точный интервал между двумя значениями даты/времени	
`function MilliSecondsBetween(const ANow, AThen: TDateTime): Int64;`	Количество миллисекунд, содержащихся в интервале времени от `ANow` до `AThen`
`function SecondsBetween(const ANow, AThen: TDateTime): Int64;`	Количество секунд, содержащихся в интервале времени от `ANow` до `AThen`. Напомню, что в одной секунде — 1000 миллисекунд
`function MinutesBetween(const ANow, AThen: TDateTime): Int64;`	Число минут в интервале времени от `ANow` до `AThen`. Неполные минуты (даже если до завершения минуты не хватает миллисекунды) в счет не идут
`function HoursBetween(const ANow, AThen: TDateTime): Int64;`	Количество часов в заданном интервале
`function DaysBetween(const ANow, AThen: TDateTime): Integer;`	Количество дней между датами `ANow` и `AThen`
`function WeeksBetween(const ANow, AThen: TDateTime): Integer;`	Приблизительное число недель между датами. Неполные недели (даже если до завершения недели не хватит одной секунды) в счет не идут
`function MonthsBetween(const ANow, AThen: TDateTime): Integer;`	Приблизительное число месяцев между датами. Метод считает, что в месяце содержится 30.4375 дней
`function YearsBetween(const ANow, AThen: TDateTime): Integer;`	Возвращает приблизительное количество лет, разделяющих даты `ANow` и `AThen`. Причем функция считает, что в году 365.25 дней
Приблизительный интервал между двумя значениями даты/времени	
`function YearSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число лет между значениями `ANow` и `AThen`
`function MonthSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число месяцев между значениями `ANow` и `AThen`
`function WeekSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число недель между значениями `ANow` и `AThen`
`function DaySpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число дней между значениями `ANow` и `AThen`
`function HourSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число часов между значениями `ANow` и `AThen`
`function MinuteSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число минут между значениями `ANow` и `AThen`
`function SecondSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число секунд между значениями `ANow` и `AThen`

***Таблица П3.2** (продолжение)*

Функция/процедура	Описание
`function MilliSecondSpan(const ANow, AThen: TDateTime): Double;`	Приблизительное число миллисекунд между значениями `ANow` и `AThen`
Проверка вхождения в заданный диапазон	
`function WithinPastYears(const ANow, AThen: TDateTime; const AYears: Integer): Boolean;`	Возвращает `true`, если разница между датами `ANow` и `AThen` не превышает `AYEars` лет
`function WithinPastMonths(const ANow, AThen: TDateTime; const AMonths: Integer): Boolean;`	Возвращает `true`, если разница между датами `ANow` и `AThen` не превышает `AMonths` месяцев
`function WithinPastWeeks(const ANow, AThen: TDateTime;const AWeeks: Integer): Boolean;`	Возвращает `true`, если разница между датами `ANow` и `AThen` не превышает `AWeeks` недель
`function WithinPastDays(const ANow, AThen: TDateTime; const ADays: Integer): Boolean;`	Возвращает `true`, если разница между датами `ANow` и `AThen` не превышает `ADays` дней
`function WithinPastHours(const ANow, AThen: TDateTime; const AHours: Int64): Boolean;`	Возвращает `true`, если разница между значениями `ANow` и `AThen` не превышает `AHours` часов
`function WithinPastMinutes(const ANow, AThen: TDateTime; const AMinutes: Int64): Boolean;`	Возвращает `true`, если разница между значениями `ANow` и `AThen` не превышает `AMinutes` минут
`function WithinPastSeconds(const ANow, AThen: TDateTime; const ASeconds: Int64): Boolean;`	Возвращает `true`, если разница между значениями `ANow` и `AThen` не превышает `ASeconds` секунд
`function WithinPastMilliSeconds(const ANow, AThen: TDateTime; const AMilliSeconds: Int64): Boolean;`	Возвращает `true`, если разница между значениями `ANow` и `AThen` не превышает `AMilliSeconds` миллисекунд
Начало/окончание периода	
`function StartOfTheYear(const AValue: TDateTime): TDateTime;` `function StartOfAYear(const AYear: Word): TDateTime;`	Возвращает дату начала года, значение времени устанавливается в `00:00:00.000`
`function EndOfTheYear(const AValue: TDateTime): TDateTime;` `function EndOfAYear(const AYear: Word): TDateTime;`	Возвращает дату окончания года, значение времени устанавливается в `23:59:59.999`
`function StartOfTheMonth(const AValue: TDateTime): TDateTime;` `function StartOfAMonth(const AYear, AMonth: Word): TDateTime;`	Возвращает дату начала месяца, значение времени устанавливается в `00:00:00.000`
`function EndOfTheMonth(const AValue: TDateTime): TDateTime;` `function EndOfAMonth(const AYear, AMonth: Word): TDateTime;`	Возвращает дату окончания месяца, значение времени устанавливается в `23:59:59.999`
`function StartOfTheWeek(const AValue: TDateTime): TDateTime;` `function StartOfAWeek(const AYear, AWeekOfYear: Word; const ADayOfWeek: Word = 1): TDateTime;`	Возвращает дату начала недели, значение времени устанавливается в `00:00:00.000`

Таблица П3.2 *(продолжение)*

Функция/процедура	Описание
function EndOfTheWeek(**const** AValue: TDateTime): TDateTime; **function** EndOfAWeek(**const** AYear, AWeekOfYear: Word; **const** ADayOfWeek: Word = 7): TDateTime;	Возвращает дату окончания недели, значение времени устанавливается в 23:59:59.999
function StartOfTheDay(**const** AValue: TDateTime): TDateTime; **function** StartOfADay(**const** AYear, AMonth, ADay: Word): TDateTime; **function** StartOfADay(**const** AYear, ADayOfYear: Word): TDateTime;	Возвращает начало суток, значение времени устанавливается в 00:00:00.000
function EndOfTheDay(**const** AValue: TDateTime): TDateTime; **function** EndOfADay(**const** AYear, AMonth, ADay: Word): TDateTime; **function** EndOfADay(**const** AYear, ADayOfYear: Word): TDateTime; overload;	Возвращает окончание суток, значение времени устанавливается в 23:59:59.999
Замена части значения даты/времени	
function RecodeYear(**const** AValue: TDateTime; **const** AYear: Word): TDateTime;	Заменит год в значении даты/времени (определенном в параметре AValue) новым из параметра AYear и возвратит новое значение
function RecodeMonth(**const** AValue: TDateTime; **const** AMonth: Word): TDateTime;	Заменит месяц в значении даты/времени (определенном в параметре AValue) новым из параметра AMonth и возвратит новое значение
function RecodeDay(**const** AValue: TDateTime; **const** ADay: Word): TDateTime;	Заменит день в значении даты/времени (определенном в параметре AValue) новым из параметра ADay и возвратит новое значение
function RecodeHour(**const** AValue: TDateTime; **const** AHour: Word): TDateTime;	Заменит часы в значении даты/времени (определенном в параметре AValue) новым значением из параметра AHour и возвратит новое значение
function RecodeMinute(**const** AValue: TDateTime; **const** AMinute: Word): TDateTime;	Заменит минуты в значении даты/времени (определенным в параметре AValue) новым значением из параметра AMinute и возвратит новое значение
function RecodeSecond(**const** AValue: TDateTime; **const** ASecond: Word): TDateTime;	Заменит секунды в значении даты/времени (определенном в параметре AValue) новым значением из параметра ASecond и возвратит новое значение
function RecodeMilliSecond(**const** AValue: TDateTime; **const** AMilliSecond: Word): TDateTime;	Заменит миллисекунды в значении даты/времени (определенном в параметре AValue) новым значением из параметра AMilliSecond и возвратит новое значение
function RecodeDate(**const** AValue: TDateTime; **const** AYear, AMonth, ADay: Word): TDateTime;	Заменит дату в значении даты/времени (определенном в параметре AValue) новым значением из параметров AYear, AMonth, ADay и возвратит новое значение

Таблица П3.2 *(окончание)*

Функция/процедура	Описание
`function RecodeTime(const AValue: TDateTime; const AHour, AMinute, ASecond, AMilliSecond: Word): TDateTime;`	Заменит время в значении даты/времени (определенном в параметре `AValue`) новым значением из параметров `AHour`, `AMinute`, `ASecond` и возвратит новое значение
`function RecodeDateTime(const AValue: TDateTime; const AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word): TDateTime;`	Сформирует новое значение даты/времени
Преобразование даты/времени в другие форматы	
`function FileDateToDateTime(FileDate: Integer): TDateTime;`	Преобразует дату/время файла Windows в формат `TDateTime`
`function DateTimeToFileDate (DateTime: TDateTime): Integer;`	Преобразует дату/время в формате `TDateTime` в формат, приемлемый файлом Windows
`function FloatToDateTime(const Value: Extended): TDateTime;`	Конвертирование вещественного числа в дату/время
`function EncodeDateTime(const AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word):TDateTime;`	Объединение методов `EncodeDate()` и `EncodeTime()`
`procedure DecodeDateTime(const AValue: TDateTime; out AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word);`	Объединение методов `DecodeDate()` и `DecodeTime()`, одновременно конвертирует дату и время
`procedure DecodeDateMonthWeek(const AValue: TDateTime; out AYear, AMonth, AWeekOfMonth, ADayOfWeek: Word);`	Декодирует дату `AValue` в год — `AYear`, месяц — `AMonth`, порядковый номер недели в месяце — `AWeekOfMonth` и день недели — `DayOfWeek`
`function EncodeDateMonthWeek(const AYear, AMonth, AWeekOfMonth: Word; const ADayOfWeek: Word = 1): TDateTime;`	Функция осуществляет преобразование года, месяца, номера недели в месяце и дня недели в дату
Взаимодействие с ОС UNIX	
`function DateTimeToUnix(const AValue: TDateTime): Int64;`	Перевод даты/времени в формат UNIX
`function UnixToDateTime(const AValue: Int64): TDateTime;`	Получение даты/времени из формата UNIX

Представление даты и времени в текстовом формате

Таблица П3.3. *Функции и процедуры преобразования даты/времени и строки* `String`

Функция/процедура	Описание
`function DateTimeToStr(DateTime: TDateTime): string;`	Конвертирует дату/время из формата `TDateTime` в текстовую строку
`function DateToStr(Date: TDateTime): string;`	Конвертирует дату из формата `TDateTime` в текстовую строку

Таблица П3.3 *(окончание)*

Функция/процедура	Описание
`function TimeToStr(Time: TDateTime): string;`	Конвертирует время из формата `TDateTime` в текстовую строку
`function StrToDateTime(const S: string): TDateTime;`	Конвертирует дату/время из текста в формат `TDateTime`
`function StrToDate(const S: string): TDateTime;`	Конвертирует дату из текста в формат `TDateTime`
`function StrToTime(const S: string): TDateTime;`	Конвертирует время из текста в формат `TDateTime`
`function FormatDateTime(const Format: string; DateTime: TDateTime): string;` `procedure DateTimeToString(var Result: string; const Format: string; DateTime: TDateTime);`	Преобразование даты/времени в текст с расширенными возможностями форматирования (см. табл. П3.4). При преобразовании текстовых строк учитывайте, что в зависимости от настроек компьютера дата и время могут иметь европейский или американский формат, разделяться различными символами-сепараторами, иметь краткое или полное представление

Таблица П3.4. *Спецификаторы форматирования (аргумент `Format`) функций `FormatDateTime()` и `DateTimeToString()`*

Спецификатор	Формат представления данных
`c`	Дата в формате `ShortDateFormat`, далее (если дробная часть `DateTime` не равна нулю) отображается время в формате `LongTimeFormat`
`d`	Число месяца без нуля в первом разряде (1—31)
`dd`	Число месяца с нулем в первом разряде (01—31)
`ddd`	Сокращенное название дня недели в соответствии с именами из `ShortDayNames`
`dddd`	Полное название дня недели в соответствии с именами из `LongDayNames`
`ddddd`	Дата в сокращенном формате `ShortDateFormat`
`dddddd`	Дата в полном формате `LongDateFormat`
`m`	Месяц в виде числа без нуля в первом разряде (1—12)
`mm`	Месяц в виде числа с нулем в первом разряде (01—12)
`mmm`	Сокращенное название месяца в формате `ShortMonthNames`
`mmmm`	Полное название месяца в соответствии с `LongMonthNames`
`yy`	Год из двух цифр (00—99)
`yyyy`	Год из четырех цифр (0000—9999)
`h`	Часы без нуля в первом разряде (0—23)
`hh`	Часы с нулем в первом разряде (00—23)
`n`	Минуты без нуля в первом разряде (0—59)
`nn`	Минуты с нулем в первом разряде (00—59)
`s`	Секунды без нуля в первом разряде (0—59)

Таблица П3.4 *(окончание)*

Спецификатор	Формат представления данных
ss	Секунды с нулем в первом разряде (00—59)
z	Миллисекунды без нуля в первом разряде (0—999)
zzz	Миллисекунды с нулем в первом разряде (000—999)
t	Сокращенный формат времени в соответствии с ShortTimeFormat
tt	Полный формат времени в соответствии с LongTimeFormat
am/pm	Время в 12-часовом формате. До полудня показываются символы "am", после — "pm"
a/p	Время в 12-часовом формате. До полудня показывается символ "a", после — "p"
ampm	Время в 12-часовом формате. Символы, показываемые до и после полудня, определяются соответственно значениями TimeAMString и TimePMString
/	Разделитель даты, определенный в DateSeparator
:	Разделитель времени, определенный в TimeSeparator
'xx'/"xx"	Символы-комментарии. Заключаются в двойные кавычки или одинарные кавычки

ПРИЛОЖЕНИЕ 4

Работа с памятью

Основные функции и процедуры, предназначенные для управления памятью, сосредоточены в модулях `System`, `SysUtils` и `ShareMem` (табл. П4.1).

Таблица П4.1. *Работа с памятью*

Функция/процедура	Описание
`procedure New(var P: Pointer);`	Создает новую динамическую переменную и устанавливает указатель на нее
`procedure Dispose(var P: Pointer);`	Очищает блок памяти, выделенный под динамическую переменную
`procedure GetMem(var P: Pointer; Size: Integer);`	Создает динамическую переменную и указатель `P` на блок памяти
`procedure FreeMem(var P: Pointer[; Size: Integer]);`	Освобождает блок памяти, связанный с указателем `P`
`function AllocMem(Size: Cardinal): Pointer;`	Выделяет блок размером `Size` в куче памяти, обнуляет его и возвращает указатель
`function GetHeapStatus: THeapStatus;`	Возвращает текущий статус менеджера памяти. Данные будут размещены в записи `HeapStatus`
`procedure GetMemoryManager(var MemMgr: TMemoryManager);`	Укажет на точки входа действующего в настоящее время менеджера памяти. Информация окажется в `MemMgr`
`procedure SetMemoryManager(const MemMgr: TMemoryManager);`	Устанавливает точки входа функциям, инкапсулированным в менеджере памяти. Изменения окажут влияние на поведение стандартных процедур: `GetMem()`, `ReallocMem()` и `FreeMem()`
`function IsMemoryManagerSet : Boolean;`	Указывает, был ли изменен стандартный менеджер памяти с помощью функции `SetMemoryManager()`
`procedure ReallocMem(var P: Pointer; Size: Integer);`	Перераспределяет динамическую переменную, распределенную ранее функциями `GetMem()`, `AllocMem()` или `ReallocMem()`
`function SysFreeMem(P: Pointer): Integer;`	Освобождает блок памяти, на который указывает `Pointer`
`function SysGetMem (Size: Integer): Pointer;`	Выделяет `Size` байтов и возвращает указатель на них. Если вы не используете модуль `ShareMem`, то блок выделяется в глобальной куче

Таблица П4.1 *(окончание)*

Функция/процедура	Описание
`function SysReallocMem(P: Pointer; Size: Integer): Pointer;`	Возвращает указатель на `Size` байтов, при этом значение `P` сохраняется
`function CompareMem(P1, P2: Pointer; Length: Integer): Boolean;`	Сравнивает содержимое двух блоков памяти по адресу `P1` и `P2` длиной `Length` байтов, возвращает `true` в случае полного соответствия
`function Assigned(var P): Boolean;`	Тест на наличие объекта по указателю `P`. Если по заданному адресу объект отсутствует (`nil`), то возвращается значение `false`
`procedure FreeAndNil(var Obj);`	Освобождает объектную ссылку и заменяет ссылку нулем. Данную процедуру можно использовать только с `TObject` и его потомками
`function SizeOf(X): Integer;`	Возвращает количество байтов, занимаемых `X`

Замечание

Модуль `ShareMem` обеспечивает подключение менеджера памяти, позволяющего разделять память между двумя и более процессами. Этот менеджер содержится в динамической библиотеке borlndmm.dll. Учитывайте это при установке приложений на другие компьютеры, т. к. без этой библиотеки приложение, обращающееся к модулю `ShareMem`, не сможет работать.

ПРИЛОЖЕНИЕ 5

Управление ходом выполнения программы

Ключевые процедуры, предназначенные для управления ходом программы, представлены в модуле `System` (табл. П5.1). В своем большинстве они принудительно прекращают выполнение циклов, методов или даже программы.

Таблица П5.1. *Управление ходом выполнения программы*

Процедура	Описание
procedure `Abort;`	Экстренное прекращение выполнения метода без вызова исключительной ситуации
procedure `Break;`	Прекращает выполнение цикла
procedure `Continue;`	Применяется внутри циклов, принудительно заставляет цикл прекратить выполнение текущего шага и перейти к следующей итерации
procedure `Exit;`	Процедура предназначена для безусловного выхода из процедуры или функции
procedure `Halt [ ( Exitcode: Integer) ];`	Экстренно прекращает выполнение программы, допускает передачу кода выхода в параметре `Exitcode`. Старайтесь избегать вызова этого метода, т. к. он не гарантирует нормального уничтожения приложения
procedure `RunError [ ( Errorcode: Byte ) ];`	Прекращает выполнение программы и генерирует ошибку выполнения приложения. Допускает явное указание кода ошибки в параметре `Errorcode`

ПРИЛОЖЕНИЕ 6

Работа с именами папок и файлов

Таблица П6.1. *Функции и процедуры, управляющие именами файлов и папок*

Функция/процедура	Описание
function ChangeFileExt(**const** FileName, Extension: **string**): **string**;	Изменяет расширение файла FileName на новое, определенное параметром Extension, возвращает новое значение имени
function ExcludeTrailingBackslash(**const** S: **string**): **string**;	Удаляет последнюю наклонную черту в пути S
function IncludeTrailingBackslash(**const** S: **string**): **string**;	Завершает путь S наклонной чертой (слэшем)
function ExpandFileName(**const** FileName: **string**): **string**;	Преобразует имя файла FileName в полное имя файла, т. е. *путь\имя.расширение*
function ExpandUNCFileName(**const** FileName: **string**): **string**;	Преобразует имя файла в путь к сетевому файлу: \\<*Имя_сервера*>\<*Ресурс*>
function ExtractFileDir(**const** FileName: **string**): **string**;	Извлекает из строки с полным именем файла путь к файлу
function ExtractFilePath(**const** FileName: **string**): **string**;	Извлекает из строки с полным именем файла путь к файлу, заканчивающийся наклонной чертой
function ExtractFileDrive(**const** FileName: **string**): **string**;	Извлекает из строки с полным именем файла имя диска, завершающееся двоеточием
function ExtractFileExt(**const** FileName: **string**): **string**;	Возвращает расширение в имени файла. В результирующую строку входит разделительная точка и непосредственно расширение
function ExtractFileName(**const** FileName: **string**): **string**;	Извлекает из строки с полным именем файла имя и расширение файла
function ExtractRelativePath(**const** BaseName, DestName: **string**): **string**;	Возвращает относительный путь к файлу (т. е. путь, заданный относительно текущего каталога; текущим каталогом может считаться каталог, в котором находится exe-файл; также относительным путем может считаться путь, установленный в переменных окружения Windows). Здесь DestName — полный путь, BaseName — путь, вычитаемый из полного пути
function ExtractShortPathName(**const** FileName: **string**): **string**;	Конвертирует полный путь к файлу в укороченный формат 8.3 — длинные имена усекаются до 8 символов

Таблица П6.1 *(окончание)*

Функция/процедура	Описание
`function IsPathDelimiter(const S: string; Index: Integer): Boolean;`	Проверяет в позиции `Index` наличие символа наклонной черты `\`
`function MatchesMask(const Filename, Mask: string): Boolean;`	(Функция определена в модуле `Masks`) Проверка соответствия имени файла шаблону маски
`procedure ProcessPath(const EditText: string; var Drive: Char; var DirPart: string; var FilePart: string);`	(Функция определена в модуле `FileCtrl`) Разделяет полное имя файла на составные части: имя диска, путь, имя файла
`function MinimizeName(const Filename: TFileName; Canvas: TCanvas; MaxLen: Integer): TFileName;`	(Функция определена в модуле `FileCtrl`) Метод применяется совместно с элементами управления, обладающими поверхностью для рисования `Canvas`. Задача метода — поместить имя файла `FileName` в области, ограниченной по ширине `MaxLen` пикселами. В соответствии с ограничениями функция минимизирует имя файла — не умещающиеся в элемент управления символы заменяются многоточием в начале, середине или конце имени в зависимости от настроек графического контекста (объект `Canvas`)

ПРИЛОЖЕНИЕ 7

Модуль *IOUtils*

В табл. П7.1 перечислены основные методы объявленной в модуле `IOUtils` записи `TDirectory`. Запись инкапсулирует основные методы по управлению отдельным каталогом. Все представленные методы статические (ключевое слово `static`), т. е. их поведение не может быть переопределено.

Все объявленные в модуле классы кроме обычных файловых путей Windows поддерживают универсальные пути (Universal Naming Convention, UNC).

Таблица П7.1. *Работа с каталогом средствами записи* `TDirectory`

Метод	Описание
`class procedure Copy(const SourceDirName, DestDirName: string);`	Копирование каталога `SourceDirName` в каталог `DestDirName`
`class procedure CreateDirectory(Path: string);`	Создание каталога с путем `Path`
`class procedure Delete(const Path: string); overload;` `class procedure Delete(const Path: string; const Recursive: Boolean); overload;`	Удаление каталога `Path`, вторая версия метода обеспечивает рекурсивный вызов метода при удалении вложенных каталогов (`Recursive=true`)
`class function Exists(const Path: string; FollowLink: Boolean = True): Boolean;`	Проверка существования каталога `Path`
`class function GetAttributes(const Path: string; FollowLink: Boolean = True): TFileAttributes;`	Получение атрибутов каталога
`class function GetCurrentDirectory: string;`	Выяснение текущего каталога
`class procedure SetCurrentDirectory(const Path: string);`	Установка текущего каталога
`class function GetLogicalDrives: TStringDynArray;`	Выяснение текущих логических дисков
`class function GetCreationTime(const Path: string): TDateTime; static;` `class function GetCreationTimeUtc(const Path: string): TDateTime;`	Время создания каталога
`class function GetLastAccessTime(const Path: string): TDateTime;` `class function GetLastAccessTimeUtc(const Path: string): TDateTime;`	Время последнего доступа к каталогу

Таблица П7.1 *(продолжение)*

Метод	Описание
class function GetLastWriteTime(**const** Path: string): TDateTime; **class function** GetLastWriteTimeUtc(**const** Path: **string**): TDateTime;	Время последней записи в каталог
class procedure SetAttributes(**const** Path: **string**; **const** Attributes: TFileAttributes);	Установка атрибутов каталога
class procedure SetCreationTime(**const** Path: **string**; **const** CreationTime: TDateTime); **class procedure** SetCreationTimeUtc(**const** Path: **string**; **const** CreationTime: TDateTime);	Установка времени создания каталога
class procedure SetLastAccessTime(**const** Path: **string**; **const** LastAccessTime: TDateTime); **class procedure** SetLastAccessTimeUtc(**const** Path: **string**; **const** LastAccessTime: TDateTime);	Установка времени последнего доступа к каталогу
class procedure SetLastWriteTime(**const** Path: **string**; **const** LastWriteTime: TDateTime); **class procedure** SetLastWriteTimeUtc(**const** Path: **string**; **const** LastWriteTime: TDateTime);	Установка времени последней записи в каталог
class function GetParent(**const** Path: **string**): **string**;	Получение имени родительского каталога
class function GetDirectories(**const** Path: **string**): TStringDynArray; **overload**; **class function** GetDirectories(**const** Path: **string**; **const** Predicate: TFilterPredicate): TStringDynArray; **overload**; **class function** GetDirectories(**const** Path, SearchPattern: **string**): TStringDynArray; **overload**; **class function** GetDirectories(**const** Path, SearchPattern: **string**; **const** Predicate: TFilterPredicate): TStringDynArray; **overload**; **class function** GetDirectories(**const** Path, SearchPattern: **string**; **const** SearchOption: TSearchOption): TStringDynArray; **overload**; **class function** GetDirectories(**const** Path, SearchPattern: **string**; **const** SearchOption: TSearchOption; **const** Predicate: TFilterPredicate): TStringDynArray; **overload**; **class function** GetDirectories(**const** Path: **string**; **const** SearchOption: TSearchOption; **const** Predicate: TFilterPredicate): TStringDynArray; **overload**;	Сведения о дочерних каталогах, в простейшем случае возвращается полный перечень каталогов. Расширенные версии метода позволяют настраивать критерии выбора
class function GetDirectoryRoot(**const** Path: **string**): **string**;	Получение корневого каталога
class function GetFiles(**const** Path: string): TStringDynArray; **overload**; **class function** GetFiles(**const** Path: **string**; **const** Predicate: TFilterPredicate): TStringDynArray; **overload**; **class function** GetFiles(**const** Path, SearchPattern: **string**): TStringDynArray; **overload**;	Сведения о дочерних файлах. В простейшей нотации метод возвратит полный перечень файлов, принадлежащих каталогу `Path`. Расширенные версии функций позволяют настраивать критерии отбора файлов

Таблица П7.1 *(окончание)*

Метод	Описание

`class function GetFiles(const Path, SearchPattern: string; const Predicate: TFilterPredicate): TStringDynArray; overload;`
`class function GetFiles(const Path, SearchPattern: string; const SearchOption: TSearchOption): TStringDynArray; overload;`
`class function GetFiles(const Path, SearchPattern: string; const SearchOption: TSearchOption; const Predicate: TFilterPredicate): TStringDynArray; overload;`
`class function GetFiles(const Path: string; const SearchOption: TSearchOption; const Predicate: TFilterPredicate): TStringDynArray; overload;`	

`class function GetFileSystemEntries(const Path: string): TStringDynArray; overload;`
`class function GetFileSystemEntries(const Path: string; const Predicate: TFilterPredicate): TStringDynArray; overload;`
`class function GetFileSystemEntries(const Path, SearchPattern: string): TStringDynArray; overload;`
`class function GetFileSystemEntries(const Path, SearchPattern: string; const Predicate: TFilterPredicate): TStringDynArray; overload;`
`class function GetFileSystemEntries(const Path: string; const SearchOption: TSearchOption; const Predicate: TFilterPredicate): TStringDynArray; overload;`
	Сведения о дочерних файлах и каталогах. В простейшей нотации метод возвратит полный перечень файлов и каталогов, принадлежащих каталогу `Path`. Расширенные версии функций позволяют настраивать критерии отбора файлов и каталогов

`class function IsEmpty(const Path: string): Boolean;`	Проверка на пустой каталог
`class function IsRelativePath(const Path: string): Boolean;`	Проверка относительного пути
`class procedure Move(const SourceDirName, DestDirName: string);`	Перемещение каталога

Запись `TFile` обладает методами, позволяющими манипулировать файлами (создание, удаление, операции ввода/вывода, поиск). Методы записи представлены в табл. П7.2.

Таблица П7.2. *Работа с файлом средствами* `TFile`

Метод	Описание
`class function Create(const Path: string): TFileStream; overload;` `class function Create(const Path: string; const BufferSize: Integer): TFileStream; overload;`	Создает файл `Path`, готовый для записи, и возвращает экземпляр потока `TFileStream`
`class procedure AppendAllText(const Path, Contents: string); overload;` `class procedure AppendAllText(const Path, Contents: string; const Encoding: TEncoding); overload;`	Добавляет к файлу весь текст из параметра `Contents`. Расширенная версия функции позволяет обеспечить шифрование данных
`class function AppendText(const Path: string): TStreamWriter;`	Открывает файл для добавления в него текстовых данных, возвращает поток `TStreamWriter`

***Таблица П7.2** (продолжение)*

Метод	Описание
class procedure Copy(**const** SourceFileName, DestFileName: **string**); **overload;** **class procedure** Copy(**const** SourceFileName, DestFileName: **string**; **const** Overwrite: Boolean); **overload;**	Копирует файл SourceFileName в файл DestFileName
class function CreateSymLink(**const** Link, Target: string): Boolean;	Создает ссылку на файл
class function CreateText(**const** Path: **string**): TStreamWriter;	Создает текстовый файл и предоставляет к нему доступ в виде потока TStreamWriter
class procedure Decrypt(**const** Path: **string**);	Расшифровка файла
class procedure Delete(**const** Path: **string**);	Удаляет файл
class procedure Encrypt(**const** Path: **string**);	Шифрование файла
class function Exists(**const** Path: **string**; FollowLink: Boolean = True): Boolean;	Проверяет факт существования файла
class function GetAttributes(**const** Path: **string**; FollowLink: Boolean = True): TFileAttributes;	Возвращает атрибуты файла
class function GetCreationTime(**const** Path: **string**): TDateTime; **class function** GetCreationTimeUtc(**const** Path: **string**): TDateTime;	Возвращает время создания файла
class function GetLastAccessTime(**const** Path: **string**): TDateTime; **class function** GetLastAccessTimeUtc(**const** Path: **string**): TDateTime;	Возвращает время последнего доступа к файлу
class function GetLastWriteTime(**const** Path: **string**): TDateTime; **class function** GetLastWriteTimeUtc(**const** Path: **string**): TDateTime;	Возвращает время последнего изменения файла
class function GetSymLinkTarget(**const** FileName: **string**; SymLinkRec: TSymLinkRec): Boolean; **overload;** **class function** GetSymLinkTarget(**const** FileName: **string**; **var** TargetName: string): Boolean; **overload;**	Дает ссылку на файл
class procedure Move(SourceFileName, DestFileName: **string**);	Перемещает файл
class function Open(**const** Path: **string**; **const** Mode: TFileMode): TFileStream; **overload;** **class function** Open(**const** Path: **string**; **const** Mode: TFileMode; **const** Access: TFileAccess): TFileStream; **overload;** **class function** Open(**const** Path: **string**; **const** Mode: TFileMode; **const** Access: TFileAccess; **const** Share: TFileShare): TFileStream; **overload;**	Открывает файл Path и предоставляет к нему доступ при посредничестве файлового потока TFileStream
class function OpenRead(**const** Path: **string**): TFileStream;	Открывает файл только для чтения

Таблица П7.2 *(окончание)*

Метод	Описание
`class function OpenText(const Path: string): TStreamReader;`	Открывает для доступа к тексту
`class function OpenWrite(const Path: string): TFileStream;`	Открывает для записи
`class function ReadAllBytes(const Path: string): TBytes;`	Открывает файл и предоставляет к нему доступ в виде массива байтов
`class function ReadAllLines(const Path: string): TStringDynArray; overload;` `class function ReadAllLines(const Path: string; const Encoding: TEncoding): TStringDynArray; overload;`	Читает строки из файла и возвращает их в виде массива строк
`class function ReadAllText(const Path: string): string; overload;` `class function ReadAllText(const Path: string; const Encoding: TEncoding): string; overload;`	Читает данные из файла и возвращает их в формате строки
`class procedure Replace(const SourceFileName, DestinationFileName, DestinationBackupFileName: string); overload;` `class procedure Replace(SourceFileName, DestinationFileName, DestinationBackupFileName: string; const IgnoreMetadataErrors: Boolean); overload;`	Заменяет файл с именем `DestinationFileName` файлом `SourceFileName`
`class procedure SetAttributes(const Path: string; const Attributes: TFileAttributes);`	Устанавливает атрибуты файла
`class procedure SetCreationTime(const Path: string; const CreationTime: TDateTime);` `class procedure SetCreationTimeUtc(const Path: string; const CreationTime: TDateTime);`	Устанавливает время создания файла
`class procedure SetLastAccessTime(const Path: string; const LastAccessTime: TDateTime);` `class procedure SetLastAccessTimeUtc(const Path: string; const LastAccessTime: TDateTime);`	Устанавливает время последнего доступа к файлу
`class procedure SetLastWriteTime(const Path: string; const LastWriteTime: TDateTime);` `class procedure SetLastWriteTimeUtc(const Path: string; const LastWriteTime: TDateTime);`	Устанавливает время последнего изменения файла
`class procedure WriteAllBytes(const Path: string; const Bytes: TBytes);`	Записывает в файл данные из массива байтов `Bytes`
`class procedure WriteAllLines(const Path: string; const Contents: TStringDynArray); overload; static;` `class procedure WriteAllLines(const Path: string; const Contents: TStringDynArray; const Encoding: TEncoding); overload;`	Записывает в файл все строки из массива
`class procedure WriteAllText(const Path, Contents: string); overload;` `class procedure WriteAllText(const Path, Contents: string; const Encoding: TEncoding); overload;`	Записывает в файл текст из параметра `Contents`

Запись `TPath` специализируется на обслуживании имен файлов и папок (табл. П7.3).

Таблица П7.3. *Путь* `TPath`

Методы и свойства	Описание
class function IsValidPathChar(**const** AChar: Char): Boolean;	Проверка символа на допустимость применения в пути
class function IsValidFileNameChar(const AChar: Char): Boolean;	Проверка допустимости символа для использования в имени файла
class function HasValidPathChars(**const** Path: **string**; **const** UseWildcards: Boolean): Boolean;	Проверка допустимости символов в пути
class function HasValidFileNameChars(**const** FileName: **string**; **const** UseWildcards: Boolean): Boolean;	Проверка допустимости символов в имени файла
class function GetExtendedPrefix(**const** Path: **string**): TPathPrefixType; TPathPrefixType = (pptNoPrefix, pptExtended, pptExtendedUNC);	Возвращает тип пути, если путь начинается с префикса `'\\?\'`, будет возвращено `pptExtended`, если с `'\\?\UNC\'` — `pptExtendedUNC`, иначе — `pptNoPrefix`
class function IsDriveRooted(**const** Path: **string**): Boolean;	Проверяет, находится ли в корне пути имя логического диска
class function IsExtendedPrefixed(**const** Path: **string**): Boolean;	Проверяет, не начинается ли путь с префикса `'\\?\UNC\'`
class function IsRelativePath(**const** Path: **string**): Boolean;	Проверяет, является ли путь относительным
class function IsUNCPath(**const** Path: **string**): Boolean;	Проверяет, является ли путь универсальным (UNC)
class function IsUNCRooted(**const** Path: **string**): Boolean;	Проверяет, находится ли в корне пути префикс `'\\?\UNC\'`
class function GetGUIDFileName(**const** UseSeparator: Boolean = False): **string**;	Возвращает глобальный универсальный идентификатор файла
class function DriveExists(**const** Path: **string**): Boolean;	Проверяет, имеется ли в составе пути логический диск
class function MatchesPattern(**const** FileName, Pattern: **string**; **const** CaseSensitive: Boolean): Boolean;	Проверка соответствия имени `FileName` шаблону `Pattern` с учетом (или без учета) регистра `CaseSensitive`
class function ChangeExtension(**const** Path, Extension: string): **string**;	Изменяет расширение имени файла
class function Combine(**const** Path1, Path2: **string**): **string**;	Проверяет два пути к одному и тому же ресурсу и возвращает абсолютный путь
class function GetDirectoryName(FileName: **string**): **string**;	Возвратит путь к файлу `FileName`
class function GetExtension(**const** FileName: **string**): **string**;	Возвратит расширение имени файла
class function GetFileName(**const** FileName: **string**): **string**;	Возвратит только имя файла из полного имени файла

Таблица П7.3 *(окончание)*

Методы и свойства	Описание
class function GetFileNameWithoutExtension(**const** FileName: **string**): **string**;	Возвратит имя файла без расширения имени
class function GetFullPath(**const** Path: **string**): **string**;	Возвратит путь к файлу
class function GetInvalidFileNameChars: TCharArray;	Возвратит массив с недопустимыми символами в имени файла
class function GetInvalidPathChars: TCharArray;	Возвратит массив с недопустимыми символами в пути
class function GetPathRoot(**const** Path: **string**): **string**;	Возвратит корневой элемент в пути
class function GetRandomFileName: **string**;	Возвратит случайное имя файла
class function GetTempFileName: **string**;	Возвратит путь и имя для временного файла
class function GetTempPath: **string**;	Возвратит путь к временной папке
class function GetHomePath: **string**;	Возвратит путь к папке Roaming
class function GetAttributes(**const** Path: **string**; FollowLink: Boolean = True): TFileAttributes;	Возвратит атрибуты файла
class procedure SetAttributes(**const** Path: **string**; **const** Attributes: TFileAttributes);	Установит атрибуты файла
class function HasExtension(**const** Path: **string**): Boolean;	Проверит наличие расширения имени
class function IsPathRooted(**const** Path: **string**): Boolean;	Проверит наличие в пути корневого элемента
class property ExtensionSeparatorChar: Char;	Хранит символ-разделитель имени и расширения имени, по умолчанию `'.'`
class property AltDirectorySeparatorChar: Char;	Альтернативный символ-разделитель между именами каталогов, по умолчанию `'/'`
class property DirectorySeparatorChar: Char;	Символ-разделитель между именами каталогов, по умолчанию `'\'`
class property PathSeparator: Char;	Символ-разделитель между путями, по умолчанию `';'`
class property VolumeSeparatorChar: Char;	Символ-разделитель имени диска, по умолчанию `':'`

ПРИЛОЖЕНИЕ 8

Константы CSIDL

В табл. П8.1 представлены константы, используемые в функциях `SHGetFolderLocation()`, `SHGetFolderPath()`, `SHGetSpecialFolderLocation()` и `SHGetSpecialFolderPath()`, предназначенных для выяснения пути к стандартным папкам Windows Vista/7.

Таблица П8.1. *Константы CSIDL*

Идентификатор	Значение	Описание
`CSIDL_DESKTOP`	`$0000`	Рабочий стол Windows C:\Users\<Пользователь>\Desktop
`CSIDL_INTERNET`	`$0001`	Виртуальная папка Internet
`CSIDL_PROGRAMS`	`$0002`	Системный каталог, содержащий группу программ пользователя C:\Users\<*Пользователь*>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs
`CSIDL_CONTROLS`	`$0003`	Виртуальная папка Панели управления
`CSIDL_PRINTERS`	`$0004`	Виртуальная папка, содержащая установленные принтеры
`CSIDL_PERSONAL`	`$0005`	Системный каталог с документами пользователя C:\Users\<*Пользователь*>\Documents
`CSIDL_FAVORITES`	`$0006`	Хранилище избранных данных C:\Users\<*Пользователь*>\Favorites
`CSIDL_STARTUP`	`$0007`	Каталог автозапуска программ для текущего пользователя C:\Users\<*Пользователь*>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup
`CSIDL_RECENT`	`$0008`	Ссылки на недавно вызываемые документы C:\Users\<*Пользователь*>\AppData\Roaming\Microsoft\Windows\Recent
`CSIDL_SENDTO`	`$0009`	Пункты меню **Отправить**, появляющиеся при щелчке правой кнопкой мыши по файлу в Проводнике: C:\Users\<*Пользователь*>\AppData\Roaming\Microsoft\Windows\SendTo
`CSIDL_BITBUCKET`	`$000a`	Виртуальная папка. Содержит объекты пользователя, помещенные в Корзину

Таблица П8.1 *(продолжение)*

Идентификатор	Значение	Описание
CSIDL_STARTMENU	$000b	Ярлыки меню **Пуск** текущего пользователя C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Start Menu
CSIDL_MYMUSIC	$000d	Музыкальные файлы C:\Users\<*Пользователь*>\Music
CSIDL_MYVIDEO	$000e	Видеофайлы: C:\Users\<*Пользователь*>\Video
CSIDL_DESKTOPDIRECTORY	$0010	Рабочий стол пользователя C:\Users\<*Пользователь*>\Desktop
CSIDL_DRIVES	$0011	Виртуальная папка **Мой компьютер**
CSIDL_NETWORK	$0012	Виртуальная папка старшего узла в иерархии папок сетевого окружения
CSIDL_NETHOOD	$0013	Ярлыки объектов сетевого окружения C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Network Shortcuts
CSIDL_FONTS	$0014	Папка шрифтов системы C:\Windows\Fonts
CSIDL_TEMPLATES	$0015	Шаблоны документов C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Templates
CSIDL_COMMON_STARTMENU	$0016	Содержит общее для всех пользователей меню запуска C:\ProgramData\Microsoft\Windows\Start Menu
CSIDL_COMMON_PROGRAMS	$0017	Общие для всех пользователей папки групп программ из меню **Пуск** C:\ProgramData\Microsoft\Windows\Start Menu\Programs
CSIDL_COMMON_STARTUP	$0018	Общая для всех пользователей папка автозапуска C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup
CSIDL_COMMON_DESKTOPDIRECTORY	$0019	Папка, содержащая общие для всех пользователей объекты рабочего стола C:\Users\Public\Desktop
CSIDL_APPDATA	$001a	Место для хранения пользовательских данных приложений C:\Users\<*Пользователь*>\AppData\Roaming
CSIDL_PRINTHOOD	$001b	Ярлыки принтеров пользователя C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Printer Shortcuts
CSIDL_LOCAL_APPDATA	$001c	Локализация данных приложений C:\Users\<*Пользователь*>\AppData\Local
CSIDL_ALTSTARTUP	$001d	Системный каталог автозапуска для нелокализованных приложений C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Start Menu\Programs\Startup

Таблица П8.1 *(продолжение)*

Идентификатор	Значение	Описание
CSIDL_COMMON_ALTSTARTUP	$001e	Папка автозапуска для всех пользователей C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup
CSIDL_COMMON_FAVORITES	$001f	Избранные данные пользователя: C:\Users\<*Пользователь*>\Favorites
CSIDL_INTERNET_CACHE	$0020	Хранилище временных файлов Интернета C:\Users\<*Пользователь*>\AppData\Local\ Microsoft\Windows\Temporary Internet Files
CSIDL_COOKIES	$0021	Системный каталог для файлов cookies C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Cookies
CSIDL_HISTORY	$0022	Ссылки на посещенные ранее узлы Интернета C:\Users\<*Пользователь*>\AppData\Local\ Microsoft\Windows\History
CSIDL_COMMON_APPDATA	$0023	Каталог данных приложений: C:\ProgramData
CSIDL_WINDOWS	$0024	Каталог C:\Windows
CSIDL_SYSTEM	$0025	Системный каталог C:\Windows\system32
CSIDL_PROGRAM_FILES	$0026	Каталог программ C:\Program Files
CSIDL_MYPICTURES	$0027	Каталог рисунков C:\Users\<*Пользователь*>\Pictures
CSIDL_PROFILE	$0028	Профайл пользователя C:\Users\<*Пользователь*>
CSIDL_SYSTEMX86	$0029	Системный каталог *x*86 для платформы RISC
CSIDL_PROGRAM_FILESX86	$002a	Каталог программ x86 для платформы RISC
CSIDL_PROGRAM_FILES_COMMON	$002b	Каталог C:\Program Files\Common Files
CSIDL_PROGRAM_FILES_COMMONX86	$002c	Каталог *x*86 Program Files\Common для платформы RISC
CSIDL_COMMON_TEMPLATES	$002d	Каталог общих шаблонов C:\ProgramData\Microsoft\Windows\Templates
CSIDL_COMMON_DOCUMENTS	$002e	Каталог общих документов C:\Users\Public\Documents
CSIDL_COMMON_ADMINTOOLS	$002f	Каталог администрирования C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Administrative Tools
CSIDL_ADMINTOOLS	$0030	Каталог администрирования пользователя C:\Users\<*Пользователь*>\AppData\Roaming\ Microsoft\Windows\Start Menu\Programs\ Administrative Tools
CSIDL_CONNECTIONS	$0031	Сетевые соединения
CSIDL_COMMON_MUSIC	$0035	Общий каталог для музыкальных файлов C:\Users\Public\Music
CSIDL_COMMON_PICTURES	$0036	Общий каталог для рисунков C:\Users\Public\Pictures

Таблица П8.1 *(окончание)*

Идентификатор	Значение	Описание
CSIDL_COMMON_VIDEO	$0037	Общий каталог видеофайлов C:\Users\Public\Videos
CSIDL_RESOURCES	$0038	Каталог ресурсов C:\Windows\resources
CSIDL_RESOURCES_LOCALIZED	$0039	Каталог локализованных ресурсов
CSIDL_CDBURN_AREA	$003b	Каталог записи оптических дисков C:\Users\<*Пользователь*>\AppData\Local\ Microsoft\Windows\Burn\Burn
CSIDL_COMPUTERSNEARME	$003d	Компьютерное окружение

ПРИЛОЖЕНИЕ 9

Холст *FMX.Types.TCanvas*

В проектах VCL объектно-ориентированным воплощением контекста графического устройства GDI выступал, выступает и наверняка еще очень долго будет выступать хорошо нам знакомый класс `TCanvas`. В двумерных проектах FireMonkey на поприще графики трудится одноименный `TCanvas` (модуль `FMX.Types`), но для того чтобы одновременно стать слугой двух господ (Mac OS X и Windows), он организован по-другому.

Для достижения универсальности принципиально изменена иерархия предков. Теперь в родительском списке здесь мы обнаружим два интерфейса:

```
type TCanvas = class(TInterfacedPersistent, IFreeNotification)
```

Первый интерфейс является прародителем всех объектов, способных сохранять в памяти и загружать из памяти значения своих полей. Второй интерфейс позволяет уведомлять другие (в первую очередь зависимые от холста) объекты о прекращении существования экземпляра класса.

Изменились и приемы программирования с холстом. Так листинг П9.1 демонстрирует порядок вывода обычной линии на поверхности формы проекта FMX.

Листинг П9.1. Вывод линии средствами `FMX.Types.TCanvas`

```
var APt1, APt2: TPointF;
begin
  APt1:=Form1.ClientRect.TopLeft;
  APt2:=Form1.ClientRect.BottomRight;
  with Form1.Canvas do
  begin
    if BeginScene then  //создание графической сцены
    begin
      DrawLine(APt1, APt2, 1); //рисуем линию
      EndScene; //завершение сцены и вывод изображения на экран
    end;
  end;
end;
```

ВНИМАНИЕ!

Графический вывод FMX осуществляется в рамках отдельных сцен. Прорисовка на поверхности физического устройства начинается только после полного формирования сцены. Такое решение позволяет улучшить качество отображаемой картинки, т. к. рисование осуществляется в памяти, и только затем уже сформированный образ переносится на экран.

Управление холстом

Первая, бросающаяся в глаза особенность листинга 9.1, — необходимость уведомлять холст о начале и завершении прорисовки. Графический вывод предваряется обращением к методу

```
function BeginScene: Boolean;
```

В момент открытия сцены устанавливается в исходное состояние свойства холста (в первую очередь кисти заливки областей и рисования линий).

Команда на закрытие сцены

```
procedure EndScene;
```

уведомляет, что все необходимые команды отправлены в систему, и она может приступать к выводу графики.

Предусмотрены два метода, позволяющих быстро очистить холст. Полную очистку всей площади холста осуществит процедура

```
procedure Clear(const Color: TAlphaColor);
```

Единственный параметр метода `Color` содержит новый цвет заливки холста.

Для очистки только заданной прямоугольной области воспользуйтесь методом

```
procedure ClearRect(const ARect: TRectF; const AColor: TAlphaColor = 0);
```

границы области подлежащей перерисовке `ARect`.

Ширину и высоту холста возвращают свойства:

```
property Width: Integer;  //только для чтения
property Height: Integer; //только для чтения
```

Для улучшения качества графического вывода FireMonkey перед выводом изображения на холст формирует картинку в памяти и только после этого переносит ее на экран. Признаком того, что холст работает в режиме буферизации, является значение `true`, возвращаемое свойством

```
property Buffered: Boolean; //только для чтения
```

Доступ к указателю на область памяти буфера и к дескриптору буфера соответственно предоставляют свойства:

```
property BufferBits: Pointer;
property BufferHandle: THandle;
```

Для управления особенностями прорисовки графических примитивов при работе с холстом VCL мы пользовались услугами кисти, пера и шрифта. У холста FMX помощники те же самые, однако работать с ними существенно интереснее (табл. П9.1).

Таблица П9.1. *Атрибуты холста* `FMX.Types.TCanvas`

Свойство	Описание
`property Fill: TBrush;`	Кисть, используемая для заливки замкнутых областей
`property Stroke: TBrush;`	Кисть, используемая для рисования линий
`property Font: TFont;`	Шрифт

Кисть *FMX.Types.TBrush*

Первой и едва ли не основной помощницей кроссплатформенной версии холста можно считать кисть TBrush. Вас не должно вводить в заблуждение знакомое название класса кисти. Кисть FMX.Types.TBrush на порядок превосходит свою "коллегу" из VCL. Это подтверждают свойства новой кисти.

Класс FMX.Types.TBrush позволяет создать шесть разновидностей инструментов для закраски поверхности:

```
property Kind: TBrushKind;
type TBrushKind = (bkNone,  //пустая кисть
           bkSolid,     //сплошная кисть
           bkGradient,  //кисть с градиентной заливкой
           bkBitmap,    //кисть на основе растрового образа
           bkResource,  //кисть из ресурса
           bkGrab);     //кисть присвоенная от другого элемента TControl
```

В зависимости от вида кисти задействуется то или другое свойство класса FMX.Types.TBrush. При определении цвета сплошной кисти обращаемся к свойству

```
property Color: TAlphaColor;
```

Кисть bkGradient умеет хранить описание градиентной заливки:

```
property Gradient: TGradient;
```

Кисть bkBitmap способна интегрировать в себя растровый образ

```
property Bitmap: TBrushBitmap;
```

или произвольный ресурс, как кисть bkResource

```
property Resource: TBrushResource;
```

Наконец, кисть bkGrab можно позаимствовать у другого элемента управления, для этого следует обратиться за помощью к свойству

```
property Grab: TBrushGrab;
```

Порядок заливки эллиптической области сплошной кистью демонстрирует листинг П9.2.

Листинг П9.2. Закраска эллипса сплошной кистью

```
with Form1.Canvas do
begin
  if BeginScene then
   begin
     Fill.Kind:=TBrushKind.bkSolid;
     Fill.Color:=TAlphaColors.AliceBlue;
     FillEllipse(Form1.ClientRect,1);
     EndScene;
   end;
end;
```

Немаловажной особенностью кисти FMX является способность реагировать на события, в частности на изменения в параметрах кисти:

```
property OnChanged: TNotifyEvent;
```

А если кисть предназначена для закраски градиентом, то и в изменениях в настройках градиента

```
property OnGradientChanged: TNotifyEvent;
```

Внешний вид линий

Читатель, внимательно изучивший табл. П9.1, наверняка заметил, что в графике FireMonkey роль пера также отводится кисти. Однако на этот раз экземпляр кисти-пера передается в свойство `Stroke` (листинг П9.3). Если вы уже попробовали поработать с кистью `FMX.Types.TBrush`, то уже поняли, что возможности FireMonkey по выводу линий поистине безграничны. Чего только стоит градиентная линия... Кроме того, внешний вид линий зависит от состояния квартета свойств.

Толщина линии определяется свойством

```
property StrokeThickness: Single;
```

Особенности начертания определяет свойство

```
property StrokeDash: TStrokeDash;
type TStrokeDash = (sdSolid, sdDash, sdDot, sdDashDot, sdDashDotDot, sdCustom);
```

При желании можно создать собственную модель штриховки линии, для этого пригодится свойство

```
procedure SetCustomDash(const Dash: array of Single; Offset: Single);
```

Наконечники линий могут быть плоскими и скругленными:

```
property StrokeCap: TStrokeCap;
type TStrokeCap = (scFlat, scRound);
```

Наконец, места соединения нескольких линий также могут настраиваться свойством

```
property StrokeJoin: TStrokeJoin;
type TStrokeJoin = (sjMiter, sjRound, sjBevel);
```

Соединения могут быть ограничены, скруглены или отсечены.

Порядок работы с пером отражен в листинге 9.3.

Листинг П9.3. Чертим контур прямоугольника и эллипса

```
var aRect:TRectF;
begin
  aRect:=RectF(10,10,200,200);
  with Form1.Canvas do
  begin
    if BeginScene then
    begin
      Stroke.Color:=TAlphaColorRec.Crimson;
      StrokeThickness:=0.5;
      DrawRect(aRect,10,10,[TCorner.crTopLeft, TCorner.crBottomRight], 1);

      Stroke.Color:=TAlphaColorRec.Blueviolet;
      StrokeDash:=TStrokeDash.sdDashDot;
      StrokeThickness:=1;
```

```
      DrawEllipse(aRect,1);
      EndScene;
    end;
  end;
end;
```

Шрифт *FMX.Types.TFont*

С точки зрения программиста Delphi нововведения FMX не сильно затронули устоявшийся со времен VCL подход к представлению шрифта. В проектах FireMonkey класс шрифта сохранил традиционное имя `TFont`, но на этот раз объявление класса перенесено в модуль `FMX.Types`.

Три базовых свойства шрифта обеспечивают управление гарнитурой, размером и стилем:

```
property Family: TFontName;  //гарнитура шрифта
property Size: Single;       //размер шрифта
property Style: TFontStyles; //стиль шрифта
type TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);
```

В листинге П9.4 предложен код, позволяющий вывести строку текста в заданной прямоугольной области. Обратите внимание, что цвет надписи определяется цветом кисти.

Листинг П9.4. Вывод текстовой надписи

```
var aRect:TRectF;
begin
  aRect:=RectF(10,10,200,40);
  with Form1.Canvas do
  begin
    if BeginScene then
    begin
      Fill.Color:=TAlphaColors.Red;
      Font.Style:=[TFontStyle.fsItalic];
      Font.Family:='Arial';
      Font.Size:=20;
      FillText(aRect,'Привет, мир!',True,1,[],TTextAlign.taCenter);
      EndScene;
    end;
  end;
end;
```

Заливка замкнутых областей

Все методы графического вывода содержат параметр `AOpacity:Single`, определяющий степень прозрачности рисуемого примитива. Параметр должен принимать значения в пределах от 0 до 1, где 0 соответствует абсолютно прозрачному изображению, а 1 — непрозрачному.

На любых графических платформах наиболее быстрой графической операцией считается заливка замкнутой области текущей кистью (кистью, выбранной в свойстве `Fill` холста FMX). Методы заливки предложены в табл. П9.2.

Таблица П9.2. *Методы заливки областей*

Метод	Описание
`procedure FillRect(const ARect: TRectF; const XRadius, YRadius: Single; const ACorners: TCorners; const AOpacity: Single; const ACornerType: TCornerType = TCornerType.ctRound);`	Заливка прямоугольной области `ARect`. Допускается вывод скругленных углов, в этом случае следует определить радиусы `XRadius` и `YRadius`. Параметр `ACorners` уточняет, какой именно угол (или углы) следует нарисовать скругленными. Особенности вывода назначаются параметром `ACornerType`
`procedure FillEllipse(const ARect: TRectF; const AOpacity: Single);`	Заливка эллиптической области в границах прямоугольника `ARect`
`procedure FillArc(const Center, Radius: TPointF; StartAngle, SweepAngle: Single; const AOpacity: Single);`	Заливка сектора окружности с центром `Center` и радиусом `Radius`. Раскрытие сектора определяется двумя лучами, проведенными из центра в точки `StartAngle` и `SweepAngle`
`procedure FillPath(const APath: TPathData; const AOpacity: Single);`	Заливка траектории, описанной в структуре `APath`
`procedure FillPolygon(const Points: TPolygon; const AOpacity: Single);`	Заливка произвольного многоугольника с вершинами, заданными в массиве `Points`

Листинг П9.5 предлагает простой пример заливки прямоугольной области градиентной кистью. Обратите внимание, что точки `StartPosition` и `StopPosition` определят координаты вектора заливки. Значения координат вектора нормируются к 2. Например, если бы мы захотели направить градиент из левого верхнего угла формы в правый нижний, то следовало передать координаты (0, 0) — (2, 2).

Листинг П9.5. Градиентная заливка клиентской области формы

```
with Form1.Canvas do
begin
  if BeginScene then
   begin
   Fill.Kind:=TBrushKind.bkGradient;
   with Fill.Gradient do
   begin
     StartPosition.X:=0.5;StartPosition.Y:=0;
     StopPosition.X:=0.5; StopPosition.Y:=2;
     Style:=TGradientStyle.gsLinear;
     Color:=TAlphaColors.White;
     Color1:=TAlphaColors.Red;
   end;
   FillRect(Form1.ClientRect,0,0, [TCorner.crTopLeft, TCorner.crTopRight,
            TCorner.crBottomLeft, TCorner.crBottomRight],1);
   EndScene;
   end;
end;
```

Вывод простейших фигур

Вывод графических примитивов осуществляется кистью, выбранной в свойстве `Stroke`. Методы, отвечающие за вывод простейших геометрических фигур, представлены в табл. П9.3.

Таблица П9.3. *Методы черчения простейших фигур*

Метод	Описание
procedure DrawLine(**const** APt1, APt2: TPointF; **const** AOpacity: Single);	Чертит отрезок из точки `APt1` в `APt2`
procedure DrawRect(**const** ARect: TRectF; const XRadius, YRadius: Single; **const** ACorners: TCorners; **const** AOpacity: Single; **const** ACornerType: TCornerType = TCornerType.ctRound);	Вывод прямоугольника. Параметры метода идентичны параметрам `FillEllipse`
procedure DrawArc(**const** Center, Radius: TPointF; StartAngle, SweepAngle: Single; **const** AOpacity: Single);	Вывод сектора. Параметры метода идентичны параметрам `FillArc`
procedure DrawRectSides(**const** ARect: TRectF; **const** XRadius, YRadius: Single; const ACorners: TCorners; **const** AOpacity: Single; **const** ASides: TSides; **const** ACornerType: TCornerType = TCornerType.ctRound);	Чертит стороны прямоугольника `ASides`
procedure DrawEllipse(**const** ARect: TRectF; **const** AOpacity: Single);	Чертит эллипс в границах прямоугольной области
procedure DrawPath(**const** APath: TPathData; **const** AOpacity: Single);	Чертит траекторию по точкам
procedure DrawPolygon(**const** Points: TPolygon; **const** AOpacity: Single);	Выводит многоугольник с вершинами `Points`

Вывод текста

Одна из сложнейших графических задач вывода текстовых данных в FireMonkey превращается в пару пустяков. Гарнитура, стиль и размер шрифта назначаются в свойстве `Font` холста, цвет шрифта определяется цветом текущей кисти. Методы холста `FMX.Types.TCanvas`, отвечающие за работу с текстом, представлены в табл. П9.4.

Таблица П9.4. *Вывод текстовых данных*

Метод	Описание
function LoadFontFromStream(AStream: TStream): Boolean;	Загрузка шрифта из потока `AStream`
function TextWidth(**const** AText: string): Single;	Расчет ширины текстовой строки
function TextHeight(**const** AText: string): Single;	Расчет высоты текста
procedure FillText(**const** ARect: TRectF; **const** AText: **string**; **const** WordWrap: Boolean; **const** AOpacity: Single; **const** Flags: TFillTextFlags; **const** ATextAlign: TTextAlign; **const** AVTextAlign: TTextAlign = TTextAlign.taCenter);	Вывод текста `AText` в области `ARect`. Перенос текста на новую строку определяется параметром `WordWrap`. Направление вывода (по умолчанию слева направо) устанавливается в `Flags`. Горизонтальное выравнивание `ATextAlign` и вертикальное выравнивание `AVTextAlign`

***Таблица П9.4** (окончание)*

Метод	Описание
`function TextToPath(Path: TPathData; const ARect: TRectF; const AText: string; const WordWrap: Boolean; const ATextAlign: TTextAlign; const AVTextAlign: TTextAlign = TTextAlign.taCenter): Boolean;`	Вывод текста по траектории `Path`

Отображение рисунков

Для прорисовки на поверхности холста графического образа следует применять метод

```
procedure DrawBitmap(const ABitmap: TBitmap; const SrcRect, DstRect: TRectF;
const AOpacity: Single; const HighSpeed: Boolean = False);
```

Процедура выводит изображение `ABitmap` полностью или его часть. Подлежащая выводу область изображения определяется параметром `SrcRect`. Место вывода и размеры результирующего изображения назначаются в параметре `DstRect`. Параметр `HighSpeed` позволит программисту отдать предпочтение скорости (`true`) или качеству (`false`) вывода.

Листинг 9.6 демонстрирует порядок работы с методом `DrawBitmap()`. В представленном примере мы загружаем графический образ из файла и выводим его на поверхности формы. Если нам потребуется отобразить только фрагмент исходного рисунка, то следует определить границы интересующего нас фрагмента в параметре `SrcRect`. Управляя размерами области вывода `DstRect`, мы сможем увеличить или уменьшить рисунок.

Листинг П9.6. Вывод графического образа без масштабирования

```
var ABitmap:TBitmap;
    aWidth,aHeight:integer;
    SrcRect, DstRect: TRectF;
begin
  if OpenDialog1.Execute then
    with Form1.Canvas do
    begin
      ABitmap:=TBitmap.CreateFromFile(OpenDialog1.FileName);
      aWidth:=ABitmap.Width;
      aHeight:=ABitmap.Height;
      SrcRect:=RectF(0,0,aWidth,aHeight);
      DstRect:=SrcRect;
      if BeginScene then
      begin
        DrawBitmap(ABitmap,SrcRect,DstRect,1,true);
        EndScene;
      end;
    end;
end;
```

Для организации быстрого просмотра изображения `ABitmap` стоит воспользоваться методом

```
procedure DrawThumbnail(const ABitmap: TBitmap;
                        const Width, Height: Single);
```

создающим графическую миниатюру заданного размера.

Отсечение

Как в Windows, так и в Mac OS X в качестве основного инструмента, позволяющего настроить границы области, в которых будет осуществляться вывод, выступает регион отсечения. Библиотека FireMonkey порадовала возможностью простого управления регионами отсечения. В распоряжении программиста предоставлены 4 метода. Установку прямоугольной области отсечения осуществляет процедура

```
procedure SetClipRects(const ARects: array of TRectF);
```

Изменить текущий регион отсечения, создав новый на основе пересечения текущего региона и прямоугольной области `Arect`, позволит процедура

```
procedure IntersectClipRect(const ARect: TRectF);
```

Метод

```
procedure ExcludeClipRect(const ARect: TRectF);
```

наоборот, исключает из текущего региона отсечения прямоугольник `ARect`. Наконец, сброс региона отсечения и установку его в состояние по умолчанию осуществит метод

```
procedure ResetClipRect;
```

Сохранение и восстановление состояния холста

При необходимости мы можем сохранить текущее состояние холста (сведения о кистях, шрифте, матрице преобразований и т. п.). Для этой цели предназначена функция

```
function SaveState: TCanvasSaveState;
```

Сведения о состоянии холста представляются в формате объекта `TCanvasSaveState`. Для восстановления состояния холста вызываем метод

```
procedure RestoreState(State: TCanvasSaveState);
```

ПРИЛОЖЕНИЕ 10

Описание электронного архива

По ссылке **ftp://85.249.45.166/9785977508254.zip** можно скачать электронный архив с рассмотренными проектами и исходными кодами примеров. Эта ссылка доступна также со страницы книги на сайте **www.bhv.ru**.

После распаковки архива examples.exe вы получите доступ к исходному коду более чем 200 примеров.

По умолчанию распаковка архива производится в папку c:\listings.

Папка c:\listings\examples содержит каталоги с исходным кодом примеров. Названия вложенных папок построены по следующему принципу:

<ch><номер главы>_<номер примера в главе>

Например:

ch01_01 — глава 1 пример 1;

ch01_02 — глава 1 пример 2;

...

ch50_05 — глава 50 пример 5.

Кроме того, в папке ch01-50 вы найдете общие примеры для всего материала книги.

Папка c:\listings\release содержит демонстрационные приложения, которые вы научитесь разрабатывать, прочитав книгу о Delphi XE2.

Предметный указатель

B

Byte Order Mark (BOM) 137

D

DataSnap:
◊ авторизация 749
◊ аутентификация 748
Drag and dock 185
Drag and drop 173

E

Exception:
◊ CreateFmt 250
◊ CreateFmtHelp 250
◊ CreateHelp 250
◊ CreateRes 250
◊ CreateResFmt 250
◊ CreateResFmtHelp 250
◊ CreateResHelp 250

F

FMX.Forms.TApplication:
◊ MainForm 794
◊ OnException 794
◊ OnIdle 794
◊ StyleFileName 794
FMX.Forms.TForm, StyleBook 796
FMX.Forms.TForm3D:
◊ Context 797
◊ OnRender 797
◊ UsingDesignCamera 797
FMX.Types.TBrush:
◊ Bitmap 861
◊ Color 861
◊ Grab 861
◊ Gradient 861
◊ Kind 861
◊ OnChanged 861
◊ OnGradientChanged 862
◊ Resource 861
FMX.Types.TCanvas:
◊ BeginScene 860
◊ BufferBits 860
◊ Buffered 860
◊ BufferHandle 860
◊ Clear 860
◊ ClearRect 860
◊ DrawArc 865
◊ DrawBitmap 866
◊ DrawEllipse 865
◊ DrawLine 865
◊ DrawPath 865
◊ DrawPolygon 865
◊ DrawRect 865
◊ DrawRectSides 865
◊ DrawThumbnail 867
◊ EndScene 860
◊ ExcludeClipRect 867
◊ Fill 860
◊ FillArc 864
◊ FillEllipse 864
◊ FillPath 864
◊ FillPolygon 864
◊ FillRect 864
◊ FillText 865
◊ Font 860
◊ Height 860
◊ IntersectClipRect 867
◊ LoadFontFromStream 865
◊ ResetClipRect 867
◊ SaveState 867
◊ SetClipRects 867
◊ SetCustomDash 862
◊ Stroke 860
◊ StrokeCap 862
◊ StrokeDash 862
◊ StrokeJoin 862
◊ StrokeThickness 862
◊ TextHeight 865
◊ TextToPath 866
◊ TextWidth 865
◊ Width 860
FMX.Types.TFont:
◊ Family 863
◊ Size 863
◊ Style 863

I

IClassFactory:
◊ CreateInstance 681
◊ LockServer 681
IEnumIDList:
◊ Clone 736
◊ Next 736
◊ Reset 736
◊ Skip 736
IHelpSystem:
◊ ShowContextHelp 204
◊ ShowHelp 204
◊ ShowTableOfContents 204
◊ ShowTopicHelp 204
IShellFolder:
◊ BindToObject 733
◊ CompareIDs 738
◊ EnumObjects 735
◊ GetAttributesOf 737
◊ GetDisplayNameOf 734
◊ ParseDisplayName 733
◊ SetNameOf 735

IUnknown:
◊ _AddRef 678
◊ _Release 678
◊ QueryInterface 678

R

RGB 477

T

TAction:
◊ ActionList 308
◊ Category 308
◊ Execute 307
◊ Index 308
◊ OnExecute 307
◊ OnUpdate 308
TActionClientItem:
◊ Action 312
◊ ActionBar 312
◊ Background 313
◊ BackgroundLayout 313
◊ Caption 313
◊ CheckUnused 314
◊ ChildActionBar 312
◊ Color 313
◊ ContextItems 312
◊ HasBackground 314
◊ HasGlyph 314
◊ ImageIndex 313
◊ Items 312
◊ LastSession 314
◊ ShortCut 313
◊ ShowCaption 313
◊ ShowGlyph 313
◊ ShowShortCut 313
◊ UsageCount 314
TActionList:
◊ ActionCount 309
◊ Actions 309
◊ ExecuteAction 309
◊ Images 309
◊ OnChange 310
◊ OnExecute 307, 309
◊ OnStateChange 310
◊ OnUpdate 310
◊ State 309
TActionMainMenuBar:
◊ AnimateDuration 315
◊ AnimationStyle 315
◊ CloseMenu 315
◊ ExpandDelay 315
◊ InMenuLoop 315
◊ OnEnterMenuLoop 316
◊ OnGetPopupClass 316
◊ OnPopup 316
◊ RootMenu 315
◊ Shadows 315
◊ WindowMenu 316
TActionManager:
◊ ActionCount 309
◊ DefaultActionBars 311
◊ ExecuteAction 309
◊ Images 309
◊ OnChange 310
◊ OnExecute 307, 309
◊ OnStateChange 310
◊ OnUpdate 310
◊ State 309
◊ TActionBars 311
TActionToolBar:
◊ ActionClients 314
◊ ActionControls 314
◊ ActionManager 314
◊ HorzSeparator 315
◊ Orientation 314
◊ TActionMainMenuBar 314
◊ VertSeparator 315
TAnimation 810
TAppletApplication:
◊ ControlPanelHandle 661
◊ ModuleCount 661
◊ Modules 661
TAppletModule:
◊ AppletIcon 659
◊ Caption 659
◊ Data 659
◊ Help 659
◊ OnActivate 660
◊ OnInquire 660
◊ OnNewInquire 660
◊ OnStartWParms 660
◊ OnStop 660
◊ ResidIcon 659
◊ ResidInfo 659
◊ ResidName 659
TApplication 193
◊ Active 196
◊ ActiveFormHandle 197
◊ CancelHint 203
◊ CreateForm 200
◊ CurrentHelpFile 204
◊ ExeName 196
◊ Handle 197
◊ HandleMessage 201
◊ HelpCommand 204
◊ HelpContext 204
◊ HelpJump 204
◊ HelpSystem 204
◊ Hint 203
◊ HintColor 203
◊ HintHidePause 203
◊ HintPause 203
◊ HintShortCuts 203
◊ HintShortPause 203
◊ Icon 195
◊ Initalize 199
◊ MainFormHandle 197
◊ MainFormOnTaskBar 200
◊ MessageBox 202
◊ Minimize 202
◊ OnActionExecute 307
◊ OnException 255
◊ OnGetActiveFormHandle 197
◊ OnGetMainFormHandle 197
◊ OnHelp 205
◊ OnHint 203
◊ OnMessage 604
◊ OnMinimize 202
◊ OnRestore 202
◊ Restore 202
◊ Run 200
◊ ShowMainForm 200
◊ Terminate 202
◊ Title 195
TApplicationEvents 205
◊ OnActionExecute 205
◊ OnActionUpdate 205
◊ OnActivate 205
◊ OnDeactivate 205
◊ OnException 255
◊ OnHint 205
◊ OnIdle 205
◊ OnMessage 205
◊ OnMinimize 205
◊ OnModalBegin 205
◊ OnModalEnd 205
◊ OnRestore 205
◊ OnShortCut 205
◊ OnShowHint 205
TArray 32
TBasicAction 164
TBindExpression:
◊ Active 771
◊ ControlComponent 770
◊ ControlExpression 770
◊ Direction 770
◊ Managed 772
◊ OnActivated 771

TBindExpression *(прод.)*:
◊ OnActivating 771
◊ OnAssignedValueEvent 772
◊ OnAssigningValueEvent 772
◊ OnEvalErrorEvent 772
◊ SourceComponent 770
◊ SourceExpression 770
TBindings:
◊ CreateAssociationScope 775
◊ CreateExpression 773
◊ CreateManagedBinding 774
◊ CreateUnmanagedBinding 775
TBitBtn 277
◊ Glyph 278
◊ Kind 277
◊ Layout 278
◊ Margin 278
◊ NumGlyphs 278
◊ Spacing 278
◊ Style 278
TBitmap:
◊ Dormant 504
◊ FreeImage 504
◊ Handle 501
◊ HandleType 500
◊ LoadFromResourceID 503
◊ LoadFromResourceName 503
◊ Mask 501
◊ MaskHandle 501
◊ Monochrome 500
◊ Palette 501
◊ PixelFormat 501
◊ ReleaseHandle 501
◊ ReleaseMaskHandle 501
◊ ReleasePalette 501
◊ TransparentColor 501
◊ TransparentMode 501
TBitmapAnimation:
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnFinish 811
◊ OnProcess 811
◊ Pause 811
◊ Running 811
◊ Start 811
◊ StartValue 813
◊ StopValue 813
TBitmapListAnimation:
◊ AnimationBitmap 814
◊ AnimationCount 814
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnFinish 811
◊ OnProcess 811
◊ Pause 811
◊ Running 811
◊ Start 811
◊ Stop 811
TBrush:
◊ Bitmap 480
◊ Color 479
◊ Style 479
TButton 274
◊ Action 307
◊ Cancel 276
◊ Caption 276
◊ CommandLinkHint 275
◊ Default 276
◊ DisabledImageIndex 277
◊ DropDownMenu 275
◊ ElevationRequired 276
◊ HotImageIndex 277
◊ ImageAlignment 277
◊ ImageIndex 277
◊ ImageMargins 277
◊ Images 277
◊ OnClick 274
◊ OnDropDownClick 275
◊ PressedImageIndex 277
◊ Style 274
◊ StylusHotImageIndex 277
TButtonCategories:
◊ Add 288
◊ AddItem 288
◊ CategoryButtons 288
◊ Clear 288
◊ Count 288
◊ Delete 288
◊ Insert 288
◊ ItemIndex 288
◊ Items 285, 288
◊ VisibleCount 288
TButtonCategory:
◊ Caption 288
◊ CategoryButtons 288
◊ Collapsed 288
◊ Color 288
◊ Data 288
◊ GradientColor 289
◊ Items 288
◊ TextColor 289
TButtonCollection:
◊ Add 289
◊ AddItem 289
◊ Category 289
◊ Clear 289
◊ Count 289
◊ Delete 289
◊ Insert 289
◊ Items 289
TButtonedEdit:
◊ Images 266
◊ LeftButton 266
◊ RightButton 266
◊ Visible 266
TButtonGroup 281
◊ ButtonHeight 283
◊ ButtonOptions 283
◊ ButtonWidth 283
◊ Images 283
◊ ItemIndex 283
◊ Items 281
◊ OnAfterDrawButton 284
◊ OnBeforeDrawButton 284
◊ OnButtonClicked 284
◊ OnDrawButton 284
◊ OnDrawIcon 284
◊ OnHotButton 284
◊ OnReorderButton 284
◊ ScrollIntoView 284
TButtonItem:
◊ Caption 289
◊ Category 289
◊ CategoryButtons 289
◊ ImageIndex 289
TBytesStream 152
◊ Bytes 153
◊ Create 152
TCalloutPanel:
◊ CalloutLength 802
◊ CalloutOffset 802
◊ CalloutPosition 801
◊ CalloutWidth 802
TCanvas:
◊ Arc 492
◊ Brush 487
◊ BrushCopy 495
◊ Chord 492
◊ CopyMode 495
◊ CopyRect 494
◊ DrawFocusRect 492
◊ Ellipse 492
◊ FillRect 487
◊ FloodFill 487
◊ Font 487
◊ FrameRect 492

TCanvas *(прод.)*:
◊ Handle 487
◊ LineTo 490
◊ MoveTo 490
◊ Pen 487
◊ PenPos 490
◊ Pie 492
◊ Pixels 489
◊ PolyBexierTo 493
◊ PolyBezier 493
◊ Polygon 491
◊ Polyline 491
◊ Rectangle 491
◊ RoundRect 492
TCategoryButtons 284
◊ ButtonFlow 286
◊ ButtonHeight 286
◊ ButtonOptions 285
◊ ButtonWidth 286
◊ Categories 285
◊ CurrentCategory 286
◊ FocusedItem 286
◊ GetButtonAt 286
◊ GetButtonRect 286
◊ GetCategoryAt 286
◊ GetCategoryRect 286
◊ GetTargetAt 286
◊ HotButtonColor 288
◊ Images 286
◊ OnAfterDrawButton 287
◊ OnBeforeDrawButton 287
◊ OnButtonClicked 287
◊ OnCancelEdit 287
◊ OnCategoryClicked 287
◊ OnCategoryCollapase 287
◊ OnCopyButton 287
◊ OnDrawButton 287
◊ OnDrawIcon 287
◊ OnDrawText 287
◊ OnEdited 287
◊ OnEditing 287
◊ OnHotButton 287
◊ OnReorderButton 287
◊ OnReorderCategory 287
◊ OnSelectedCategoryChange 287
◊ OnSelectedItemChange 287
◊ RegularButtonColor 288
◊ SelectedButtonColor 288
◊ SelectedItem 286
TCheckBox 278
◊ AllowGrayed 278
◊ Checked 279
◊ State 278
TCheckListBox 330
◊ AllowGrayed 330
◊ AutoComplete 327
◊ AutoCompleteDelay 327
◊ BorderStyle 326
◊ Checked 330
◊ Columns 326
◊ Count 325
◊ ExtendedSelect 324
◊ Header 330
◊ HeaderBackgroundColor 331
◊ HeaderColor 331
◊ IntegralHeight 326
◊ ItemAtPos 325
◊ ItemEnabled 330
◊ ItemRect 326
◊ Items 321, 324
◊ MultiSelect 324
◊ OnClickCheck 331
◊ OnData 329
◊ OnDataFind 329
◊ OnDataObject 329
◊ OnDrawItem 327
◊ OnMeasureItem 328
◊ ScrollWidth 326
◊ SelCount 324
◊ Selected 324
◊ State 330
◊ TabWidth 326
◊ TopIndex 325
TClipboard:
◊ Assign 596
◊ AsText 596
◊ Clear 597
◊ Close 597
◊ FormatCount 595
◊ Formats 595
◊ GetAsHandle 596
◊ GetTextBuf 596
◊ HasFormat 596
◊ Open 597
◊ SetAsHandle 596
◊ SetTextBuf 596
TCollection 142
◊ Add 143
◊ BeginUpdate 143
◊ Clear 143
◊ Count 143
◊ Create 143
◊ Delete 143
◊ Destroy 143
◊ EndUpdate 144
◊ FindItemID 144
◊ Insert 143
◊ ItemClass 143
◊ Items 143
◊ Owner 143
TCollectionItem 142
◊ Collection 142
◊ Create 142
◊ Destroy 142
◊ ID 142
◊ Index 142
TColorAnimation:
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnFinish 811
◊ OnProcess 811
◊ Pause 811
◊ Running 811
◊ Start 811
◊ StartValue 813
◊ Stop 811
◊ StopValue 813
TColorDialog:
◊ Color 447
◊ CustomColors 447
◊ Execute 434
TColorListBox 331
◊ AutoComplete 327
◊ AutoCompleteDelay 327
◊ BorderStyle 326
◊ ColorNames 332
◊ Colors 332
◊ Columns 326
◊ Count 325
◊ DefaultColorColor 333
◊ ExtendedSelect 324
◊ IntegralHeight 326
◊ ItemAtPos 325
◊ ItemRect 326
◊ Items 324
◊ MultiSelect 324
◊ NoneColorColor 333
◊ OnGetColors 333
◊ ScrollWidth 326
◊ SelCount 324
◊ Selected 324, 333
◊ Style 332
◊ TabWidth 326
◊ TopIndex 325
TComboBox:
◊ AddItem 323, 334
◊ Clear 323, 334
◊ ClearSelection 323, 334

TComboBox *(прод.)*:
◊ CopySelection 324, 334
◊ DeleteSelected 323, 334
◊ DroppedDown 334
◊ EditHandle 333
◊ GetCount 323, 334
◊ ItemIndex 323
◊ Items 321, 334
◊ ListHandle 333
◊ MoveSelection 324
◊ SelectAll 323, 334
◊ SelLength 334
◊ SelStart 334
TComboBoxEx 335
◊ Images 336
◊ Items 321, 335
◊ ItemsEx 335
◊ Style 336
◊ StyleEx 337
TComboExItem:
◊ Data 336
◊ ImageIndex 336
◊ Indent 336
◊ OverlayImageIndex 336
◊ SelectedImageIndex 336
◊ Caption 336
TComboExItems:
◊ Add 336
◊ AddItem 336
◊ Clear 336
◊ ComboItems 336
◊ Delete 336
◊ Insert 336
◊ OnCompare 336
◊ Sort 336
◊ SortType 336
TComboTrackBar:
◊ Max 804
◊ Min 804
◊ OnChange 804
◊ OnChangeTracking 804
◊ TrackBar 804
◊ Value 804
TCommonCalendar 422
TCommonDialog:
◊ Execute 450
◊ OnClose 435
◊ OnShow 435
TComObject:
◊ Controller 686
◊ CreateFromFactory 686
◊ Factory 686
◊ Initialize 686
◊ RefCount 686
TComObjectFactory:
◊ ClassID 684
◊ ClassName 684
◊ ComClass 684
◊ ComServer 684
◊ Create 683
◊ CreateComObject 685
◊ ErrorIID 685
◊ Instancing 684
◊ LicString 685
◊ RegisterClassObject 685
◊ ShowErrors 685
◊ SupportsLicensing 685
◊ ThreadingModel 684
◊ UpdateRegistry 685
TComponent 126
◊ ComponentCount 127
◊ ComponentIndex 128
◊ Components 127
◊ Create 127
◊ DestroyComponents 128
◊ FindComponent 128
◊ InsertComponent 128
◊ Name 127
◊ Owner 127
◊ RemoveComponent 128
◊ Tag 127
TComponentList 133
◊ Items 134
TComServer:
◊ HelpFileName 688
◊ Initialize 687
◊ IsInprocServer 688
◊ LoadTypeLib 687
◊ ObjectCount 688
◊ OnLastRelease 687
◊ ServerFileName 688
◊ ServerKey 688
◊ ServerName 688
◊ SetServerName 687
◊ StartMode 688
◊ TypeLib 688
◊ UIInteractive 687
◊ UpdateRegistry 687
TControl 157
◊ Align 161
◊ Anchors 161
◊ AutoSize 159
◊ BeginDrag 174
◊ BoundsRect 159
◊ BringToFront 163
◊ Caption 163
◊ ClientHeight 159
◊ ClientOrigin 159
◊ ClientRect 159
◊ ClientToScreen 160
◊ ClientWidth 159
◊ Color 163
◊ Constraints 160
◊ Dock 189
◊ DockClients 186
◊ DragCursor 174
◊ DragKind 173
◊ DragMode 173
◊ Enabled 162
◊ EndDrag 174
◊ FloatingDockSiteClass 186
◊ Font 163
◊ Height 158
◊ Hide 162
◊ Hint 163
◊ Left 158
◊ ManualDock 189
◊ ManualFloat 189
◊ OnCanResize 160
◊ OnClick 165
◊ OnConstrainedResize 160
◊ OnContextPopup 164
◊ OnDblClick 168
◊ OnDragDrop 175
◊ OnDragOver 175
◊ OnEndDrag 175
◊ OnGesture 165, 456
◊ OnMouseActivate 169—171
◊ OnMouseDown 168
◊ OnMouseMove 171
◊ OnMouseUp 168
◊ OnMouseWheel 172
◊ OnMouseWheelDown 173
◊ OnMouseWheelUp 173
◊ OnResize 160
◊ OnStartDrag 174
◊ Parent 157
◊ ParentColor 163
◊ ParentFont 163
◊ ParentShowHint 164
◊ PopupMenu 164, 293
◊ ScreenToClient 160
◊ SendToBack 163
◊ Show 162
◊ ShowHint 164
◊ Text 163
◊ Top 158
◊ Touch 165, 456
◊ Visible 162
◊ Width 158

TControlBar:
◊ AutoDock 406
◊ AutoSize 407
◊ OnBandDrag 407
◊ OnBandInfo 407
◊ OnBandMove 407
◊ OnBandPaint 407
◊ Picture 407
◊ RowSize 407
TControlScrollBar:
◊ ButtonSize 225
◊ Increment 225
◊ Margin 225
◊ Position 224
◊ Range 224
◊ Size 225
◊ Style 225
TCoolBand:
◊ Bitmap 406
◊ Break 406
◊ Control 406
◊ FixedSize 406
◊ ImageIndex 406
◊ MinHeight 406
◊ MinWidth 406
◊ ParentBitmap 406
◊ Text 406
TCoolBar:
◊ Add 405
◊ Align 405
◊ Bands 405
◊ Count 405
◊ FindBand 405
◊ Images 405
◊ Items 405
◊ OnChange 405
◊ Vertical 405
TCriticalSection:
◊ Enter 590
◊ Leave 590
TCustomCombo 333
TCustomEdit 262
◊ AutoSelect 264
◊ AutoSize 264
◊ CanUndo 265
◊ CharCase 264
◊ Clear 263
◊ ClearSelection 265
◊ CopyToClipboard 265
◊ CutToClipboard 265
◊ GetSelTextBuf 265
◊ HideSelection 264
◊ Modified 265
◊ NumbersOnly 264
◊ OEMConvert 264
◊ OnChange 263
◊ PasswordChar 264
◊ PasteFromClipboard 265
◊ ReadOnly 263
◊ SelectAll 265
◊ SelLength 265
◊ SelStart 265
◊ SelText 265
◊ SetSelTextBuf 265
◊ Text 263
◊ Text MaxLength 264
TCustomizeDlg:
◊ ActionManager 317
◊ FileName 317
◊ LoadFromFile 318
◊ LoadFromStream 318
◊ OnClose 317
◊ OnShow 317
◊ ResetActionBar 318
◊ ResetUsageData 317
◊ SaveToFile 318
◊ SaveToStream 318
◊ Show 317
TCustomListControl 323
TCustomMemoryStream 151
◊ Memory 151
◊ Read 151
◊ SaveToFile 152
◊ SaveToStream 152
TDataModule 243
TDateTimePicker 425
◊ BoldDays 424
◊ Checked 426
◊ Date 422, 425
◊ DateFormat 426
◊ DateTime 422
◊ EndDate 423
◊ FirstDayOfWeek 423
◊ Format 425
◊ Kind 425
◊ MaxDate 423
◊ MaxSelectRange 423
◊ MinDate 423
◊ MultiSelect 423
◊ OnChange 426
◊ OnCloseUp 426
◊ OnDropDown 426
◊ OnGetMonthBoldInfo 424
◊ OnGetMonthInfo 424
◊ OnUserInput 425, 426
◊ ParseInput 425
◊ ShowCheckbox 426
◊ ShowToday 423
◊ ShowTodayCircle 423
◊ Time 425
◊ TMonthCalColors 423
◊ WeekNumbers 423
TDBXTraceInfo:
◊ CustomCategory 744
◊ Message 744
◊ TraceFlag 744
◊ TraceLevel 744
TDependencies, Items 649
TDirectory:
◊ Copy 848
◊ CreateDirectory 848
◊ Delete 87, 848
◊ Exists 848
◊ GetAttributes 848
◊ GetCreationTime 848
◊ GetCurrentDirectory 848
◊ GetDirectories 849
◊ GetDirectoryRoot 849
◊ GetFiles 849
◊ GetFileSystemEntries 850
◊ GetLastAccessTime 848
◊ GetLastWriteTime 849
◊ GetLogicalDrives 81, 848
◊ GetParent 849
◊ IsEmpty 850
◊ IsRelativePath 850
◊ Move 850
◊ SetAttributes 849
◊ SetCurrentDirectory 848
◊ SetLastAccessTime 849
◊ SetLastWriteTime 849
TDockTabSet:
◊ DestinationDockSite 416
◊ FirstIndex 416
◊ ItemWidth 416
◊ OnChange 416
◊ OnDrawTab 416
◊ OnGetImageIndex 416
◊ OnMeasureTab 416
◊ Style 416
◊ TabHeight 416
◊ TabIndex 416
◊ TabPosition 416
◊ Tabs 416
◊ VisibleTabs 416
TDrawGrid 354
◊ CellRect 356
◊ Col 356
◊ ColCount 354
◊ Color 354
◊ ColWidths 354
◊ DefaultColWidth 354

TDrawGrid *(прод.)*:
◊ DefaultDrawing 358
◊ DefaultRowHeight 354
◊ DrawingStyle 354
◊ FixedColor 354
◊ FixedCols 354
◊ GridHeight 355
◊ GridLineWidth 355
◊ GridWidth 355
◊ LeftCol 355
◊ MouseToCell 356
◊ OnColumnMoved 357
◊ OnDrawCell 358
◊ OnFixedCellClick 357
◊ OnGetEditMask 357
◊ OnRowMoved 357
◊ OnSelectCell 357
◊ OnSetEditText 357
◊ OnTopLeftChanged 357
◊ Options 355
◊ Row 356
◊ RowCount 354
◊ RowHeights 354
◊ ScrollBars 355
◊ Selection 356
◊ TopRow 355
◊ VisibleColCount 355
◊ VisibleRowCount 355
TDSAuthenticationManager:
◊ OnUserAuthenticate 749
◊ OnUserAuthorize 749
TDSHTTPService:
◊ Active 747
◊ AuthenticationManager 748
◊ DSHostname 748
◊ DSPort 748
◊ Filters 746
◊ HttpPort 747
◊ RESTContext 748
◊ Server 746
◊ ServerSoftware 748
◊ Trace 748
TDSRoleItem:
◊ ApplyTo 749
◊ AuthorizedRoles 749
◊ DeniedRoles 749
TDSServer:
◊ AutoStart 742
◊ HideDSAdmin 742
◊ LifeCycle 745
◊ OnConnect 743
◊ OnCreateInstance 746
◊ OnDestroyInstance 746
◊ OnDisconnect 743
◊ OnError 744
◊ OnGetClass 745
◊ OnPrepare 744, 746
◊ OnTrace 744
◊ Start 742
◊ Started 742
◊ Stop 742
TDSServerClass, Server 745
TDSTCPServerTransport:
◊ BufferKBSize 747
◊ Filters 746
◊ MaxThreads 747
◊ PoolSize 747
◊ Port 747
◊ Server 746
TEdit:
◊ AutoSelect 264
◊ AutoSize 264
◊ CanUndo 265
◊ CharCase 264
◊ Clear 263
◊ ClearSelection 265
◊ CopyToClipboard 265
◊ CutToClipboard 265
◊ GetSelTextBuf 265
◊ HideSelection 264
◊ Modified 265
◊ NumbersOnly 264
◊ OEMConvert 264
◊ OnChange 263
◊ PasteFromClipboard 265
◊ SelectAll 265
◊ SelLength 265
◊ SelStart 265
◊ SelText 265
◊ SetSelTextBuf 265
◊ Text 263
TEvent:
◊ Create 587
◊ LastError 588
◊ ResetEvent 587
◊ SetEvent 587
◊ WaitFor 587
TExpander:
◊ IsChecked 802
◊ IsExpanded 802
◊ OnCheckChange 802
◊ ShowCheck 802
◊ Text 802
TFile:
◊ AppendAllText 850
◊ Copy 851
◊ Create 149, 850
◊ CreateText 851
◊ Decrypt 851
◊ Delete 851
◊ Encrypt 851
◊ Exists 851
◊ GetAttributes 851
◊ GetCreationTime 851
◊ GetCreationTimeUtc 851
◊ GetLastAccessTime 851
◊ GetLastWriteTime 851
◊ Move 851
◊ OpenRead 149, 851
◊ OpenText 852
◊ OpenWrite 149
◊ ReadAllBytes 852
◊ ReadAllLines 852
◊ Replace 852
◊ SetCreationTime 852
◊ SetLastAccessTime 852
◊ SetLastWriteTime 852
◊ WriteAllLines 852
◊ WriteAllText 852
TFileStream 147
◊ Create 147
◊ Destroy 149
◊ FileName 148
TFindDialog:
◊ FindText 441
◊ Options 442
◊ Position 442
TFloatAnimation:
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnProcess 811
◊ Pause 811
◊ Running 811
◊ Start 811
◊ StartValue 814
◊ Stop 811
◊ StopValue 814
TFlowPanel:
◊ AutoWrap 390
◊ FlowStyle 390
TFont:
◊ Charset 264, 485
◊ Color 485
◊ FontAdapter 485
◊ Handle 485
◊ Height 485
◊ Name 484
◊ Orientation 485
◊ Pitch 485

TFont *(прод.)*:
◊ PixelsPerInch 485
◊ Quality 485
◊ Size 485
◊ Style 485
TFontDialog:
◊ Device 446
◊ Execute 434, 445
◊ Font 445
◊ MaxFontSize 446
◊ MinFontSize 446
TForm 215
◊ Active 220
◊ ActiveControl 223
◊ ActiveMDIChild 236
◊ ActiveOLEControl 224
◊ ArrangeIcons 236
◊ AutoScroll 224
◊ BorderIcons 219
◊ Caption 218
◊ Cascade 236
◊ ClientHandle 228
◊ Close 222, 229
◊ Create 220
◊ DefocusControl 224
◊ DoubleBuffered 225
◊ FocusControl 224
◊ FormState 220
◊ FormStyle 218
◊ GlassFrame 227
◊ Handle 228
◊ Hide 222
◊ HorzScrollBar 224
◊ Icon 219
◊ KeyPreview 232
◊ MDIChildCount 235
◊ MDIChildren 235
◊ Menu 223
◊ ModalResult 222
◊ Next 237
◊ OnActivate 229
◊ OnCanResize 229
◊ OnClose 229, 231
◊ OnCloseQuery 229
◊ OnConstrainedResize 229
◊ OnDeactivate 229
◊ OnDestroy 229, 231
◊ OnHide 229
◊ OnResize 229
◊ OnShortCut 231
◊ OnShow 229
◊ PixelsPerInch 226
◊ PopupMenu 223
◊ Position 218
◊ Previous 237
◊ Release 223
◊ Scaled 226
◊ ScreenSnap 228
◊ Show 221
◊ ShowModal 221
◊ SnapBuffer 228
◊ Tile 236
◊ TileMode 237
◊ Touch 456
◊ VertScrollBar 224
◊ WindowMenu 223
◊ WindowState 219
TFrame 237
TGestureListView,
GestureManager 462
TGestureManager:
◊ LoadFromFile 461
◊ LoadFromStream 461
◊ SaveToFile 461
◊ SaveToStream 461
TGesturePreview,
GestureProvider 462
TGestureRecorder:
◊ GestureManager 463
◊ OnGestureRecorded 463
TGifFrame:
◊ Clear 508
◊ Draw 509
◊ Empty 508
◊ HasBitmap 508
◊ Height 508
◊ Left 508
◊ LoadFromStream 509
◊ SaveToStream 509
◊ StretchDraw 509
◊ Top 508
◊ Width 508
TGIFImage:
◊ Add 509
◊ Animate 507
◊ AnimateLoop 507
◊ AnimationSpeed 507
◊ BackgroundColor 507
◊ BackgroundColorIndex 507
◊ BitsPerPixel 506
◊ ColorReduction 510
◊ ColorResolution 506
◊ Dithering 510
◊ DitherMode 510
◊ GlobalColorMap 509
◊ Images 508
◊ IsTransparent 507
◊ OnAfterPaint 511
◊ OnEndPaint 511
◊ OnLoop 511
◊ OnPaint 511
◊ OnStartPaint 511
◊ OnWarning 511
◊ Optimize 510
◊ OptimizeColorMap 510
◊ ReductionBits 510
◊ ShouldDither 510
◊ Version 506
TGIFImageList:
◊ Add 508
◊ Clear 508
◊ Count 508
◊ Delete 508
◊ Exchange 508
◊ First 508
◊ Frame 508
◊ IndexOf 508
◊ Insert 508
◊ Items 508
◊ Last 508
◊ Move 508
◊ Remove 508
TGlassFrame, SheetOfGlass 227
TGradientAnimation:
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnFinish 811
◊ OnProcess 811
◊ Pause 811
◊ Running 811
◊ Start 811
◊ StartValue 813
◊ Stop 811
◊ StopValue 813
TGraphic:
◊ Empty 498
◊ Height 498
◊ LoadFromFile 497
◊ LoadFromStream 497
◊ Modified 498
◊ OnProgress 498
◊ Palette 498
◊ PaletteModified 498
◊ SaveToFile 498
◊ SaveToStream 497
◊ Transparent 498

TGrid:
◊ CellControlByRow 806
◊ ColumnByPoint 806
◊ ColumnCount 806
◊ Columns 806
◊ OnEdititingDone 807
◊ OnGetValue 806
◊ OnSetValue 806
◊ RowCount 806
TGridPanel:
◊ CellCount 392
◊ CellRect 392
◊ CellSize 392
◊ ColumnCollection 390
◊ ControlCollection 392
◊ ExpandStyle 391
◊ RowCollection 390
TGrpButtonItem:
◊ ButtonGroup 282
◊ Caption 282
◊ Collection 282
◊ Data 282
◊ ImageIndex 282
◊ Index 282
TGrpButtonItems:
◊ Add 282
◊ AddItem 282
◊ ButtonGroup 282
◊ Clear 282
◊ Count 282
◊ Delete 282
◊ Items 282
THandleStream 147
◊ Create 147
◊ Handle 147
THotKey:
◊ HotKey 320
◊ InvalidKeys 320
◊ Modifiers 320
◊ OnChange 320
TIcon:
◊ Handle 499
◊ Height 499
◊ ReleaseHandle 499
◊ Transparent 499
◊ Width 499
TImageList:
◊ Add 519
◊ AddIcon 519
◊ AddMasked 519
◊ BkColor 522
◊ BlendColor 521
◊ Clear 521
◊ Delete 521
◊ Draw 522
◊ DrawingStyle 521
◊ DrawOverlay 522
◊ FileLoad 520
◊ GetBitmap 523
◊ GetIcon 523
◊ GetImageBitmap 523
◊ GetInstRes 520
◊ GetMaskBitmap 523
◊ GetResource 520
◊ Height 518
◊ ImageType 518
◊ Masked 523
◊ Move 519
◊ OnChange 521
◊ Overlay 522
◊ ResInstLoad 520
◊ ResourceLoad 520
◊ Width 518
TIniFile 554
TJPEGImage:
◊ Compress 505
◊ CompressionQuality 505
◊ DIBNeeded 505
◊ JPEGNeeded 505
◊ Performance 505
◊ PixelFormat 506
◊ ProgressiveDisplay 505
◊ ProgressiveEncoding 505
◊ Scale 506
◊ Smoothing 505
TLabel 257
◊ Alignment 259
◊ AutoSize 258
◊ Caption 258
◊ EllipsisPosition 259
◊ FocusControl 259
◊ Layout 259
◊ ShowAccelChar 259
◊ WordWrap 258
TLabeledEdit 266
◊ EditLabel 266
◊ LabelPosition 266
◊ LabelSpacing 267
TLinkLabel 260
◊ Caption 258, 260
◊ OnLinkClick 261
TList 130
◊ Add 131
◊ Capacity 131
◊ Clear 131
◊ Count 131
◊ Delete 131
◊ Exchange 131
◊ Expand 131
◊ Extract 131
◊ ExtractItem 131
◊ First 131
◊ IndexOf 131
◊ Insert 132
◊ Items 131
◊ Last 132
◊ List 131
◊ Move 132
◊ Pack 132
◊ Remove 132
◊ RemoveItem 132
◊ Sort 132
TListBox 324
◊ AddItem 323
◊ AutoComplete 327
◊ AutoCompleteDelay 327
◊ BorderStyle 326
◊ Clear 323
◊ ClearSelection 323
◊ Columns 326
◊ CopySelection 324
◊ Count 325
◊ DeleteSelected 323
◊ ExtendedSelect 324
◊ GetCount 323
◊ IntegralHeight 326
◊ ItemAtPos 325
◊ ItemIndex 323
◊ ItemRect 326
◊ Items 321, 324
◊ MoveSelection 324
◊ MultiSelect 324
◊ OnData 329
◊ OnDataFind 329
◊ OnDataObject 329
◊ OnDrawItem 327
◊ OnMeasureItem 328
◊ ScrollWidth 326
◊ SelCount 324
◊ SelectAll 323
◊ Selected 324
◊ Style 325
◊ TabWidth 326
◊ TopIndex 325
TListColumn:
◊ Alignment 340
◊ AutoSize 340
◊ Caption 340
◊ Index 340
◊ MaxWidth 340
◊ MinWidth 340
◊ Width 340

TListColumns:
◊ Count 339
◊ Items 339
TListGroup:
◊ Footer 346
◊ FooterAlign 346
◊ GroupHeaderImages 346
◊ GroupID 346
◊ GroupView 346
◊ Header 346
◊ HeaderAlign 346
◊ State 346
◊ Subtitle 346
◊ TitleImage 346
TListItem 342
◊ CancelEdit 343
◊ Caption 342
◊ Checked 342
◊ Data 342
◊ Delete 343
◊ EditCaption 343
◊ Focused 342
◊ GetPosition 342
◊ GroupID 342
◊ ImageIndex 339
◊ Index 342
◊ Left 342
◊ OverlayIndex 339
◊ Position 342
◊ Selected 342
◊ SetPosition 342
◊ StateIndex 339
◊ SubItems 342
◊ Top 342
TListItems:
◊ Add 341
◊ AddItem 341
◊ BeginUpdate 341
◊ Clear 341
◊ Count 341
◊ Delete 341
◊ EndUpdate 341
◊ IndexOf 341
◊ Insert 341
◊ Item 340
◊ ItemIndex 341
◊ Owner 341
◊ TopItem 341
◊ ViewOrigin 341
TListView 337
◊ AlphaSort 345
◊ Checkboxes 339
◊ ClearSelection 344
◊ Columns 339
◊ CopySelection 344
◊ DeleteSelected 344
◊ Deleting 343
◊ DisplayRect 347
◊ FindCaption 345
◊ FindData 345
◊ GridLines 339
◊ Groups 346
◊ HideSelection 344
◊ HotTrack 339
◊ HotTrackStyles 339
◊ HoverTime 339
◊ IconOptions 339
◊ ItemFocused 344
◊ Items 337
◊ LargeImages 338
◊ MoveSelection 344
◊ MultiSelect 343
◊ OnChange 344
◊ OnChanging 344
◊ OnCompare 345
◊ OnCustomDraw 347
◊ OnCustomDrawItem 347
◊ OnCustomDrawSubItem 348
◊ OnDrawItem 346
◊ OnEdited 343
◊ OnEditing 343
◊ SelCount 344
◊ SelectAll 343
◊ Selected 344
◊ ShowColumnHeaders 339
◊ SmallImages 338
◊ SortType 345
◊ StateImages 338
◊ ViewStyle 338
◊ VisibleRowCount 340
TMainMenu 292
◊ AutoHotkeys 292
◊ AutoMerge 292
◊ Items 291
◊ Merge 293
◊ OnChange 292
◊ Unmerge 293
TMaskEdit 267
◊ EditMask 267
◊ EditText 267
◊ GetTextLen 267
◊ Text 267
◊ ValidateEdit 268
TMemo 268
◊ CaretPos 268
◊ Lines 268
◊ ScrollBars 268
◊ WantReturns 269
◊ WantTabs 269
◊ WordWrap 268
TMemoryStream 152
◊ Capacity 152
◊ Clear 152
◊ LoadFromFile 152
◊ LoadFromStream 152
◊ SetSize 152
TMenu 291
TMenuItem 294
◊ Action 307
◊ Add 298
◊ AutoCheck 296
◊ AutoHotkeys 295
◊ AutoLineReduction 300
◊ Bitmap 298
◊ Break 300
◊ Caption 294
◊ Checked 295
◊ Clear 300
◊ Count 297
◊ Default 295
◊ Delete 300
◊ Find 298
◊ GetParentMenu 294
◊ GroupIndex 296
◊ ImageIndex 298
◊ IndexOf 297
◊ Insert 298
◊ InsertNewLineAfter 300
◊ InsertNewLineBefore 300
◊ IsLine 301
◊ Items 297
◊ MenuIndex 297
◊ NewBottomLine 300
◊ NewTopLine 300
◊ OnAdvancedDrawItem 302
◊ OnClick 295
◊ OnDrawItem 301
◊ OnMeasureItem 302
◊ Parent 297
◊ RadioItem 296
◊ Remove 300
◊ RethinkHotkeys 295
◊ RethinkLines 301
◊ ShortCut 294
◊ SubMenuImages 298
TMetafile:
◊ CreatedBy 515
◊ Description 515
◊ Enhanced 515

TMetafile *(прод.)*:
◊ Handle 515
◊ Height 515
◊ Inch 515
◊ MMHeight 515
◊ MMWidth 515
◊ Palette 515
◊ Width 515
TMetafileCanvas:
◊ Create 515
◊ CreateWithComment 515
TMonitor:
◊ BoundsRect 475
◊ Handle 474
◊ Height 475
◊ Left 475
◊ MonitorNum 474
◊ Primary 474
◊ Top 475
◊ Width 475
◊ WorkareaRect 475
TMonthCalendar 424
◊ BoldDays 424
◊ Date 422
◊ DateTime 422
◊ EndDate 423
◊ FirstDayOfWeek 423
◊ MaxDate 423
◊ MaxSelectRange 423
◊ MinDate 423
◊ MultiSelect 423
◊ OnGetMonthBoldInfo 424
◊ OnGetMonthInfo 424
◊ ShowToday 423
◊ ShowTodayCircle 423
◊ TMonthCalColors 423
◊ WeekNumbers 423
TMutex:
◊ Acquire 591
◊ Create 591
◊ Release 591
TObject 119
◊ ClassName 123
◊ ClassNameIs 123
◊ ClassParent 123
◊ ClassType 123
◊ Create 121
◊ Destroy 121
◊ FieldAddress 124
◊ Free 123
◊ GetInterface 724
◊ InheritsFrom 123
◊ MethodAddress 123
◊ MethodName 123
◊ UnitName 123
TObjectList 133
◊ Create 133
◊ FindInstanceOf 133
◊ OwnsObjects 133
TOpenDialog:
◊ DefaultExt 436
◊ Execute 434, 438
◊ FileEditStyle 437
◊ FileName 438
◊ Filter 435
◊ FilterIndex 437
◊ HistoryList 437
◊ InitialDir 437
◊ OnCanClose 440
◊ OnFolderChange 439
◊ OnIncludeItem 440
◊ OnSelectionChange 439
◊ OnTypeChange 440
◊ OptionsEx 438
◊ Title 437
TOpenPictureDialog, Execute 434
TOpenTextFileDialog:
◊ EncodingIndex 441
◊ Encodings 441
◊ Execute 434
TPageControl:
◊ ActivePage 417
◊ ActivePageIndex 417
◊ FindNextPage 418
◊ PageCount 417
◊ Pages 417
◊ SelectNextPage 417
TPageScroller 394
◊ ButtonSize 395
◊ Control 394, 395
◊ OnScroll 395
◊ Orientation 395
TPageSetupDialog:
◊ Execute 434
◊ MarginLeft 448
◊ MinMarginBottom 448
◊ MinMarginLeft 448
◊ MinMarginRight 448
◊ MinMarginTop 448
◊ Options 448
◊ PageHeight 447
◊ PageWidth 447
◊ Units 447
TPaintBox, Canvas 487
TPanel:
◊ Alignment 390
◊ BevelEdges 389
◊ BevelInner 389
◊ BevelKind 389
◊ BevelOuter 389
◊ BevelWidth 389
◊ BorderStyle 389
◊ BorderWidth 389
◊ ShowCaption 389
◊ VerticalAlignment 390
TParaAttributes:
◊ Alignment 270
◊ FirstIndent 270
◊ LeftIndent 270
◊ Numbering 270
◊ RightIndent 270
◊ Tab 270
◊ TabCount 271
TPath:
◊ AltDirectorySeparatorChar 854
◊ ChangeExtension 853
◊ Combine 853
◊ DirectorySeparatorChar 854
◊ DriveExists 853
◊ ExtensionSeparatorChar 854
◊ GetAttributes 854
◊ GetDirectoryName 853
◊ GetExtendedPrefix 853
◊ GetExtension 853
◊ GetFileName 853
◊ GetFileNameWithoutExtension 854
◊ GetFullPath 854
◊ GetGUIDFileName 853
◊ GetHomePath 85, 854
◊ GetInvalidFileNameChars 854
◊ GetInvalidPathChars 854
◊ GetPathRoot 854
◊ GetRandomFileName 854
◊ GetTempFileName 854
◊ GetTempPath 854
◊ HasExtension 854
◊ HasValidFileNameChars 853
◊ HasValidPathChars 853
◊ IsDriveRooted 853
◊ IsExtendedPrefixed 853
◊ IsPathRooted 854
◊ IsRelativePath 853
◊ IsUNCPath 853
◊ IsUNCRooted 853
◊ IsValidFileNameChar 853
◊ IsValidPathChar 853
◊ MatchesPattern 853

TPath *(прод.)*:
◊ PathSeparator 854
◊ SetAttributes 854
◊ VolumeSeparatorChar 854
TPathAnimation:
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnFinish 811
◊ OnProcess 811
◊ Path 814
◊ Pause 811
◊ Rotate 814
◊ Running 811
◊ Start 811
◊ Stop 811
TPen:
◊ Color 481
◊ Handle 483
◊ Mode 483
◊ Style 481
◊ Width 481
TPersistent 125
TPicture:
◊ Bitmap 516
◊ Icon 516
◊ LoadFromClipboardFormat 516
◊ Metafile 516
◊ SaveToClipboardFormat 516
TPNGImage:
◊ AddtEXt 514
◊ AddzTXt 514
◊ Canvas 512
◊ CompressionLevel 513
◊ CreateAlpha 514
◊ CreateBlank 512
◊ Draw 513
◊ DrawUsingPixelInformation 513
◊ Empty 512
◊ Filters 513
◊ InterlaceMethod 513
◊ Pixels 512
◊ RemoveTransparency 514
◊ Resize 514
◊ TransparentColor 514
TPopupMenu:
◊ Alignment 293
◊ AutoHotkeys 292
◊ AutoPopup 293
◊ Items 291
◊ MenuAnimation 294
◊ OnChange 292
◊ OnPopup 293
◊ Popup 293
◊ PopupAlignment 293
◊ PopupComponent 293
TPosition:
◊ DefaultValue 784
◊ Empty 784
◊ OnChange 784
◊ Point 784
◊ Vector 784
◊ X 784
◊ Y 784
TPrintDialog:
◊ Copies 451
◊ Execute 434
◊ FromPage 450
◊ MaxPage 450
◊ MinPage 450
◊ Options 450
◊ PrintRange 450
◊ PrintToFile 451
◊ ToPage 450
TPrinter:
◊ Abort 542
◊ Aborted 542
◊ BeginDoc 542
◊ Canvas 541
◊ Capabilities 541
◊ EndDoc 542
◊ NewPage 542
◊ Orientation 541
◊ PageHeight 541
◊ PageNumber 542
◊ PageWidth 541
◊ PrinterIndex 540
◊ Printers 540
◊ Title 542
TPrinterSetupDialog 449
◊ Execute 434
TRadioButton:
◊ Alignment 280
◊ Checked 280
TRadioGroup 280
◊ Columns 281
◊ ItemIndex 281
◊ Items 281
TRectAnimation:
◊ Delay 811
◊ Duration 810
◊ Interpolation 811
◊ Inverse 812
◊ Loop 811
◊ OnFinish 811
◊ OnProcess 811
◊ Pause 811
◊ Running 811
◊ Start 811
◊ StartValue 814
◊ Stop 811
◊ StopValue 814
TRegistry:
◊ Access 559
◊ CloseKey 560
◊ Create 559
◊ CreateKey 560
◊ CurrentKey 559
◊ CurrentPath 559
◊ DeleteKey 560
◊ DeleteValue 562
◊ GetDataInfo 563
◊ GetDataSize 563
◊ GetDataType 563
◊ GetKeyInfo 562
◊ GetKeyNames 562
◊ GetValueNames 561, 563
◊ HasSubKeys 560
◊ KeyExists 560
◊ LazyWrite 560
◊ LoadKey 563
◊ MoveKey 560
◊ OpenKey 560
◊ OpenKeyReadOnly 560
◊ ReadBinaryData 561
◊ ReadBool 561
◊ ReadCurrency 561
◊ ReadDate 561
◊ ReadDateTime 561
◊ ReadFloat 561
◊ ReadInteger 561
◊ ReadString 561
◊ ReadTime 561
◊ RegistryConnect 559
◊ RenameValue 562
◊ ReplaceKey 564
◊ RestoreKey 563
◊ RootKey 559
◊ SaveKey 563
◊ UnLoadKey 564
◊ ValueExists 561
◊ WriteBinaryData 562
◊ WriteBool 561
◊ WriteCurrency 562
◊ WriteDate 562
◊ WriteDateTime 562
◊ WriteFloat 562
◊ WriteInteger 562

TRegistry *(прод.)*:
◊ WriteString 561
◊ WriteTime 562
TRegistryIniFile:
◊ Create 555
◊ DeleteKey 557
◊ EraseSection 557
◊ ReadBool 556
◊ ReadDate 556
◊ ReadDateTime 556
◊ ReadFloat 556
◊ ReadInteger 556
◊ ReadSection 556
◊ ReadSections 556
◊ ReadSectionValues 556
◊ ReadString 556
◊ ReadTime 556
◊ RegIniFile 555
◊ SectionExists 556
◊ WriteBool 557
◊ WriteDate 557
◊ WriteDateTime 557
◊ WriteFloat 557
◊ WriteInteger 557
◊ WriteString 557
◊ WriteTime 557
TReplaceDialog:
◊ FindText 441
◊ OnReplace 444
◊ Options 442
◊ Position 442
◊ ReplaceText 444
TRichEdit 269
◊ FindText 270
◊ Lines 269
◊ OnProtectChange 272
◊ OnResizeRequest 273
◊ OnSaveClipboard 273
◊ OnSelectionChange 271
◊ paragraph 269
◊ PlainText 269
◊ Print 270
TSaveDialog:
◊ DefaultExt 436
◊ Execute 434, 438
◊ FileEditStyle 437
◊ FileName 438
◊ Filter 435
◊ HistoryList 437
◊ InitialDir 437
◊ OnCanClose 440
◊ OnFolderChange 439
◊ OnIncludeItem 440
◊ OnSelectionChange 439
◊ OnTypeChange 440
◊ OptionsEx 438
◊ Title 437
TSavePictureDialog, Execute 434
TSaveTextFileDialog:
◊ EncodingIndex 441
◊ Encodings 441
TScreen:
◊ ActiveControl 473
◊ ActiveCustomForm 473
◊ ActiveForm 473
◊ Cursor 472
◊ Cursors 472
◊ CustomFormCount 473
◊ CustomForms 473
◊ DataModuleCount 473
◊ DataModules 473
◊ DesktopHeight 472
◊ DesktopLeft 472
◊ DesktopTop 472
◊ DesktopWidth 472
◊ DisableAlign 473
◊ EnableAlign 473
◊ Fonts 472
◊ FormCount 473
◊ Forms 473
◊ Height 472
◊ HintFont 473
◊ IconFont 473
◊ MenuFont 473
◊ MonitorCount 474
◊ MonitorFromPoint 474
◊ MonitorFromRect 474
◊ MonitorFromWindow 474
◊ Monitors 474
◊ OnActiveControlChange 473
◊ OnActiveFormChange 473, 474
◊ PixelsPerInch 472
◊ ResetFonts 473
◊ Width 472
TScrollBox 393
◊ AutoScroll 394
◊ HorzScrollBar 394
◊ VertScrollBar 394
TSemaphore:
◊ Acquire 592
◊ Create 592
◊ Release 593
TService:
◊ ErrCode 653
◊ LogMessage 653
◊ OnExecute 652
◊ ServiceThread 652
◊ Terminated 653
◊ WaitHint 653
◊ Win32ErrCode 653
TServiceApplication:
◊ RegisterServices 654
◊ ServiceCount 654
TServiceThread, ProcessRequests 652
TServiceType:
◊ AfterInstall 651
◊ AfterUninstall 651
◊ AllowPause 650
◊ AllowStop 650
◊ BeforeInstall 651
◊ BeforeUninstall 651
◊ DisplayName 647
◊ ErrorSeverity 650
◊ Interactive 648
◊ LoadGroup 649
◊ Name 647
◊ OnContinue 651
◊ OnPause 651
◊ OnShutdown 651
◊ OnStart 651
◊ OnStop 651
◊ Param 649
◊ Password 648
◊ ReportStatus 650
◊ ServiceStartName 648
◊ ServiceType 648
◊ StartType 649
◊ Status 649
◊ TagID 649
TSplitter:
◊ AutoSnap 396
◊ MinSize 396
◊ OnCanResize 396
◊ OnMoved 396
◊ ResizeStyle 396
TSQLConnection:
◊ AfterConnect 751
◊ AfterDisconnect 751
◊ BeforeConnect 751
◊ BeforeDisconnect 751
◊ Close 750
◊ Connected 750
◊ ConnectionState 752
◊ LoginPrompt 751
◊ OnLogin 752
◊ Open 750
◊ Params 750
TStaticText:
◊ BorderStyle 260
◊ Caption 258

TStatusBar:
◊ Add 411
◊ AutoHint 410
◊ Delete 411
◊ Insert 411
◊ Items 409
◊ OnCreatePanelClass 411
◊ OnDrawPanel 410
◊ OnHint 410
◊ Panels 409
◊ SimplePanel 408
◊ SimpleText 408
◊ Style 411
◊ Событие 410
TStatusPanel, Text 409
TStream 145
◊ CopyFrom 147
◊ Position 145
◊ Read 146
◊ ReadBuffer 147
◊ Seek 146
◊ Size 145
◊ Write 146
◊ WriteBuffer 147
TStringGrid 359
◊ CellControlByRow 806
◊ CellRect 356
◊ Cells 359, 807
◊ Col 356
◊ ColCount 354
◊ Color 354
◊ Cols 359
◊ ColumnByPoint 806
◊ ColumnCount 806
◊ Columns 806
◊ ColWidths 354
◊ DefaultColWidth 354
◊ DefaultDrawing 358
◊ DefaultRowHeight 354
◊ DrawingStyle 354
◊ FixedColor 354
◊ FixedRows 354
◊ GridHeight 355
◊ GridLineWidth 355
◊ GridWidth 355
◊ LeftCol 355
◊ MouseToCell 356
◊ Objects 361
◊ OnColumnMoved 357
◊ OnDrawCell 358
◊ OnEdititingDone 807
◊ OnFixedCellClick 357
◊ OnGetEditMask 357
◊ OnGetEditText 357
◊ OnGetValue 806
◊ OnRowMoved 357
◊ OnSelectCell 357
◊ OnSetEditText 357
◊ OnSetValue 806
◊ OnTopLeftChanged 357
◊ Options 355
◊ Row 356
◊ RowCount 354, 806
◊ RowHeights 354
◊ Rows 359
◊ ScrollBars 355
◊ Selection 356
◊ TopRow 355
◊ VisibleColCount 355
◊ VisibleRowCount 355
TStringList 140
◊ CaseSensitive 141
◊ Duplicates 141
◊ Find 140
◊ OnChange 141
◊ OnChanging 141
◊ Sort 140
◊ Sorted 140
TStrings 134
◊ Add 135
◊ AddObject 135
◊ AddStrings 135
◊ Append 135
◊ BeginUpdate 140
◊ Clear 135
◊ CommaText 138
◊ Count 135
◊ DefaultEncoding 135
◊ Delete 136
◊ DelimitedText 138
◊ Delimiter 138
◊ Encoding 137
◊ EndUpdate 140
◊ Equals 136
◊ Exchange 136
◊ GetText 138
◊ IndexOf 139
◊ IndexOfName 139
◊ IndexOfObject 139
◊ Insert 136
◊ InsertObject 136
◊ LineBreak 138
◊ LoadFromFile 137
◊ LoadFromStream 137
◊ Move 136
◊ Names 139
◊ NameValueSeparator 139
◊ Objects 135
◊ Put 136
◊ PutObject 136
◊ QuoteChar 138
◊ SaveToFile 137
◊ SaveToStream 137
◊ SetText 138
◊ Strings 135
◊ Text 138
◊ ToObjectArray 138
◊ ToStringArray 138
◊ UpdateCount 140
◊ Values 139
◊ WriteBOM 137
TStringStream 153
◊ Create 153
◊ DataString 154
◊ Encoding 153
◊ ReadString 153
◊ WriteString 153
TStyleBook, Resource 795
TStyleManager:
◊ ActiveStyle 213
◊ IsValidStyle 211
◊ LoadFromFile 211
◊ LoadFromResource 211
◊ SetStyle 212
◊ StyleNames 212
◊ TrySetStyle 212
TTabControl:
◊ IndexOfTabAt 415
◊ MultiLine 414
◊ MultiSelect 414
◊ OnChange 415
◊ OnChanging 415
◊ OnDrawTab 415
◊ OnGetImageIndex 415
◊ OwnerDraw 415
◊ RaggedRight 414
◊ RowCount 414
◊ ScrollOpposite 414
◊ ScrollTabs 414
◊ Style 413
◊ TabHeight 414
◊ TabIndex 414
◊ TabPosition 413
◊ TabRect 415
◊ Tabs 413
◊ TabWidth 414
TTabSet:
◊ FirstIndex 416
◊ ItemWidth 416

TTabSet *(прод.)*:
◊ OnChange 416
◊ OnDrawTab 416
◊ OnGetImageIndex 416
◊ OnMeasureTab 416
◊ OnTabAdded 416
◊ OnTabRemoved 416
◊ Style 416
◊ TabHeight 416
◊ TabIndex 416
◊ TabPosition 416
◊ Tabs 416
◊ VisibleTabs 416
TTabSheet:
◊ Highlighted 419
◊ ImageIndex 419
◊ PageControl 418
◊ PageIndex 419
◊ TabIndex 419
◊ TabVisible 419
TTaskDialog:
◊ Button 452
◊ Caption 452
◊ CommonButtons 452
◊ DefaultButton 452
◊ ExpandedText 452
◊ Flags 451
◊ FooterText 452
◊ OnButtonClicked 453
◊ OnRadioButtonClicked 453
◊ OnTimer 453
◊ OnVerificationClicked 452
◊ ProgressBar 453
◊ RadioButtons 452
◊ Text 452
◊ Title 452
◊ VerificationText 452
TTCPClient:
◊ Connect 636
◊ Connected 636
◊ Disconnect 636
TTCPServer:
◊ Accept 635
◊ BlockMode 634
◊ BytesReceived 634
◊ BytesSent 634
◊ Close 636
◊ Domain 632
◊ GetSocketAddr 632
◊ Listening 635
◊ LocalDomainName 633
◊ LocalHost 632
◊ LocalHostAddr 633
◊ LocalHostName 633
◊ LocalPort 632
◊ LookupHostAddr 633
◊ LookupHostName 633
◊ LookupPort 633
◊ LookupProtocol 633
◊ MapDomain 632
◊ MapSockType 631
◊ OnAccept 635
◊ OnCreateHandle 635
◊ OnDestroyHandle 636
◊ OnListening 635
◊ OnReceive 633
◊ OnSend 633
◊ PeekBuf 633
◊ Protocol 630
◊ ReceiveBuf 633
◊ Receiveln 633
◊ RemoteHost 632
◊ RemotePort 632
◊ SendBuf 633
◊ Sendln 633
◊ SendStream 633
◊ ServerSocketThread 634
◊ SockType 631
TTCPSocket:
◊ BlockMode 632
◊ BytesReceived 634
◊ BytesSent 634
◊ Domain 632
◊ GetSocketAddr 632
◊ LocalDomainName 633
◊ LocalHost 632
◊ LocalHostAddr 633
◊ LocalHostName 633
◊ LocalPort 632
◊ LookupHostAddr 633
◊ LookupHostName 633
◊ LookupPort 633
◊ LookupProtocol 633
◊ MapDomain 632
◊ MapSockType 631
◊ OnReceive 633
◊ OnSend 633
◊ PeekBuf 633
◊ ReceiveBuf 633
◊ Receiveln 633
◊ RemoteHost 632
◊ RemotePort 632
◊ SendBuf 633
◊ Sendln 633
◊ SendStream 633
◊ SockType 631
TTextAttributes:
◊ DefAttributes 271
◊ SelAttributes 271
TThread:
◊ Create 577
◊ FatalException 579
◊ FreeOnTerminate 579
◊ Handle 579
◊ IsSingleProcessor 579
◊ OnTerminate 579
◊ Priority 581
◊ ProcessorCount 579
◊ Queue 582
◊ Resume 578
◊ ReturnValue 579
◊ SetReturnValue 579
◊ Sleep 578
◊ Start 577, 578
◊ Suspend 578
◊ Synchronize 582
◊ Terminate 578
◊ Terminated 578
◊ WaitFor 580
TTimer:
◊ Enabled 421
◊ Interval 421
◊ OnTimer 421
TToolBar 398
◊ ButtonCount 403
◊ ButtonHeight 403
◊ Buttons 403
◊ ButtonWidth 403
◊ Caption 404
◊ Customizable 403
◊ CustomizeKeyName 403
◊ CustomizeValueName 403
◊ DisabledImages 404
◊ Flat 404
◊ HotImages 404
◊ Images 403
◊ Indent 404
◊ List 404
◊ Menu 404
◊ RowCount 404
◊ ShowCaptions 403
◊ Transparent 404
◊ Wrapable 404
TToolButton 398
◊ AllowAllUp 399
◊ AllowTextButtons 400
◊ Down 399
◊ DropdownMenu 400
◊ Grouped 399

TToolButton *(прод.)*:
◊ Indeterminate 399
◊ Marked 399
TTouchKeyboard:
◊ KeyCaptions 464
◊ Layout 463
◊ RepeatDelay 464
◊ RepeatRate 464
TTouchManager, GestureManager 456
TTrayIcon:
◊ Animate 206
◊ AnimateInterval 206
◊ BalloonFlags 207
◊ BalloonHint 207
◊ BalloonTimeout 207
◊ BalloonTitle 207
◊ Icon 206
◊ IconIndex 206
◊ Icons 206
◊ OnAnimate 208
◊ OnBalloonClick 207
◊ OnClick 207
◊ OnDblClick 207
◊ OnMouseDown 208
◊ OnMouseMove 207
◊ OnMouseUp 207
◊ PopupMenu 207
◊ ShowBalloonHint 207
◊ Visible 207
TTreeNode 371
◊ AbsoluteIndex 372
◊ Collapse 376
◊ Count 373
◊ Cut 371
◊ Data 371
◊ Delete 375
◊ DeleteChildren 375
◊ Expand 376
◊ Expanded 376
◊ Focused 371
◊ FullCollapse 376
◊ FullExpand 376
◊ GetFirstChild 373
◊ GetLastChild 373
◊ GetNext 373
◊ GetNextChild 373
◊ GetNextSibling 373
◊ GetNextVisible 374
◊ GetPrev 373
◊ GetPrevChild 373
◊ GetPrevSibling 373
◊ GetPrevVisible 374
◊ HasAsParent 372
◊ HasChildren 373
◊ ImageIndex 375
◊ Index 372
◊ IsEditing 383
◊ IsVisible 374
◊ Item 373
◊ Level 372
◊ OverlayIndex 375
◊ Owner 371
◊ Parent 372
◊ Selected 371
◊ SelectedIndex 375
◊ StateIndex 375
◊ Text 371
◊ TreeView 372
TTreeNodes 377
◊ AlphaSort 382
◊ CustomSort 382
◊ Item 378
TTreeView:
◊ Add 378
◊ AddChild 378
◊ AddChildFirst 379
◊ AddChildObject 379
◊ AddChildObjectFirst 379
◊ AddFirst 378
◊ AddObject 379
◊ AddObjectFirst 379
◊ AutoExpand 376
◊ BeginUpdate 380
◊ ChangeDelay 370
◊ Clear 382
◊ Count 378
◊ Delete 382
◊ Deselect 369
◊ DisplayRect 385
◊ EndUpdate 380
◊ FindNextToSelect 371
◊ GetFirstNode 378
◊ GetHitTestInfoAt 370
◊ GetNodeAt 370
◊ Handle 378
◊ Images 375, 384
◊ Insert 379
◊ InsertNode 379
◊ InsertObject 379
◊ LoadFromFile 369
◊ LoadFromStream 369
◊ MultiSelect 370
◊ MultiSelectStyle 370
◊ OnAddition 380
◊ OnAdvancedCustomDraw 384
◊ OnAdvancedCustomDrawItem 384
◊ OnChange 370
◊ OnChanging 369
◊ OnCollapsed 376
◊ OnCollapsing 376
◊ OnCompare 381
◊ OnCreateNodeClass 380
◊ OnCustomDrawItem 385
◊ OnDeletion 382
◊ OnEdited 383
◊ OnEditing 383
◊ OnExpanded 376
◊ OnExpanding 376
◊ OnGetImageIndex 384
◊ OnGetSelectedIndex 384
◊ Owner 378
◊ SaveToFile 369
◊ SaveToStream 369
◊ Selected 369
◊ SelectionCount 370
◊ Selections 370
◊ ShowButtons 383
◊ ShowLines 383
◊ ShowRoot 384
◊ SortType 381
◊ StateImages 375, 384
◊ событие 376
TTypedComObject, GetClassInfo 686
TUDPClient:
◊ Connect 636
◊ Connected 636
◊ Disconnect 636
TUDPSocket:
◊ BlockMode 632
◊ BytesReceived 634
◊ BytesSent 634
◊ Domain 632
◊ GetSocketAddr 632
◊ LocalDomainName 633
◊ LocalHost 632
◊ LocalHostAddr 633
◊ LocalHostName 633
◊ LocalPort 632
◊ LookupHostAddr 633
◊ LookupHostName 633
◊ LookupPort 633
◊ LookupProtocol 633
◊ MapDomain 632
◊ MapSockType 631
◊ OnReceive 633
◊ OnSend 633

TUDPSocket *(прод.)*:
◊ PeekBuf 633
◊ ReceiveBuf 633
◊ Receiveln 633
◊ RemoteHost 632
◊ RemotePort 632
◊ SendBuf 633
◊ Sendln 633
◊ SendStream 633
◊ SockType 631
TUpDown:
◊ AlignButton 279
◊ ArrowKeys 279
◊ Associate 279
◊ Increment 279
◊ Max 279
◊ Min 279
◊ OnChanging 280
◊ OnChangingEx 280
◊ Orientation 280
◊ Position 279
◊ Thousands 279
◊ Wrap 279
TValueListEditor:
◊ ColCount 365
◊ DeleteRow 364
◊ FindRow 364
◊ InsertRow 363
◊ Keys 364
◊ OnGetEditMask 365
◊ OnGetPickList 365
◊ OnStringsChanging 365
◊ OnValidate 365
◊ RowCount 365
◊ Strings 363
◊ Values 364
TWICImage:
◊ EncoderContainerFormat 517
◊ Handle 517
◊ ImageFormat 517
◊ ImagingFactory 517
TWinControl 178
◊ AutoSize 186
◊ Broadcast 181
◊ CanFocus 182
◊ ContainsControl 181
◊ ControlAtPos 181
◊ ControlCount 181
◊ Controls 181
◊ CustomAlignInsertBefore 182
◊ DockClientCount 186
◊ DockClients 186
◊ DockSite 185
◊ Focused 182
◊ GetTabOrderList 183
◊ InsertControl 181
◊ MouseInClient 171
◊ OnAlignInsertBefore 182
◊ OnAlignPosition 181
◊ OnAlignPosition() 161
◊ OnDockDrop 188
◊ OnDockOver 188
◊ OnEndDock 189
◊ OnEnter 183
◊ OnExit 183
◊ OnGetSiteInfo 188
◊ OnKeyDown 183
◊ OnKeyPress 184
◊ OnKeyUp 183
◊ OnStartDock 187
◊ OnUnDock 189
◊ RemoveControl 181
◊ SetFocus 182
◊ ShowControl 181
◊ TabOrder 183
◊ TabStop 183
TZCompressionStream:
◊ CompressionRate 154
◊ Create 154
◊ OnProgress 154
TZDecompressionStream 155
◊ Create 155

V

VCL Style Designer, утилита 213

W

Win32 API:
◊ CreateFileMapping 609
◊ DeleteService 647
◊ GetSystemDirectory 85
◊ GetTempPath 85
◊ GetVolumeInformation 82
◊ GetWindowsDirectory 85
◊ SHBrowseForFolder 729
◊ SHGetSpecialFolderPath 731
Windows API:
◊ AlphaBlend 528
◊ CallNamedPipe 624
◊ ChangeDisplaySettings 468
◊ ChangeServiceConfig 647
◊ CloseServiceHandle 642
◊ CombineTransform 536
◊ ControlService 645
◊ CopyFile 87
◊ CreateMailslot 615
◊ CreateNamedPipe 620
◊ CreateProcess 261, 566
◊ CreateToolhelp32Snapshot 572
◊ DisconnectNamedPipe 623
◊ EnumDisplayDevices 466
◊ EnumDisplaySettings 467
◊ EnumServicesStatus 644
◊ EnumWindows 602
◊ ExitProcess 571
◊ ExtCreatePen 481, 483
◊ ExtractAssociatedIcon 499
◊ FindFirstChangeNotification 656
◊ FindWindow 602
◊ FreeLibrary 673
◊ GetCurrentProcess 569
◊ GetDeviceCaps 470
◊ GetDiskFreeSpaceEx 82
◊ GetDriveType 81
◊ GetFileSizeEx 87
◊ GetGraphicsMode 534, 535
◊ GetMailslotInfo 615
◊ GetMapMode 533
◊ GetNamedPipeHandleState 623
◊ GetNamedPipeInfo 623
◊ GetPriorityClass 569
◊ GetProcAddress 673
◊ GetProcessTimes 570
◊ GetROP2 525
◊ GetThreadTimes 582
◊ GetWindowModuleFileName 260
◊ LoadBitmap 502
◊ LoadLibrary 673
◊ MapViewOfFile 609
◊ MoveFile 87
◊ OpenProcess 568
◊ OpenSCManager 642
◊ OpenService 644
◊ PostMessage 606
◊ Process32First 572
◊ Process32Next 572
◊ QueryServiceConfig 646
◊ QueryServiceStatus 645
◊ RegisterClipboardFormat 598
◊ RegisterWindowMessage 604
◊ SendMessage 606

Windows API *(прод.)*:
◊ SetMailslotInfo 615
◊ SetMapMode 533
◊ SetNamedPipeHandleState 622
◊ SetPriorityClass 569
◊ SetROP2 525
◊ SetViewportOrgEx 531
◊ SetVolumeLabel 83
◊ SetWindowOrgEx 532
◊ SetWorldTransform 535
◊ SHGetDesktopFolder 732
◊ SHGetPathFromIDList 730
◊ StartService 644
◊ TerminateProcess 572
◊ TerminateThread 578
◊ WaitNamedPipe 624

А

Анимация 808
Аргумент 58
Аффинные преобразования 530

Б

Буфер обмена (clipboard) 594
Блокировка взаимная 582

В

Видеоустройство 465
Внешнее объявление процедур и функций 65
Выражение 36

Г

Генерация случайного числа:
◊ RandG 823
◊ Random 823
◊ Randomize 823
Графика 497
Группа:
◊ кнопок 281
◊ переключателей 280

Д

Дескриптор процесса 566
Диалоги и сообщения:
◊ CreateMessageDialog 431
◊ InputBox 431
◊ InputQuery 431
◊ MessageDlg 429
◊ MessageDlgPos 430
◊ MessageDlgPosHelp 430
◊ PromptForFileName 432
◊ SelectDirectory 432
◊ ShowMessage 427
◊ ShowMessageFmt 428
◊ ShowMessagePos 427
◊ TaskMessageDlg 430
Диапазон 19
Директива:
◊ {$E cpl} 660
◊ Cdecl 667
◊ dynamic 101
◊ external 65
◊ forward 63
◊ Forward 63
◊ inline 65
◊ overload 62
◊ Pascal 667
◊ Register 667
◊ reintroduce 105
◊ static 101
◊ virtual 101
◊ компилятора 8
▫ {$ASSERTIONS OFF} 253
▫ {$ASSERTIONS ON} 253
▫ {$MAXSTACKSIZE} 62
▫ {$MINSTACKSIZE} 62
▫ {$WARN SYMBOL_PLATFORM OFF} 690
Диспинтерфейс 703

Ж

Жест 454

З

Запись 27
◊ упакованная 29
Защищенная секция:
◊ try..except 245, 254
◊ try..finally 245
Значок 499
◊ приложения в области уведомлений 206

И

Идентификатор 14
◊ квалифицированный 14
◊ неквалифицированный 14
Инкапсуляция 92
Интерфейс 676
◊ IClassFactory 681
◊ IUnknown 678
◊ диспетчерский 703
◊ дуальный 703
◊ исходящий 711
Исключительная ситуация 244
◊ EAbort 249, 251
◊ EAbstractError 249
◊ EAssertionFailed 249, 252
◊ EConvertError 249
◊ EExternal 249
◊ EHeapException 249
◊ EInOutError 249
◊ EInvalidCast 249
◊ EOSError 249
◊ EPackageError 249
◊ EPropReadOnly 249
◊ EPropWriteOnly 249
◊ ESafecallException 249
◊ EVariantError 249
◊ Exception 249

К

Календарь 422, 424
Канал 619

Каталог:
◊ дата/время создания 88
◊ перемещение 87
▫ и переименование 87
◊ поиск 83
◊ проверка существования 85
◊ системный 85
◊ создание 86
◊ удаление 86
Категория кнопок 284
Кисть 479
Клавиатура виртуальная 463
Класс 91, 93
◊ абстрактный 93, 121
◊ опережающее объявление 97
◊ экземпляр 93
Клиентская область 159
Кнопка 274
◊ с рисунком 277
Команда 164
Комментарий 7
Компиляция 9
◊ в режиме отладки 9
◊ оптимизация параметров 9
◊ режим выпуска конечного продукта 9
Компоновщик 9
Константа 13
◊ глобальная 13
◊ локальная 13
◊ область объявления 13
◊ строка-ресурс 13
Контроллер автоматизации 702
Координаты:
◊ мировые 530, 536
◊ страничные 530
◊ устройства 530
◊ физические 530
Критическая секция 590

Л

Линия 490

М

Макрос:
◊ GetBValue 478
◊ GetGValue 478
◊ GetRValue 478
◊ RGB 478
Маркер последовательности байтов 137
Массив 31
◊ вариантный 34
◊ динамический 33
Меню:
◊ главное 292
◊ дочернее 297
◊ разделитель 300
◊ элемент 294
◊ элемент в виде флажка 295
Метка 257
◊ ссылка 260
Метод:
◊ виртуальный 101
◊ динамический 101
◊ класса 102
◊ объекта 91
◊ статический 101
Множество 26
Модальный результат 221
Модуль:
◊ Clipbrd 594
◊ ComServ 687, 691
◊ CtlPanel 659, 660
◊ FileCtrl 432, 847
◊ GIFimg 506
◊ GraphUtil 489
◊ IOUtils 81
◊ Jpeg 441, 504
◊ Masks 847
◊ Messages 194
◊ pngimage 511
◊ Printers 540
◊ Registry 555
◊ ShareMem 844
◊ ShellApi 728
◊ ShlObj 728
◊ Tlhelp32 573
◊ WinSock 631
◊ zlib 154
Модульность 92
Мышь, указатель 472
Мьютекс 590

Н

Наследование 93

О

Область:
◊ вывода 531
◊ объявления:
▫ констант 13
▫ переменных 12
Обработка ошибок, OleCheck 732
Обслуживание ИС, Abort 251
Объект 90
◊ Application 195
Объектно-ориентированное программирование (ООП) 90
Объявление процедур опережающее 63
Округление чисел:
◊ Abs 820
◊ Ceil 820
◊ Floor 820
◊ Frac 820
◊ Int 820
◊ Odd 820
◊ Round 820
◊ SimpleRoundTo 820
◊ Trunc 820
Оператор:
◊ @ 24
◊ ^ 23
◊ AND 38
◊ begin..end 41
◊ break 50
◊ case 43
◊ continue 50
◊ goto 45
◊ if..then..else 42
◊ in 27
◊ NOT 38
◊ OR 38
◊ with..do 45
◊ XOR 38
◊ арифметический 36
◊ класса 105
◊ конкатенации 37
◊ логический 38
◊ множеств 40
◊ отношения 40
◊ перехода 45
◊ приведения типа 106
◊ присваивания 36
▫ адреса 24
◊ сдвига 39
◊ селектор 43
◊ условный 42
◊ цикла 46
Операция, бинарная растровая 484, 524
Очередь сообщений 192, 601

П

Панель 388

Параметр:
◊ функции:
▫ константа 58
▫ переменная 60
▫ формальный 58
◊ цикла 47
Перегрузка функций 62
Передача параметра:
◊ по адресу 60
◊ по значению 59
Переключатель 280
Переменная 11
◊ Application 193
◊ DateSeparator 842
◊ DecimalSeparator 828
◊ ThousandSeparator 828
◊ TimeSeparator 842
◊ глобальная 12
◊ локальная 12, 53
◊ область объявления 12
Перо 481
Печать 539
◊ выбор принтера 540
◊ изображения 544
◊ многострочного текста 543
◊ предварительный просмотр 545
Пиксел 158
Планка 397
Подпрограмма 53
Подсказка 203
Поле:
◊ ввода:
▫ ввода данных 262
▫ с маской 267
▫ с меткой 266
◊ многострочное текстовое 268
▫ с форматированием 269
◊ статический текст 260
Полиморфизм 93
Полоса 397
Поток:
◊ данных 145
◊ приоритет 581
Преобразование:
◊ даты/времени:
▫ DateTimeToFileDate 840
▫ DateTimeToStr 840
▫ DateTimeToString 841
▫ DateTimeToSystemTime 832
▫ DateTimeToTimeStamp 831
▫ DateToStr 840
▫ DecodeDate 832
▫ DecodeDateFully 832
▫ DecodeDateMonthWeek 840
▫ DecodeDateTime 840
▫ DecodeTime 832
▫ EncodeDate 832
▫ EncodeDateMonthWeek 840
▫ EncodeDateTime 840
▫ EncodeTime 832
▫ FileDateToDateTime 840
▫ FloatToDateTime 840
▫ FormatDateTime 841
▫ StrToDate 841
▫ StrToDateTime 841
▫ StrToTime 841
▫ SystemTimeToDateTime 832
▫ TimeStampToDateTime 831
▫ TimeToStr 841
▫ TryEncodeDate 832
▫ TryEncodeTime 832
◊ типов 826
▫ CurrToStr 826
▫ FloatToStr 57, 826
▫ FloatToStrF 826
▫ FormatCurr 826
▫ FormatFloat 826
▫ IntToHex 826
▫ IntToStr 826
▫ IntToStrDef 826
▫ StrToCurr 826
▫ StrToInt 826
Препроцессор 8
Приведение типов 19, 20
Приложение:
◊ MDI 233
◊ SDI 233
◊ многопоточное 582
Принтер 540
Процедура 53
◊ анонимная 67
◊ внешнее объявление 65
◊ опережающее объявление 63
◊ управления программой:
▫ Abort 845
▫ Break 845
▫ Continue 845
▫ Exit 845
▫ Halt 845
▫ RunError 845
Процесс 565
◊ дескриптор 566

P

Работа с памятью 843
Редактор списка значений 363
Реестр 553
Режим отображения:
◊ MM_ANISOTROPIC 533
◊ MM_HIENGLISH 533
◊ MM_HIMETRIC 533
◊ MM_ISOTROPIC 533
◊ MM_LOENGLISH 533
◊ MM_LOMETRIC 533
Рекурсия 65

C

Свойство объекта 91
Связывание:
◊ позднее 101
◊ статическое 101
Семафор 592
Сервер автоматизации 702
Сервис 641
Сетка 354
◊ строк 359
Слот 613
Служба 641
Событие:
◊ AfterConnect 751
◊ AfterDisconnect 751
◊ AfterInstall 651
◊ AfterUninstall 651
◊ BeforeConnect 751
◊ BeforeDisconnect 751
◊ BeforeInstall 651
◊ BeforeUninstall 651
◊ OnAccept 635
◊ OnActionExecute 205, 307
◊ OnActionUpdate 205
◊ OnActivate 205, 229, 660
◊ OnActivated 771
◊ OnActivating 771
◊ OnActiveControlChange 473
◊ OnActiveFormChange 473, 474
◊ OnAddition 380
◊ OnAdvancedCustomDraw 384
◊ OnAdvancedCustomDrawItem 384
◊ OnAdvancedDrawItem 302
◊ OnAfterDrawButton 284, 287
◊ OnAlignInsertBefore 182
◊ OnAlignPosition 161, 181
◊ OnAssignedValueEvent 772

Событие *(прод.)*:
◊ OnAssigningValueEvent 772
◊ OnBandDrag 407
◊ OnBandInfo 407
◊ OnBandPaint 407
◊ OnBeforeDrawButton 284, 287
◊ OnButtonClicked 284, 287, 453
◊ OnCancelEdit 287
◊ OnCanClose 440
◊ OnCanResize 160
◊ OnCategoryClicked 287
◊ OnCategoryCollapase 287
◊ OnChange 141, 263, 292, 310, 320, 344, 370, 405, 415, 426
◊ OnChanging 141, 280, 344, 369, 415
◊ OnChangingEx 280
◊ OnClick 165, 295
◊ OnClickCheck 331
◊ OnClose 229, 231, 317, 435
◊ OnCloseQuery 229
◊ OnCloseUp 426
◊ OnCollapsed 376
◊ OnColumnMoved 357
◊ OnCompare 381
◊ OnCompare 345
◊ OnConnect 743
◊ OnConstrainedResize 160, 229
◊ OnContextPopup 164
◊ OnContinue 651
◊ OnCopyButton 287
◊ OnCreateHandle 635
◊ OnCreateInstance 746
◊ OnCreateNodeClass 380
◊ OnCreatePanelClass 411
◊ OnCustomDraw 347
◊ OnCustomDrawItem 347, 385
◊ OnCustomDrawSubItem 348
◊ OnData 329
◊ OnDataFind 329
◊ OnDataObject 329
◊ OnDblClick 168
◊ OnDeactivate 205, 229
◊ OnDeletion 382
◊ OnDestroy 229, 231
◊ OnDestroyHandle 636
◊ OnDestroyInstance 746
◊ OnDisconnect 743
◊ OnDockDrop 188
◊ OnDockOver 188
◊ OnDragDrop 175
◊ OnDragOver 175
◊ OnDrawButton 284, 287
◊ OnDrawCell 358
◊ OnDrawIcon 284, 287
◊ OnDrawItem 301, 327, 346
◊ OnDrawTab 415
◊ OnDrawText 287
◊ OnDropDown 426
◊ OnDropDownClick 275
◊ OnEditButtonClick 365
◊ OnEdited 287, 343, 383
◊ OnEditing 287, 343, 383
◊ OnEndDock 189
◊ OnEndDrag 175
◊ OnEnter 183
◊ OnEnterMenuLoop 316
◊ OnError 744
◊ OnEvalErrorEvent 772
◊ OnException 255
◊ OnExecute 307, 309
◊ OnExit 183
◊ OnExpanded 376
◊ OnExpanding 376
◊ OnFixedCellClick 357
◊ OnFolderChange 439
◊ OnGesture 165, 456
◊ OnGestureRecorded 463
◊ OnGetActiveFormHandle 197
◊ OnGetColors 333
◊ OnGetEditMask 357, 365
◊ OnGetImageIndex 384
◊ OnGetImageIndex 415
◊ OnGetMainFormHandle 197
◊ OnGetMonthBoldInfo 424
◊ OnGetMonthInfo 424
◊ OnGetPickList 365
◊ OnGetPopupClass 316
◊ OnGetSelectedIndex 384
◊ OnGetSiteInfo 188
◊ OnHelp 205
◊ OnHide 229
◊ OnHint 203, 205
◊ OnHotButton 284, 287
◊ OnIdle 205
◊ OnIncludeItem 440
◊ OnInquire 660
◊ OnKeyDown 183
◊ OnKeyPress 184
◊ OnKeyUp 183
◊ OnLastRelease 687
◊ OnLinkClick 261
◊ OnLogin 752
◊ OnMeasureItem 302, 328
◊ OnMeasureTab 416
◊ OnMessage 205, 604
◊ OnMinimize 202, 205
◊ OnModalBegin 205
◊ OnModalEnd 205
◊ OnMouseActivate 169
◊ OnMouseDown 168
◊ OnMouseEnter 170
◊ OnMouseLeave 171
◊ OnMouseMove 171
◊ OnMouseUp 168
◊ OnMouseWheel 172
◊ OnMouseWheelDown 173
◊ OnMouseWheelUp 173
◊ OnNewInquire 660
◊ OnPause 651
◊ OnPopup 293, 316
◊ OnPrepare 744, 746
◊ OnProgress 154, 498
◊ OnProtectChange 272
◊ OnRadioButtonClicked 453
◊ OnReceive 633
◊ OnReorderButton 284, 287
◊ OnReorderCategory 287
◊ OnReplace 444
◊ OnResize 160, 229
◊ OnResizeRequest 273
◊ OnRestore 202, 205
◊ OnRowMoved 357
◊ OnSaveClipboard 273
◊ OnScroll 395
◊ OnSelectCell 357
◊ OnSelectedCategoryChange 287
◊ OnSelectedItemChange 287
◊ OnSelectionChange 271, 439
◊ OnSend 633
◊ OnSetEditText 357
◊ OnShortCut 205, 231
◊ OnShow 229, 317, 435
◊ OnShowHint 205
◊ OnShutdown 651
◊ OnStart 651
◊ OnStartDock 187
◊ OnStartDrag 174
◊ OnStartWParms 660
◊ OnStateChange 310
◊ OnStop 651, 660
◊ OnStringsChange 365
◊ OnStringsChanging 365
◊ OnTabAdded 416
◊ OnTabRemoved 416
◊ OnTerminate 579
◊ OnTimer 421, 453
◊ OnTopLeftChanged 357

Событие *(прод.)*:
◊ OnTrace 744
◊ OnTypeChange 440
◊ OnUnDock 189
◊ OnUpdate 308, 310
◊ OnUserAuthenticate 749
◊ OnUserAuthorize 749
◊ OnUserInput 425, 426
◊ OnValidate 365
◊ OnVerificationClicked 452
◊ TLastReleaseEvent 687
◊ объекта 91
Соглашение о вызовах:
◊ Pascal 667
◊ Register 667
Сокет 629
Сообщение:
◊ WM_COPYDATA 608
◊ пользовательское 604
Список 321, 324
◊ выбора цвета 331
◊ графических элементов 340
◊ комбинированный 333
▫ улучшенный 335
◊ просмотра 337
◊ флажков 330
Справочная система 204
Сравнение строк ANSI 825
Сток 711, 713
Строка-ресурс 13

Т

Таймер 421
Тип данных 14
◊ TFormState 220
◊ вариантный 24
◊ действительный 20
◊ порядковый 16
◊ простой 16
▫ логический (булев) 17
▫ перечисление 18
▫ символьный 16
▫ целые числа 16
◊ строковый 21
◊ указатель 22
◊ файловый 68

У

Удаление пробелов 825
Указатель 22
◊ неопределенный 23
◊ нетипизированный 24

Ф

Файл 68
◊ Borlndmm.dll 844
◊ атрибуты 89
◊ дата/время создания 88
◊ двоичный 79
◊ перемещение, переименование 87
◊ поиск 83
◊ проверка существования 85
◊ размер 87
◊ расширение:
▫ bmp 500
▫ cpl 660
▫ dfm 125, 216
▫ dib 500
▫ dpr 7
▫ dproj 7
▫ emf 515
▫ exe 7, 9
▫ gdf 462
▫ gif 506
▫ ico 499
▫ png 511
▫ res 7
▫ style 794
▫ tiff 517
▫ vsf 211
▫ wmf 515
▫ xfm 125
◊ текстовый 77
◊ типизированный 68
◊ удаление 86
Флажок 278
Форма 215
◊ активная 220
◊ главная 215
◊ дескриптор 228
◊ доступная 220
◊ дочерняя 218
◊ закрытие 222
◊ интерфейс 218
◊ меню 223
◊ модальная 221, 276
◊ немодальная 221
◊ прозрачная 226
◊ размытая 227
◊ родительская 218, 223
◊ скрытие 222
◊ уничтожение экземпляра 223
◊ фокус 223
Формат представления числа:
◊ научный 57
◊ обычный 57
Форматирование:
◊ даты/времени:
▫ FormatDateTime 841
◊ строк ANSI 825
▫ Format 827, 828
Фрейм 237
Функция 55
◊ CoCreateInstance 682
◊ CoCreateInstanceEx 682
◊ CoGetClassObject 682
◊ FreeAndNil 844
◊ GetLongHint 164
◊ GetShortHint 164
◊ GradientFillCanvas 488
◊ GraphUtil:
▫ ScaleImage 504
◊ Printer 540
◊ Slice 491
◊ анонимная 66
◊ внешнее объявление 65
◊ встроенная 65
◊ даты/времени:
▫ CompareDate 836
▫ CompareDateTime 836
▫ CompareTime 837
▫ CurrentYear 831
▫ Date 831
▫ DateOf 834
▫ DayOf 834
▫ DayOfTheMonth 835
▫ DayOfTheWeek 834, 835
▫ DayOfTheYear 834
▫ DayOfWeek 831, 834
▫ DaysBetween 837
▫ DaySpan 837
▫ EndOfADay 839
▫ EndOfTheMonth 838
▫ EndOfTheWeek 839
▫ EndOfTheYear 838
▫ HourOf 834
▫ HourOfTheDay 835
▫ HourOfTheMonth 835
▫ HourOfTheWeek 835
▫ HourOfTheYear 834
▫ HoursBetween 837
▫ HourSpan 837
▫ IncDay 836
▫ IncHour 836
▫ IncMilliSecond 836
▫ IncMinute 836
▫ IncMonth 831
▫ IncSecond 836
▫ IncWeek 836
▫ IncYear 836
▫ IsLeapYear 831
▫ MilliSecondOf 834

Функция *(прод.)*:
◊ даты/времени *(прод.)*:
▫ MilliSecondOfTheDay 835
▫ MilliSecondOfTheHour 836
▫ MilliSecondOfTheMinute 836
▫ MilliSecondOfTheMonth 835
▫ MilliSecondOfTheSecond 836
▫ MilliSecondOfTheWeek 835
▫ MilliSecondOfTheYear 835
▫ MilliSecondsBetween 837
▫ MilliSecondSpan 838
▫ MinuteOf 834
▫ MinuteOfTheDay 835
▫ MinuteOfTheHour 835
▫ MinuteOfTheMonth 835
▫ MinuteOfTheWeek 835
▫ MinuteOfTheYear 835
▫ MinutesBetween 837
▫ MinuteSpan 837
▫ MonthOf 834
▫ MonthOfTheYear 834
▫ MonthsBetween 837
▫ MonthSpan 837
▫ Now 831
▫ RecodeDate 839
▫ RecodeDateTime 840
▫ RecodeDay 839
▫ RecodeHour 839
▫ RecodeMilliSecond 839
▫ RecodeMinute 839
▫ RecodeMonth 839
▫ RecodeSecond 839
▫ RecodeTime 840
▫ RecodeYear 839
▫ ReplaceDate 831
▫ ReplaceTime 831
▫ SameDate 837
▫ SameDateTime 836
▫ SameTime 837
▫ SecondOf 834
▫ SecondOfTheDay 835
▫ SecondOfTheHour 835
▫ SecondOfTheMinute 836
▫ SecondOfTheMonth 835
▫ SecondOfTheWeek 835
▫ SecondOfTheYear 835
▫ SecondsBetween 837
▫ SecondSpan 837
▫ StartOfADay 839
▫ StartOfTheMonth 838
▫ StartOfTheWeek 838
▫ StartOfTheYear 838
▫ Time 831
▫ TimeOf 834
▫ WeekOf 834
▫ WeekOfTheMonth 835
▫ WeeksBetween 837
▫ WeekSpan 837
▫ YearOf 834
▫ YearsBetween 837
▫ YearSpan 837
◊ вхождения в диапазон:
▫ Hi 821
▫ High 821
▫ Lo 821
▫ Low 821
▫ VarInRange 821
◊ для работы с памятью:
▫ Addr 24
▫ AllocMem 843
▫ CompareMem 844
▫ Dispose 843
▫ FreeMem 843
▫ GetHeapStatus 843
▫ GetMem 843
▫ GetMemoryManager 843
▫ IsMemoryManagerSet 843
▫ New 843
▫ ReallocMem 843
▫ SetMemoryManager 843
▫ SysFreeMem 843
▫ SysGetMem 843
▫ SysReallocMem 844
◊ для работы с указателями:
▫ Assigned 844
▫ SizeOf 844
◊ для работы с файлами:
▫ Append 79
▫ AssignFile 69
▫ BlockRead 79
▫ BlockWrite 79
▫ ChangeFileExt 846
▫ CloseFile 69
▫ CreateDir 86
▫ DateTimeToFileDate 88
▫ DeleteFile 86
▫ DiskFree 82
▫ DiskSize 82
▫ Eof 71
▫ Erase 76
▫ ExcludeTrailingBackslash 846
▫ ExpandFileName 846
▫ ExpandUNCFileName 846
▫ ExtractFileDir 846
▫ ExtractFileDrive 846
▫ ExtractFileExt 846
▫ ExtractFileName 846
▫ ExtractFilePath 846
▫ ExtractRelativePath 846
▫ ExtractShortPathName 846
▫ FileAge 88
▫ FileDateToDateTime 88
▫ FileExists 73, 85
▫ FileGetAttr 89
▫ FileGetDate 88
▫ FilePos 72
▫ FileSetAttr 89
▫ FileSetDate 88
▫ FileSetReadOnly 89
▫ FileSize 72
▫ FindClose 84
▫ FindFirst 83
▫ FindNext 84
▫ FindSearch 85
▫ ForceDirectories 86
▫ GetCurrentDir 85
▫ IncludeTrailingBackslash 846
▫ IsPathDelimiter 847
▫ MatchesMask 847
▫ MinimizeName 847
▫ ProcessPath 847
▫ Read 70
▫ ReadLn 77
▫ RemoveDir 86
▫ Rename 76
▫ Reset 70
▫ Rewrite 70
▫ Seek 72
▫ SeekEof 77
▫ SeekEoln 77
▫ SetCurrentDir 85
▫ Truncate 76
▫ Write 70
▫ WriteLn 77
◊ ИС:
▫ Assert 252
◊ массивов:
▫ MaxIntValue 821
▫ MaxValue 821
▫ MinIntValue 821
▫ MinValue 821
▫ Sum 821
▫ SumInt 821
▫ SumOfSquares 821
▫ SumsAndSquares 821
▫ VarArrayCreate 34
◊ математическая:
▫ Exp 819
▫ Frexp 819
▫ IntPower 819
▫ Ldexp 819
▫ Ln 819
▫ LnXP1 819

Функция *(прод.)*:
◊ математическая *(прод.)*:
▫ Log10 819
▫ Log2 819
▫ LogN 819
▫ Power 819
▫ Sign 819
▫ Sqr 819
▫ Sqrt 819
◊ опережающее объявление 63
◊ перегрузка 62
◊ порядковая:
▫ Dec 20
▫ High 20
▫ Inc 20
▫ Low 20
▫ Odd 19
▫ Ord 19
▫ Pred 20
▫ Succ 20
◊ рекурсивная 65
◊ статистическая:
▫ Max 821
▫ Mean 823
▫ MeanAndStdDev 823
▫ Min 821
▫ Norm 823
▫ PopnVariance 823
▫ PopnVariance 823
▫ StdDev 823
▫ Sum 823
▫ SumAndSquares 823
▫ SumInt 823
▫ SumOfSquares 823
▫ TotalVariance 823
▫ Variance 823
◊ строк ANSI:
▫ AdjustLineBreaks 827
▫ AnsiQuotedStr 826
▫ AnsiReverseString 827
▫ AnsiSameText 825
▫ CompareStr 825
▫ CompareText 825
▫ Concat 827
▫ Copy 827
▫ Delete 826
▫ FmtStr 825
▫ Format 825
▫ Insert 827
▫ Length 827
▫ LowerCase 825
▫ Pos 827
▫ QuotedStr 825, 826
▫ Trim 825
▫ TrimLeft 825
▫ TrimRight 825
▫ UpperCase 825
◊ тригонометрическая:
▫ ArcCos 822
▫ ArcCosh 822
▫ ArcSin 822
▫ ArcSinh 822
▫ ArcTan 822
▫ ArcTan2 822
▫ ArcTanh 822
▫ Cos 822
▫ Cosh 822
▫ Cotan 822
▫ CycleToRad 822
▫ DegToRad 822
▫ GradToRad 822
▫ Hypot 822
▫ Pi 822
▫ RadToDeg 822
▫ RadToGrad 822
▫ Sin 822
▫ SinCos 822
▫ Sinh 822
▫ Tan 822
◊ финансовая:
▫ DoubleDecliningBalance 824
▫ FutureValue 824
▫ InterestPayment 824
▫ InterestRate 824
▫ InternalRateOfReturn 824
▫ NetPresentValue 824
▫ NumberOfPeriods 824
▫ Payment 824
▫ PeriodPayment 824
▫ PresentValue 824
▫ SLNDepreciation 824
▫ SYDDepreciation 824

Х

Холст 487

Ц

Цвет 477
Цикл:
◊ break 50
◊ continue 50
◊ for..downto 46
◊ for..in..do 48
◊ for..to 46
◊ repeat..until 49
◊ while..do 48
◊ вложенный 50
◊ обработки сообщений 192
◊ с параметром 46
◊ с постусловием 49
◊ с предусловием 48

Ш

Шаблон маски 267
Шрифт 484