



Петр Дарахвелидзе
Евгений Марков

Программирование в

Delphi 7



+Дискета

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



МАСТЕР ПРОГРАММ

**Петр Дарахвелидзе
Евгений Марков**

Программирование в Delphi 7

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.06
ББК 32.973.26-018.2
Д20

Дарахвелидзе П. Г., Марков Е. П.

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

ISBN 5-94157-116-X

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

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.10.02.

Формат 70×100¹/₁₆. Печать офсетная. Усл. печ. л. 63,21.

Тираж 6000 экз. Заказ №

"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

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

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

Содержание

ЧАСТЬ I. ОБЪЕКТНАЯ КОНЦЕПЦИЯ DELPHI 7.....	15
Глава 1. Объектно-ориентированное программирование	16
Объект и класс.....	17
Поля, свойства и методы.....	20
События.....	23
Инкапсуляция.....	27
Наследование.....	27
Полиморфизм.....	28
Методы.....	30
Перегрузка методов.....	34
Области видимости.....	35
Объект изнутри.....	38
Резюме.....	42
Глава 2. Библиотека визуальных компонентов VCL и ее базовые классы	44
Иерархия базовых классов.....	44
Класс <i>TObject</i>	47
Класс <i>TPersistent</i>	49
Класс <i>TComponent</i>	51
Базовые классы элементов управления.....	53
Класс <i>TControl</i>	54
Группа свойств <i>Visual</i> . Местоположение и размер элемента управления.....	54
Выравнивание элемента управления.....	56
Внешний вид элемента управления.....	57
Связь с родительским элементом управления.....	59
Класс <i>TWinControl</i>	60
Класс <i>TCustomControl</i>	63
Класс <i>TGraphicControl</i>	63
Резюме.....	63

Глава 3. Обработка исключительных ситуаций	64
Исключительная ситуация как класс	64
Защитные конструкции языка Object Pascal.....	69
Блок <i>try..except</i>	70
Блок <i>try..finally</i>	72
Использование исключительных ситуаций	74
Протоколирование исключительных ситуаций.....	76
Коды ошибок в исключительных ситуациях.....	78
Исключительная ситуация <i>EAbort</i>	82
Функция <i>Assert</i>	82
Резюме	83
Глава 4. Кроссплатформенное программирование для Linux	84
Проект CLX	86
Объектная концепция кроссплатформенного программирования	87
Библиотека компонентов CLX	88
Сходства и различия визуальных компонентов CLX и VCL.....	90
Особенности программирования для Linux	92
Приложения баз данных для Linux	94
Internet-приложения для Linux	94
Резюме	95
ЧАСТЬ II. ИНТЕРФЕЙС И ЛОГИКА ПРИЛОЖЕНИЯ	97
Глава 5. Элементы управления Win32	98
Что такое библиотека ComCtl32	98
Многостраничный блокнот — компоненты <i>TTabControl</i> и <i>TPageControl</i>	100
Компонент <i>TToolBar</i>	105
Компонент <i>TImageList</i>	110
Компоненты <i>TTreeView</i> и <i>TListView</i>	112
Календарь	125
Компонент <i>TMonthCalendar</i>	126
Компонент <i>TDateTimePicker</i>	127
Панель состояния <i>TStatusBar</i>	129
Расширенный комбинированный список <i>TComboBoxEx</i>	130
Создание нового компонента на базе элементов управления из библиотеки ComCtl32	131
Резюме	141
Глава 6. Элементы управления Windows XP	142
Пользовательский интерфейс Windows XP.....	142
Манифест Windows XP	143
Компонент <i>TXPManifest</i>	145
Включение манифеста Windows XP в ресурсы приложения.....	145
Визуальные стили и темы оформления	146
Визуальные стили в Delphi.....	147
Theme API	149

Компоненты настройки цветовой палитры.....	151
Резюме.....	152
Глава 7. Списки и коллекции	153
Список строк.....	154
Класс <i>TStrings</i>	154
Класс <i>TStringList</i>	155
Список указателей.....	162
Класс <i>TList</i>	163
Пример использования списка указателей.....	166
Коллекции.....	170
Класс <i>TCollection</i>	171
Класс <i>TCollectionItem</i>	172
Резюме.....	173
Глава 8. Действия (Actions) и связанные с ними компоненты	174
Действия. Компонент <i>TActionList</i>	175
События, связанные с действиями.....	176
Свойства, распространяемые на клиентов действия.....	178
Прочие свойства.....	179
Стандартные действия.....	180
Категория <i>Edit</i>	183
Категория <i>Search</i>	183
Категория <i>Help</i>	183
Категория <i>File</i>	183
Категория <i>Dialog</i>	184
Категория <i>Window</i>	184
Категория <i>Tab</i>	184
Категория <i>List</i>	184
Категория <i>Internet</i>	185
Категория <i>Format</i>	187
Категория <i>Dataset</i>	187
Категория <i>Tools</i>	187
Компонент <i>TActionManager</i>	187
Изменение и настройка внешнего вида панелей.....	189
Ручное редактирование коллекций панелей и действий.....	191
Резюме.....	194
Глава 9. Файлы и устройства ввода/вывода.....	195
Использование файловых переменных. Типы файлов.....	195
Операции ввода/вывода.....	197
Ввод/вывод с использованием функций Windows API.....	204
Отложенный (асинхронный) ввод/вывод.....	208
Контроль ошибок ввода/вывода.....	210
Атрибуты файла. Поиск файла.....	211
Потоки.....	213
Базовые классы <i>TStream</i> и <i>THandleStream</i>	213
Класс <i>TFileStream</i>	215

Класс <i>TMemoryStream</i>	217
Класс <i>TStringStream</i>	218
Оповещение об изменениях в файловой системе.....	218
Использование отображаемых файлов.....	220
Резюме	223
Глава 10. Использование графики	224
Графические инструменты Delphi	224
Класс <i>TFont</i>	225
Класс <i>TPen</i>	226
Класс <i>TBrush</i>	227
Класс <i>TCanvas</i>	227
Класс <i>TGraphic</i>	233
Класс <i>TPicture</i>	235
Класс <i>TMetafile</i>	237
Класс <i>TIcon</i>	238
Класс <i>TBitmap</i>	238
Графический формат JPEG. Класс <i>TJPEGImage</i>	243
Компонент <i>TImage</i>	245
Использование диалогов для загрузки и сохранения графических файлов.....	247
Класс <i>TClipboard</i>	254
Класс <i>TScreen</i>	256
Вывод графики с использованием отображаемых файлов	259
Класс <i>TAnimate</i>	263
Резюме	264
ЧАСТЬ III. ПРИЛОЖЕНИЯ БАЗ ДАННЫХ	265
Глава 11. Архитектура приложений баз данных	266
Как работает приложение баз данных	268
Модуль данных.....	271
Подключение набора данных.....	272
Настройка компонента <i>TDataSource</i>	274
Отображение данных	276
Резюме	278
Глава 12. Набор данных.....	279
Абстрактный набор данных.....	281
Стандартные компоненты	286
Компонент таблицы	287
Компонент запроса	289
Компонент хранимой процедуры	292
Индексы в наборе данных.....	293
Механизм подключения индексов.....	294
Список описаний индексов	295
Описание индекса	295
Использование описаний индексов	297
Параметры запросов и хранимых процедур	298

Класс <i>TParams</i>	301
Класс <i>TParam</i>	302
Состояния набора данных.....	304
Резюме.....	307
Глава 13. Поля и типы данных	308
Объекты полей.....	309
Статические и динамические поля	311
Класс <i>TField</i>	313
Виды полей	317
Поля синхронного просмотра.....	317
Вычисляемые поля.....	320
Внутренние вычисляемые поля	321
Агрегатные поля	321
Объектные поля.....	322
Типы данных	323
Ограничения.....	328
Резюме.....	331
Глава 14. Механизмы управления данными.....	333
Связанные таблицы.....	334
Отношение "один-ко-многим".....	334
Отношение "многие-ко-многим".....	336
Поиск данных.....	337
Поиск по индексам.....	337
Поиск в диапазоне	338
Поиск по произвольным полям.....	339
Фильтры	340
Быстрый переход к помеченным записям.....	342
Диапазоны.....	344
Резюме.....	346
Глава 15. Компоненты отображения данных	347
Классификация компонентов отображения данных	347
Табличное представление данных	349
Компонент <i>TDBGrid</i>	349
Компонент <i>TDBCtrlGrid</i>	359
Навигация по набору данных	361
Представление отдельных полей	364
Компонент <i>TDBText</i>	364
Компонент <i>TDBEdit</i>	365
Компонент <i>TDBCheckBox</i>	365
Компонент <i>TDBRadioGroup</i>	366
Компонент <i>TDBListBox</i>	366
Компонент <i>TDBComboBox</i>	366
Компонент <i>TDBMemo</i>	367
Компонент <i>TDBImage</i>	367
Компонент <i>TDBRichEdit</i>	368

Синхронный просмотр данных	368
Механизм синхронного просмотра	369
Компонент <i>TDBLookupListBox</i>	372
Компонент <i>TDBLookupComboBox</i>	372
Графическое представление данных	372
Резюме	375
ЧАСТЬ IV. ТЕХНОЛОГИИ ДОСТУПА К ДАННЫМ	377
Глава 16. Процессор баз данных Borland Database Engine	378
Архитектура и функции BDE.....	379
Псевдонимы баз данных и настройка BDE	383
Интерфейс прикладного программирования BDE.....	392
Соединение с источником данных.....	401
Компоненты доступа к данным.....	406
Класс <i>TBDEDataSet</i>	406
Класс <i>TDBDataSet</i>	412
Компонент <i>TTable</i>	413
Компонент <i>TQuery</i>	419
Компонент <i>TStoredProc</i>	421
Резюме	423
Глава 17. Технология dbExpress	424
Драйверы доступа к данным	425
Соединение с сервером баз данных	426
Управление наборами данных	431
Транзакции.....	434
Использование компонентов наборов данных	435
Класс <i>TCustomSQLDataSet</i>	435
Компонент <i>TSQLDataSet</i>	438
Компонент <i>TSQLTable</i>	438
Компонент <i>TSQLQuery</i>	439
Компонент <i>TSQLStoredProc</i>	439
Компонент <i>TSimpleDataSet</i>	440
Способы редактирования данных	443
Интерфейсы dbExpress.....	447
Интерфейс <i>ISQLDriver</i>	447
Интерфейс <i>ISQLConnection</i>	448
Интерфейс <i>ISQLCommand</i>	449
Интерфейс <i>ISQLCursor</i>	450
Отладка приложений с технологией dbExpress	451
Распространение приложений с технологией dbExpress	453
Резюме	454
Глава 18. Сервер баз данных InterBase и компоненты InterBase Express	455
Механизм доступа к данным InterBase Express.....	456
Компонент <i>TIBDatabase</i>	456
Компонент <i>TIBTransaction</i>	461

Компоненты доступа к данным.....	465
Область дескрипторов <i>XSQLDA</i>	467
Структура <i>XSQLVAR</i>	468
Компонент <i>TIBTable</i>	469
Компонент <i>TIBQuery</i>	470
Компонент <i>TIBStoredProc</i>	471
Компонент <i>TIBDataSet</i>	472
Компонент <i>TIBSQL</i>	474
Обработка событий	477
Информация о состоянии базы данных	479
Компонент <i>TIBDatabaseInfo</i>	479
Компонент <i>TIBSQLMonitor</i>	481
Резюме.....	482
Глава 19. Использование ADO средствами Delphi.....	483
Основы ADO.....	483
Перечислители.....	486
Объекты соединения с источниками данных.....	487
Сессия.....	487
Транзакции.....	488
Наборы рядов.....	488
Команды.....	489
Провайдеры ADO.....	490
Реализация ADO в Delphi.....	491
Компоненты ADO	491
Механизм соединения с хранилищем данных ADO	492
Компонент <i>TADOConnection</i>	492
Настройка соединения.....	493
Управление соединением	498
Доступ к связанным наборам данных и командам ADO.....	501
Объект ошибок ADO.....	503
Транзакции.....	504
Наборы данных ADO.....	505
Класс <i>TCustomADODataset</i>	506
Набор данных.....	506
Курсор набора данных	507
Локальный буфер.....	509
Состояние записи	510
Фильтрация.....	511
Поиск.....	512
Сортировка	513
Команда ADO.....	513
Групповые операции	515
Параметры.....	516
Класс <i>TParameters</i>	516
Класс <i>TParameter</i>	517
Компонент <i>TADODataSet</i>	519
Компонент <i>TADOTable</i>	520

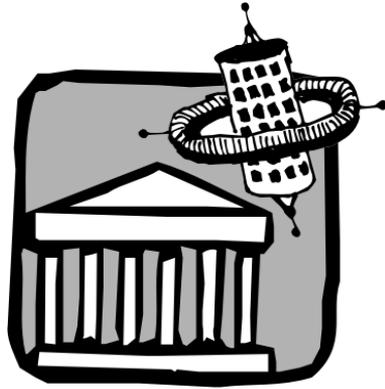
Компонент <i>TADOQuery</i>	520
Компонент <i>TADOStoredProc</i>	521
Команды ADO	521
Объект ошибок ADO.....	523
Пример приложения ADO	524
Соединение с источником данных.....	527
Групповые операции	528
Фильтрация	529
Сортировка.....	529
Резюме.....	529
ЧАСТЬ V. РАСПРЕДЕЛЕННЫЕ ПРИЛОЖЕНИЯ БАЗ ДАННЫХ	531
Глава 20. Технология DataSnap. Механизмы удаленного доступа.....	532
Структура многозвенного приложения в Delphi.....	533
Трехзвенное приложение в Delphi	535
Сервер приложений	536
Клиентское приложение.....	538
Механизм удаленного доступа к данным DataSnap	538
Компонент <i>TDCOMConnection</i>	539
Компонент <i>TSocketConnection</i>	540
Компонент <i>TWebConnection</i>	543
Провайдеры данных.....	545
Вспомогательные компоненты — брокеры соединений.....	548
Компонент <i>TSimpleObjectBroker</i>	548
Компонент <i>TLocalConnection</i>	550
Компонент <i>TSharedConnection</i>	551
Компонент <i>TConnectionBroker</i>	551
Резюме.....	552
Глава 21. Сервер приложения	553
Структура сервера приложения.....	554
Интерфейс <i>IAppServer</i>	555
Интерфейс <i>IProviderSupport</i>	558
Удаленные модули данных.....	558
Удаленный модуль данных для сервера Автоматизации	559
Дочерние удаленные модули данных	563
Регистрация сервера приложения.....	564
Пример простого сервера приложения.....	565
Главный удаленный модуль данных.....	566
Дочерний удаленный модуль данных.....	567
Регистрация сервера приложения.....	568
Резюме.....	569
Глава 22. Клиент многозвенного распределенного приложения	570
Структура клиентского приложения	571
Клиентские наборы данных.....	572

Компонент <i>TClientDataSet</i>	574
Получение данных от компонента-провайдера	575
Кэширование и редактирование данных	577
Управление запросом на сервере	579
Использование индексов	580
Сохранение набора данных в файлах	582
Работа с данными типа BLOB	582
Представление данных в формате XML	583
Агрегаты	583
Объекты-агрегаты	584
Агрегатные поля	586
Группировка и использование индексов	587
Вложенные наборы данных	587
Дополнительные свойства полей клиентского набора данных	588
Обработка ошибок	589
Пример "тонкого" клиента	592
Соединение клиента с сервером приложения	594
Наборы данных клиентского приложения	595
Резюме	596
ЧАСТЬ VI. ГЕНЕРАТОР ОТЧЕТОВ RAVE REPORTS 5.0	597
Глава 23. Компоненты Rave Reports и отчеты в приложении Delphi	598
Генератор отчетов Rave Reports 5.0	599
Компоненты Rave Reports и их назначение	600
Отчет в приложении Delphi	601
Компонент отчета <i>TRvProject</i>	602
Компонент управления отчетом <i>TRvSystem</i>	605
Резюме	610
Глава 24. Визуальная среда создания отчетов	611
Инструментарий визуальной среды создания отчетов	612
Проект отчета	614
Библиотека отчетов	615
Каталог глобальных страниц	616
Словарь просмотров данных	616
Стандартные элементы оформления и их свойства	617
Элементы для представления текста и изображений	618
Графические элементы управления	619
Штрихкоды	619
Обработка событий	620
Внешние источники данных в отчете	620
Соединение с источником данных и просмотры	621
Безопасность доступа к данным	622
Отображение данных в отчетах	623
Структурные элементы отчета	623
Элементы отображения данных	625
Резюме	626

Глава 25. Разработка, просмотр и печать отчетов.....	627
Этапы создания отчета и включение его в приложение.....	628
Простой отчет в визуальной среде Rave Reports.....	628
Нумерация страниц отчета.....	629
Использование элемента <i>FontMaster</i>	630
Добавление страниц к отчету.....	630
Отчет в приложении.....	631
Просмотр и печать отчета.....	633
Сохранение отчета во внешнем файле.....	634
Компонент <i>TRvNDRWriter</i>	635
Преобразование форматов данных.....	637
Резюме.....	638
Глава 26. Отчеты для приложений баз данных.....	639
Соединения с источниками данных в Rave Reports.....	640
Соединения с источниками данных в визуальной среде Rave Reports.....	642
Соединение через драйвер Rave Reports.....	642
Соединение через компонент приложения Delphi.....	644
Соединения с источниками данных в приложении.....	645
Компонент <i>TRvDataSetConnection</i>	645
Компоненты, использующие BDE.....	647
Компонент <i>TRvCustomConnection</i>	648
Аутентификация пользователя в отчете.....	651
Типы отчетов.....	652
Простой табличный отчет.....	652
Отчет "один-ко-многим".....	653
Группирующий отчет.....	656
Использование вычисляемых значений.....	657
Вычисляемые значения по одному источнику.....	657
Вычисляемые значения по нескольким источникам.....	659
Управляющие вычислительные элементы.....	661
Резюме.....	662
ЧАСТЬ VII. ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ.....	663
Глава 27. Стандартные технологии программирования.....	664
Интерфейс переноса Drag-and-Drop.....	664
Интерфейс присоединения Drag-and-Dock.....	669
Усовершенствованное масштабирование.....	674
Управление фокусом.....	674
Управление мышью.....	675
Ярлыки.....	680
Резюме.....	683
Глава 28. Динамические библиотеки.....	684
Проект DLL.....	685
Экспорт из DLL.....	687

Соглашения о вызовах.....	689
Директива <i>register</i>	689
Директива <i>pascal</i>	690
Директива <i>stdcall</i>	690
Директива <i>cdecl</i>	690
Директива <i>safecall</i>	690
Инициализация и завершение работы DLL.....	690
Вызов DLL.....	694
Неявный вызов.....	694
Явный вызов.....	696
Ресурсы в DLL.....	699
Использование модуля <i>Share Mem</i>	703
Резюме.....	703
Глава 29. Потоки и процессы.....	704
Обзор потоков.....	705
Потоки и процессы.....	706
Фоновые процедуры, или способ обойтись без потоков.....	707
Приоритеты потоков.....	707
Класс <i>TThread</i>	711
Пример создания многопоточного приложения в Delphi.....	715
Проблемы при синхронизации потоков.....	719
Тупики.....	719
Гонки.....	720
Средства синхронизации потоков.....	721
Событие.....	722
Взаимные исключения.....	724
Семафор.....	724
Критическая секция.....	724
Процесс. Порождение дочернего процесса.....	725
Поток.....	727
Консольный ввод.....	727
Оповещение об изменении в файловой системе.....	727
Локальные данные потока.....	729
Как избежать одновременного запуска двух копий одного приложения.....	729
Резюме.....	730
Глава 30. Многомерное представление данных.....	732
Понятие кросстаба.....	732
Взаимосвязь компонентов многомерного представления данных.....	733
Подготовка набора данных.....	735
Компонент <i>TDecisionQuery</i>	739
Компонент <i>TDecisionCube</i>	739
Компонент <i>TDecisionSource</i>	743
Отображение данных.....	744
Компонент <i>TDecisionGrid</i>	744
Компонент <i>TDecisionGraph</i>	747

Управление данными	747
Компонент <i>TDecisionPivot</i>	748
Пример многомерного представления данных	749
Резюме	751
Глава 31. Использование возможностей Shell API	752
Понятие пространства имен	752
Размещение значка приложения на <i>System Tray</i>	753
Интерфейс <i>IShellLink</i>	758
Интерфейс <i>IShellFolder</i>	760
Добавление пунктов в системное контекстное меню	767
Резюме	771
Приложение. Описание дискеты	773
Предметный указатель	776



✧ ЧАСТЬ I ✧

Объектная концепция Delphi 7

- Глава 1.** Объектно-ориентированное программирование
- Глава 2.** Библиотека визуальных компонентов VCL и ее базовые классы
- Глава 3.** Обработка исключительных ситуаций
- Глава 4.** Кроссплатформенное программирование для Linux

ГЛАВА 1



Объектно-ориентированное программирование

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

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

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

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

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

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

Объект и класс

Перед началом работы необходимо ввести основные понятия и определения.

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

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

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

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

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

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

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

Разрешено опережающее объявление классов, как в следующем примере:

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

```

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

```

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

```

var
  AMyObject: TMyObject;

```

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

Как создаются и уничтожаются объекты?

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

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

```

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

```

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

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

```

type
  TMyObject = class(TObject)
    MyField: Integer;

```

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

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

```
AMyObject.Free;
```

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

Примечание

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

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

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

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

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

Поля, свойства и методы

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

Примечание

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

События

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

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

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

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

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

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

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

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

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

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

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

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

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

```
begin
```

```
end;
```

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

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

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

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

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

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

  T2ndObject = class;
    procedure SetValue1(Sender: TObject; var AValue: Integer);
    procedure SetValue2(Sender: TObject; var AValue: Integer);
  end;

...
var
  Obj1: T1stObject;
  Obj2: T2ndObject;
begin
  Obj1 := T1stObject.Create;
  Obj2 := T2ndObject.Create;
  Obj1.OnMyEvent := Obj2.SetValue1;
  Obj1.OnMyEvent := Obj2.SetValue2;
...
end.
```

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

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

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

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

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

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

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

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

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

```
procedure TForm1.CheckBoxClick(Sender: TObject);
```

```
begin
```

```
  with TCheckBox(Sender) do
```

```
    if Checked
```

```
      then Color := Colors[Tag]
```

```
      else Color := clBtnFace;
```

```
end;
```

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

Инкапсуляция

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

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

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

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

Примечание

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

Наследование

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

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

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

Примечание

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

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

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

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

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

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

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

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

Полиморфизм

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

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

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

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

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

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

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

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

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